# Relacyjne bazy danych

Gdańsk – wykład RBD

2024

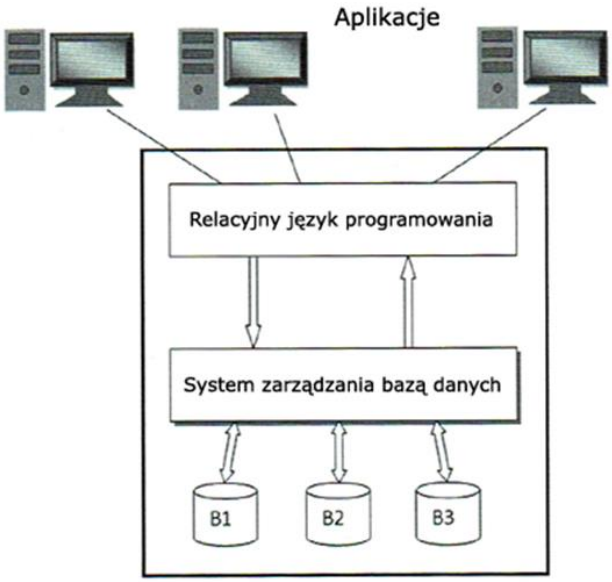
# Spis treści

# Architektura DBMS

Do obsługi baz danych tworzone są złożone systemy zawierające zbiory gotowych narzędzi zapewniających dostęp do danych. Umożliwiają one manipulowanie danymi zgromadzonymi w systemach komputerowych i aktualizowanie tych danych.

**Do najważniejszych cech SZBD (Systemu Zarządzania Bazą Danych) można zaliczyć:**
**- operowanie na dużych i bardzo dużych zbiorach danych**
**- zarządzanie złożonymi strukturami.**

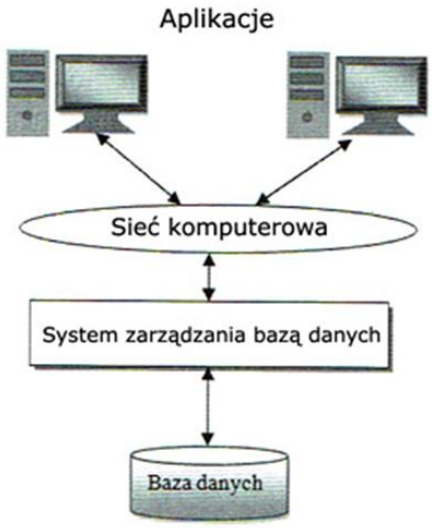
System zarządzania bazą danych wraz z bazami danych i językiem komunikowania się tworzą system baz danych. Interakcja programu użytkowego (aplikacji) z bazą danych odbywa się najczęściej za pomocą języka SQL.
Jest to jedyny sposób komunikowania się aplikacji z bazą danych.




W praktyce stosuje się dwa sposoby komunikacji z bazą danych:
- architektura klient-serwer
- architektura 3-warstwowa

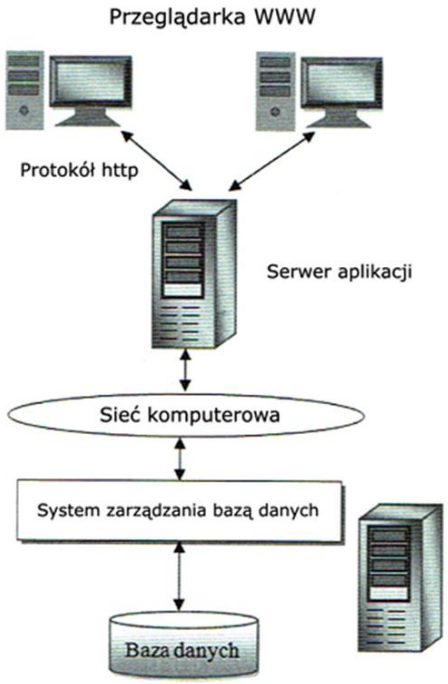
# Architektura klient-serwer

W architekturze klient-serwer aplikacje zainstalowane na stacjach użytkowników komunikują się z bazą danych, wykorzystując sieciowe oprogramowanie dedykowane do komunikacji z systemem zarządzania bazą danych.




## Architektura 3-warstwowa

W architekturze 3-warstwowej pomiędzy użytkownikami a serwerem bazy danych znajduje się tzw. serwer aplikacji, który udostępnia umieszczone na nim aplikacje. Jest to architektura typowa dla aplikacji WWW. Aplikacje są udostępniane przez serwer aplikacji w postaci stron internetowych. Użytkownik komunikuje się z bazą danych przez przeglądarkę stron WWW. W odpowiedzi na polecenia użytkownika serwer aplikacji wysyła odpowiednie żądania do systemu zarządzania bazą danych, który wykonuje polecenia i przesyła ich wyniki do serwera aplikacji. Serwer aplikacji przesyła te wyniki do aplikacji użytkownika.




## Charakterystyka / cechy baz danych

Baza danych to zbiór danych z określonej dziedziny posiadający ściśle zdefiniowaną wewnętrzną strukturę.

Baza danych powinna charakteryzować się następującymi cechami:

- ✓ **Bezpieczeństwo** danych
- ✓ **Integralność** danych
- ✓ **Trwałość** danych
- ✓ **Współdzielenie** danych
- ✓ **Integracja** danych
- ✓ **Niezależność** danych (logiczna i fizyczna)
- ✓ **Abstrakcja** danych

# Standard SQL

- Standard nieformalnie nazywany SQL/92 – pełna nazwa: Międzynarodowy Standardowy Język Baz Danych SQL (1992) – skrót od Structured Query Language;

- Istniejące implementacje nie implementują w pełni powyższego standardu – ale rozszerzają niektóre aspekty standardu;

- Język deklaratywny – użytkownik deklaruje swoje potrzeby, optymalizator przekształca zapytanie na ciąg instrukcji;

- Zawiera w sobie język definiowania danych i język manipulowania danymi;

- SQL realizuje raczej rachunek krotek niż algebrę relacji;

- Relacja nazywana jest tabelą (table), może zawierać powtórzenia i oczywiście ma ustaloną kolejność;

- Projekt bazy danych składa się głównie z zestawu tabel, są one zgrupowane w schemacie;

- Standard wymaga by wielkość liter w nazwach nie grała roli
  - zasadniczo wszystkie słowa są konwertowane na duże litery
  - PostgreSQL zamienia wszystko na małe litery
  - gra to rolę jedynie gdy występują napisy w cudzysłowach

- Standard nie określa sposobu kończenia zapytania,
  - w PostgreSQL jest to średnik

- Każdy element musi mieć nazwę, nawet gdy nie mamy zamiaru odwoływać się do niego

# Typy wbudowane

Typy wbudowane (built-in types) w kontekście baz danych to predefiniowane typy danych, które są dostarczane przez system zarządzania bazą danych (DBMS) do przechowywania różnych rodzajów danych. Obejmują one różne kategorie, takie jak tekstowe, numeryczne, daty i godziny, logiczne oraz inne.

Jest wiele typów wbudowanych, najważniejsze z nich:

– **CHAR(_), VARCHAR(_)**, pole znakowe o stałej długości, pole znakowe o zmiennej długości
– **INTEGER, SMALLINT**, wartość numeryczna liczbowa całkowita
– **DATE, TIME, TIMESTAMP**, obsługa czasu i daty
– **BOOLEAN**, wartości np. 't', TRUE, '1','y', 'yes', ( SQL/99 )
– **NUMERIC(_,_)**, liczba cyfr np. NUMERIC (7,2), 7 cyfr, w tym 2 po przecinku
– **FLOAT(_)**, liczba zmiennoprzecinkowa np. FLOAT(15), 15 cyfr znaczących
– **BIT, VARBIT**, ciąg bitów o ustalonej długości wartości np. B'10011101'
– **MONEY**, to samo co **NUMERIC** (9,2)

Typy wbudowane ułatwiają definiowanie struktury tabeli i określanie rodzaju danych, jakie mogą być przechowywane w poszczególnych kolumnach.

# Tabele

**CREATE TABLE nazwa_tabeli**
**lista-( definicja_kolumny** [ wartość_domyślna ]
| [ definicja_klucza_kandydującego ]
| [ definicja_klucza_obcego ]
| [ definicja_warunku_poprawności ] ) ;

*Wyjaśnienie:*

Powyższy SQL-owy fragment to polecenie **CREATE TABLE**, które służy do tworzenia nowej tabeli w bazie danych, gdzie można określić definicje kolumn, kluczy kandydujących, kluczy obcych lub warunków poprawności.

**CREATE TABLE nazwa_tabeli:** Tworzenie nowej tabeli za pomocą polecenia CREATE TABLE. To jest punkt wyjścia dla definiowania struktury danych.

**definicja_kolumny:** określenie kolumn w tabeli. Każda kolumna ma swoją nazwę i dziedzinę, czyli typ danych.

Na przykład, możemy mieć kolumnę "Imie" o typie VARCHAR.

**Wartość domyślna i NOT NULL:** Wartość domyślna to opcjonalny element, który określa, co dzieje się, gdy dane nie są dostarczone. NOT NULL z kolei oznacza, że kolumna musi mieć zawsze przypisaną wartość.

- Wartość domyślna może zmienić wartość podaną w definicji dziedziny, brak definicji wartości domyślnej oznacza NULL

- Można żądać, by atrybut był zawsze określony: **NOT NULL**

**Klucze i ograniczenia:** Polecenia takie jak **PRIMARY KEY** lub **FOREIGN KEY** pozwalają nam definiować klucze główne i obce, co jest kluczowe dla relacyjnego modelu danych.

**Definicja_warunku_poprawności:** Na koniec, możemy dodać dodatkowe warunki, które muszą być spełnione dla danych w tabeli. To zapewnia integrowanie i poprawność danych.

**definicja_kolumny** ::= nazwa_kolumny nazwa_dziedziny
**nazwa_dziedziny** ::= typ_wbudowany | nazwa_zdefiniowana
::= “to jest”
| “albo”
[ ] “alternatywnie”

```
CREATE TABLE Uzytkownicy (
   ID INT PRIMARY KEY,
   Imie VARCHAR(50) NOT NULL,
   Nazwisko VARCHAR(50) NOT NULL,
   Email VARCHAR(100) UNIQUE,
   DataUtworzenia DATE DEFAULT CURRENT_DATE
);
```

Kod SQL zawiera polecenie CREATE TABLE, które definiuje tabelę o nazwie "Uzytkownicy" z kolumnami ID, Imie, Nazwisko, Email i DataUtworzenia. Klucz główny jest określony dla kolumny ID, a dla kolumny Email ustawiono unikalność. Ponadto, kolumny Imie i Nazwisko są oznaczone jako NOT NULL, co oznacza, że muszą zawierać wartości.

# Klucze

Klucze w SQL są kluczowe dla utrzymania integralności danych i określania relacji między tabelami. Definiują one unikalność i relacje między kolumnami w tabelach.

**definicja_klucza_kandydującego** ::=
**UNIQUE** ( lista_kolumn ) |
**PRIMARY KEY** ( lista_kolumn )

Oto kilka kluczowych punktów dotyczących kluczy w SQL:

1. **Definicja Klucza Kandydującego:**
   - UNIQUE: Klucz kandydujący oznacza, że wartości w określonych kolumnach muszą być unikalne dla każdego wiersza w tabeli.
   - PRIMARY KEY: Klucz główny identyfikuje jednoznacznie każdy rekord w tabeli. Może być tylko jeden klucz główny w tabeli.
2. **Warunki dotyczące Listy Kolumn:**
   - Lista kolumn w definicji klucza nie może być pusta.
   - W przypadku klucza głównego, wszystkie kolumny tego klucza muszą mieć warunek poprawności NOT NULL.
3. **Obsługa Wartości NULL:**
   - Klucz alternatywny (UNIQUE) dopuszcza wartości NULL.
   - Klucz główny (PRIMARY KEY) nie dopuszcza wartości NULL.

4. **Zgodność z różnymi systemami baz danych:**
    - W PostgreSQL klucze nie naruszają warunku NOT NULL dla klucza alternatywnego.
    - W SQL92 (np. MS SQL Server) klucze naruszają warunek NOT NULL dla klucza alternatywnego.
5. **Nazwa dla Warunku Poprawności:**
    - Warunki poprawności dla kluczy można opcjonalnie nazwać, używając klauzuli CONSTRAINT nazwa definicja_klucza_kand

***Pytanie kontrolne:***

Jakie są kluczowe punkty dotyczące kluczy w SQL, a także czym jest CONSTRAINT w kontekście baz danych?

***Odpowiedź:***

Klucze w SQL są kluczowe dla utrzymania integralności danych i określania relacji między tabelami. Definiują unikalność i relacje między kolumnami w tabelach. Klucze kandydujące mogą być definiowane za pomocą klauzuli UNIQUE lub PRIMARY KEY, która oznacza, że wartości w określonych kolumnach muszą być unikalne dla każdego wiersza w tabeli. Klucz główny (PRIMARY KEY) identyfikuje jednoznacznie każdy rekord w tabeli.

Ogólne kluczowe punkty dotyczące kluczy w SQL obejmują warunki dotyczące listy kolumn, obsługę wartości NULL, zgodność z różnymi systemami baz danych oraz możliwość nadawania nazwy warunkom poprawności za pomocą klauzuli CONSTRAINT.

W kontekście baz danych, CONSTRAINT to warunek lub reguła narzucana na dane w celu utrzymania integralności bazy danych. CONSTRAINT może dotyczyć różnych aspektów, takich jak klucze główne, klucze obce, unikalność, zasady, itp. CONSTRAINTy są używane do określenia reguł, które muszą być spełnione przez dane w tabeli, co zapewnia spójność i poprawność danych. Na przykład, CONSTRAINT może narzucać, że dana kolumna musi mieć unikalne wartości lub może definiować relacje pomiędzy różnymi tabelami.

# Klucze obce

Klucze obce umożliwiają utrzymanie spójności danych poprzez określanie relacji między tabelami.

**definicja_klucza_obcego** ::=
**FOREIGN KEY** ( lista_kolumn )
**REFERENCES** tabela_bazowa [ ( lista_kolumn ) ]
[ **ON DELETE** opcja ]
[ **ON UPDATE** opcja ]

Kluczowe punkty związane z definicją klucza obcego:

1. **Definicja Klucza Obcego:**
    - FOREIGN KEY: Określa kolumny, które stanowią klucz obcy w aktualnej tabeli.
    - REFERENCES tabela_bazowa: Wskazuje na tabelę, do której odnosi się klucz obcy. Może być również określona lista kolumn, jeśli klucze nie mają tej samej nazwy.
2. **Opcje dla Operacji DELETE i UPDATE:**
    - [ON DELETE opcja]: Określa, co ma się stać, gdy rekord w tabeli bazowej zostanie usunięty. Opcje to m.in. NO ACTION, CASCADE, SET DEFAULT lub SET NULL.
    - [ON UPDATE opcja]: Podobnie jak dla DELETE, ale dotyczy aktualizacji.
    - nie jest wymagane podanie listy kolumn, jeśli klucz obcy odwołuje się do klucza o tej samej nazwie

    opcja ::= **NO ACTION** | **CASCADE** | **SET DEFAULT** | **SET NULL**

3. **Warunki Poprawności i Nazwanie:**
    - Warunki poprawności dla kluczy obcych można opcjonalnie nazwać, używając klauzuli **CONSTRAINT** nazwa **definicja_klucza_obcego**

***Pytanie kontrolne:***
Czym jest klucz obcy w kontekście baz danych, a jakie są kluczowe punkty związane z jego definicją w języku SQL?

***Odpowiedź:***
Klucz obcy w bazach danych umożliwia utrzymanie spójności danych poprzez określanie relacji między tabelami. Klucz obcy definiuje się za pomocą klauzuli FOREIGN KEY, określając kolumny stanowiące klucz obcy w aktualnej tabeli oraz wskazując tabelę bazową, do której odnosi się klucz obcy za pomocą klauzuli REFERENCES. Opcje ON DELETE i ON UPDATE pozwalają określić, co ma się stać, gdy rekord w tabeli bazowej zostanie usunięty lub zaktualizowany, oferując różne opcje, takie jak CASCADE, SET DEFAULT, SET NULL itp.
Warunki poprawności dla kluczy obcych można opcjonalnie nazwać, używając klauzuli CONSTRAINT.

# Warunki poprawności

Warunki poprawności są kluczowym narzędziem dla zapewnienia spójności, integralności danych w bazie SQL. Ich precyzyjne definiowanie pozwala na kontrolowanie i zabezpieczanie danych przed wprowadzeniem nieprawidłowych informacji.

**Kluczowe punkty związane z definicją warunków poprawności:**

1. **Definicja Warunku Poprawności:**

   definicja_warunku_poprawności ::=
   **CHECK** ( wyrażenie_warunkowe )

   - Określa warunek, który musi być spełniony dla każdego wiersza tabeli.
2. **Własności Wyrażenia Warunkowego:**
   - wyrażenie_warunkowe: Może być dowolnie skomplikowane i nie musi ograniczać się do jednej tabeli. Musi być określone dla każdego wiersza tabeli.
3. **Spełnianie Warunków Poprawności:**
   - Więzy poprawności są spełnione, jeśli wyrażenie_warunkowe ma wartość "true" dla każdego wiersza tabeli.
4. **Zachowanie Systemu Zarządzania Bazą Danych (DBMS):**
   - System zarządzania bazą danych nie zezwala na wprowadzenie lub aktualizację danych, które naruszają warunki poprawności.
   - Kolejność sprawdzania warunków jest nieokreślona.
5. **Nazwanie Warunków Poprawności:**
   - Warunki poprawności można opcjonalnie nazwać, co pozwala na identyfikację konkretnych warunków w przypadku potrzeby.

     **CONSTRAINT** nazwa **definicja_warunku_poprawności**

# Zmiana definicji tabeli podstawowej

W języku SQL, polecenie ALTER TABLE jest kluczowe do dostosowywania struktury tabeli do bieżących potrzeb. Pozwala ono na różne operacje na tabeli, umożliwiając dynamiczną modyfikację jej definicji.

Oto kilka kluczowych operacji związanych z ALTER TABLE:

**ALTER TABLE** klient

**ADD COLUMN** rabat **INT**

**DEFAULT** 0;

**Operacje mogą obejmować:**

1. **Dodanie Kolumny:**
   - *Przykład:*

**ALTER TABLE** klient
**ADD COLUMN** rabat **INT**
DEFAULT 0;

Dodajemy nową kolumnę "rabat" do tabeli "klient" o typie INT i wartości domyślnej ustawionej na 0.

2. **Zmiana Kolumny:**
    - *Przykład:*

**ALTER TABLE** klient
**ALTER COLUMN** telefon **DROP NOT NULL**;

Usuwamy ograniczenie NOT NULL dla kolumny "telefon" w tabeli "klient".

3. **Zmiana Nazwy Kolumny:**
    - *Przykład:*

**ALTER TABLE** klient
**RENAME COLUMN** stara_nazwa **TO** nowa_nazwa;

Zmieniamy nazwę kolumny z "stara_nazwa" na "nowa_nazwa" w tabeli "klient".

4. **Zmiana Wartości Domyślnej:**
    - *Przykład:*

**ALTER TABLE** klient
**ALTER COLUMN** rabat **SET DEFAULT** 10;

Zmieniamy wartość domyślną kolumny "rabat" na 10 w tabeli "klient".

5. **Usunięcie Kolumny:**
    - *Przykład:*

**ALTER TABLE** klient
**DROP COLUMN** rabat;

Usuwamy kolumnę "rabat" z tabeli "klient".

6. **Dodanie lub Usunięcie Warunku Poprawności:**
    - *Przykład (dodanie warunku):*

**ALTER TABLE** klient
**ADD CONSTRAINT** nazwa_warunku **CHECK** (warunek);

Dodajemy warunek poprawności do tabeli "klient".

- *Przykład (dodanie warunku):*

**ALTER TABLE** klient

**ADD CONSTRAINT** check_rabat_positive **CHECK** (rabat >= 0);

W tym przykładzie dodajemy warunek poprawności do tabeli "klient", który wymusza, aby wartość w kolumnie "rabat" była większa lub równa 0. Ten warunek poprawności zapewni, że żadna wartość rabatu nie będzie ujemna.

- *Przykład (usunięcie warunku):*

**ALTER TABLE** klient
**DROP CONSTRAINT** nazwa_warunku;

Usuwamy warunek poprawności z tabeli "klient".

- *Przykład (usunięcie warunku):*

**ALTER TABLE** klient

**DROP CONSTRAINT** check_rabat_positive;

W tym przykładzie usuwamy warunek poprawności o nazwie "check_rabat_positive" z tabeli "klient". Zakładamy, że ten warunek wcześniej narzucał, aby wartość w kolumnie "rabat" była większa lub równa 0. Usunięcie tego warunku oznacza, że dane w kolumnie "rabat" nie muszą już spełniać tego warunku, co może umożliwić wprowadzanie wartości ujemnych.

# Usuwanie tabeli podstawowej

Polecenie **DROP TABLE** pozwala na usunięcie tabeli z bazy danych wraz z jej zawartością, z możliwością wyboru opcji **RESTRICT** lub **CASCADE** w zależności od potrzeb i zależności między obiektami w bazie danych.

**Składnia Polecenia:**

**DROP TABLE** nazwa_tabeli [ **RESTRICT** | **CASCADE** ];

**Opis krok po kroku:**

- **DROP TABLE**: To polecenie wskazuje, że chcemy usunąć tabelę.
- **nazwa_tabeli**: Określa nazwę tabeli, którą chcemy usunąć.
- [ **RESTRICT** | **CASCADE** ]: To opcjonalna klauzula, która określa, jakie działania mają być podjęte w zależności od istnienia powiązań (np. perspektyw) i więzów poprawności.
    - **RESTRICT**: Jeśli wybrano tę opcję, to instrukcja **DROP TABLE** nie powiedzie się, jeśli tabela jest używana w jakiejkolwiek definicji perspektywy.
    - **CASCADE**: Jeśli wybrano tę opcję, to instrukcja **DROP TABLE** powiedzie się i usunie daną tabelę wraz ze wszystkimi perspektywami bazującymi na tej tabeli oraz więzami poprawności.

**Przykłady:**

-- *Przykład z użyciem RESTRICT*

**DROP TABLE** produkt **RESTRICT**;

-- Instrukcja nie powiedzie się, jeśli tabela "produkt" jest używana w jakiejkolwiek definicji perspektywy.

-- *Przykład z użyciem CASCADE*

**DROP TABLE** zamowienie **CASCADE**;

-- Instrukcja powiedzie się, a tabela "zamowienie" zostanie usunięta razem ze wszystkimi perspektywami bazującymi na tej tabeli oraz więzami poprawności.

# CRUD

Operacje CRUD (Create, Read, Update, Delete) to zestaw podstawowych operacji, które umożliwiają manipulację danymi w bazie SQL, obejmujące tworzenie, odczyt, aktualizację i usuwanie danych.

Poniżej przedstawiono krótki opis każdej z tych operacji w kontekście SQL:

1. **CREATE (INSERT):**
    - INSERT: Operacja tworzenia lub wstawiania danych do tabeli.
    - Przykład:

**INSERT INTO** tabela (kolumna1, kolumna2) **VALUES** (wartosc1, wartosc2);

2. **READ (SELECT):**
    - SELECT: Główna operacja odczytu, umożliwiająca wyszukiwanie danych w bazie danych.
    - Przykład:

**SELECT** kolumna1, kolumna2 **FROM** tabela **WHERE** warunek;

3. **UPDATE:**
    - UPDATE: Operacja aktualizacji, pozwalająca na zmianę wartości istniejących danych w tabeli.
    - Przykład:

**UPDATE** tabela **SET** kolumna1 = nowa_wartosc **WHERE** warunek;

4. **DELETE:**
    - DELETE: Operacja usuwania danych z tabeli.
    - Przykład:

**DELETE FROM** tabela **WHERE** warunek;

**Dodatkowe informacje:**

**SELECT** – główna operacja wyszukiwania danych, realizuje zmianę nazwy, obcięcie, rzut i złączenie relacji
**INSERT** – realizuje aktualizację /wstawianie danych
**UPDATE** – realizuje aktualizację /zmianę wartości danych
**DELETE** – realizuje aktualizację /usuwanie danych

# INSERT

Polecenie **INSERT INTO** w SQL służy do wstawiania nowych danych do tabeli.

Ogólna struktura polecenia oraz przykłady jego użycia:

**INSERT INTO** cel ( lista_elementów ) źródło;

- cel: Nazwa tabeli, do której wstawiamy dane.
- lista_elementów: Lista nazw atrybutów, którym chcemy nadać wartość.
- źródło: Ma jedną z dwóch postaci:

**VALUES** (lista_wartości)

*Przykład*

**INSERT INTO** kod_kreskowy **VALUES** ('4892840112975', 17)

- Wstawia jeden wiersz do tabeli kod_kreskowy.
- Nadaje wartości atrybutom zadeklarowanym w definicji tabeli, w kolejności deklaracji.
- Wartości '4892840112975' i 17 są przypisywane odpowiednio pierwszemu i drugiemu atrybutowi tabeli kod_kreskowy.
- W przypadku tabeli kod_kreskowy, przyjmuje się, że pierwszy atrybut to potencjalny kod kreskowy, a drugi to liczba.
- Nie można opuścić żadnego z atrybutów.
- Muszą być dostarczone wartości dla wszystkich atrybutów tabeli.

**INSERT INTO** towar ( opis, koszt ) **VALUES** ( 'donica duża', 26.43 ), ('donica mała', 13.36)

- Wstawia dwa wiersze do tabeli towar.
- Określa wartości dla konkretnych atrybutów (opis i koszt) przy użyciu klauzuli VALUES.
- Każdy nawias okrągły zawiera zestaw wartości dla pojedynczego wiersza.
- Wartości przypisywane są atrybutom zgodnie z kolejnością ich podania, tj. 'donica duża' dla opis i 26.43 dla koszt w pierwszym wierszu, oraz 'donica mała' i 13.36 w drugim wierszu.

# “Chwilówki” - Tymczasowe Tabele w SQL

Tymczasowe tabele są usuwane automatycznie po zakończeniu sesji, co oznacza, że istnieją tylko przez czas trwania bieżącej sesji.

- INSERT INTO dla tymczasowej tabeli na podstawie warunku:

**INSERT INTO** chwilowa
**SELECT** imie, nazwisko, ulica_dom
**FROM** klient
**WHERE** miasto = 'Gdańsk';

Wstawia dane z kolumn imie, nazwisko, ulica_dom z tabeli klient do tymczasowej tabeli chwilowa tylko dla rekordów, gdzie miasto to 'Gdańsk'.

- Tworzenie Tymczasowej Tabeli:

**CREATE TEMP TABLE** chwilowa ( imię varchar(11), …

Tworzy tymczasową tabelę o nazwie chwilowa zdefiniowaną z kolumnami, takimi jak imię, której dane są związane z sesją i zostaną usunięte po jej zakończeniu.

- INSERT INTO dla Tabeli towar z Wartością Nieokreśloną:

**INSERT INTO** towar ( opis, koszt, cena )
**VALUES** ( E'ramka do fotografii 3\'x4\'', 13.36, NULL )

Wstawia nowy wiersz do tabeli towar z danymi:

- ✓ Kolumna opis o wartości 'ramka do fotografii 3'x4'' (znak ukośnika \ jest używany do uniknięcia problemów z interpretacją apostrofu).
- ✓ Kolumna koszt o wartości 13.36.
- ✓ Kolumna cena o wartości nieokreślonej (NULL).

# UPDATE

Polecenie UPDATE aktualizuje dane w tabeli zgodnie z określonymi warunkami.

Oto krótka charakteryzacja:

- **UPDATE** cel **SET** element = wartość **WHERE** warunek;
  - **cel** to nazwa tabeli, w której dokonywane są aktualizacje.
  - **element** to nazwa kolumny, której wartość zostanie zaktualizowana.
  - **wartość** to nowa wartość przypisywana do określonej kolumny.
  - klauzula **WHERE** wyznacza wiersze, w których będzie dokonana aktualizacja. Ma ona identyczne znaczenie jak w instrukcji SELECT Jeśli nie jest podana, aktualizacja zostanie wykonana dla wszystkich wierszy tabeli.

- SQL nie przewiduje możliwości aktualizacji kilku atrybutów w jednym poleceniu. Niektóre implementacje dopuszczają taką możliwość.

Przykłady:

1. **UPDATE** towar **SET** cena = 1.15 **WHERE** nr=5;
    - Aktualizuje wartość w kolumnie cena na 1.15 dla pojedynczego wiersza, gdzie klucz główny (nr) wynosi 5.
2. **UPDATE** towar **SET** cena = cena*1.15 **WHERE** opis LIKE '%układanka%';
    - Aktualizuje wartość w kolumnie cena dla wszystkich (wielu jednocześnie) wierszy, których opis zawiera frazę 'układanka', mnożąc obecną wartość cena przez 1.15.
3. **UPDATE** towar **SET** cena = (**SELECT** cena **FROM** towar **WHERE** nr=5);
    - Aktualizuje wartość w kolumnie cena dla wszystkich wierszy, przypisując wartość kolumny cena z wiersza, gdzie klucz główny (nr) wynosi 5.
    - Zastosowanie podzapytania, które działa jak pojedyncza wartość, jest istotne, gdy warunek WHERE w podzapytaniu odnosi się do wartości klucza.
    - Tabela 1x1 występuje w roli pojedynczej wartości (gdyby warunek WHERE w zagnieżdżonym zapytaniu nie odwoływał się do wartości kluczowej, polecenie UPDATE mogłoby produkować błąd).
4. **UPDATE** towar **SET** cena = cena*1.15;
    - Aktualizuje wartość w kolumnie cena dla wszystkich wierszy w tabeli, mnożąc obecną wartość cena przez 1.15. Bez warunku **WHERE**, operacja dotyczy całej tabeli.

# DELETE

Polecenie DELETE w SQL jest używane do usuwania danych z tabeli zgodnie z określonymi warunkami.

Oto krótka charakterystyka:

- **DELETE FROM** cel **WHERE** warunek;
    - **cel** to nazwa tabeli, z której usuwane są dane.
    - Klauzula **WHERE** wyznacza warunki, na podstawie których dokonywane jest usuwanie. Jeśli nie jest podana, wszystkie wiersze są usuwane.
- PostgreSQL i inne implementacje pozwalają na nieodwołalne usunięcie całej zawartości tabeli:
    - **TRUNCATE TABLE** cel;

- Usuwa wszystkie wiersze z tabeli, ale pozostawia samą strukturę tabeli.

- Usuwanie tabeli (usunięcie całej tabeli):
    - **DROP TABLE** cel;
        - Usuwa zarówno dane, jak i strukturę tabeli. Uwaga: To nie jest tożsame z usunięciem danych z tabeli.

Przykłady:

**1.**

**DELETE FROM** klient **WHERE** miasto = 'Gdańsk';

- Usuwa wszystkie wiersze z tabeli klient, gdzie wartość w kolumnie miasto wynosi 'Gdańsk'.

2.

```
DELETE FROM zamowienie Z
WHERE (
SELECT miasto
FROM klient K
WHERE K.nr = Z.klient_nr
) = 'Gdynia'
```

- Usuwa informacje o zamówieniach składanych przez klientów z Gdyni, używając zagnieżdżonego zapytania w klauzuli WHERE.

# SELECT

Polecenie SELECT w SQL jest kluczowym narzędziem do pobierania danych z bazy danych.

Oto krótka charakterystyka:

- **SELECT [ ALL | DISTINCT ]** lista_atrybutów_wynikowych [ lista_klauzul ];
    - **ALL** lub **DISTINCT** określa, czy wyniki mają zawierać duplikaty czy nie.
    - **lista_atrybutów_wynikowych** obejmuje kolumny, na których mają być wykonane operacje, takie jak rzut i zmiana nazwy kolumny. Nie może być pusta.
    - **lista_klauzul** obejmuje klauzule takie jak FROM, WHERE, ORDER BY, GROUP BY, HAVING, które wpływają na rezultat zapytania.

*Przykład:*

```
SELECT DISTINCT imie, nazwisko
        -- rzut na atrybuty
FROM klient
WHERE miasto = 'Gdańsk';
```

-- obcięcie wyników do wierszy spełniających warunek

Powyższe zapytanie wybiera unikalne kombinacje imion i nazwisk z tabeli klient, gdzie miasto jest równe 'Gdańsk'. Klauzula DISTINCT eliminuje duplikaty, a WHERE filtruje wyniki na podstawie warunku.

# Lista atrybutów

Atrybut wynikowy w poleceniu SELECT SQL może być reprezentowany jako gwiazdka *, co oznacza wszystkie atrybuty, lub jako wyrażenie skalarne, zazwyczaj będące nazwą pojedynczego atrybutu.

Może także zawierać opcjonalną klauzulę AS, która pozwala na nadanie atrybutowi wynikowemu nowej nazwy.

Kilka kluczowych punktów związanych z atrybutami wynikowymi to:

- **Gwiazdka (*):**

**SELECT * FROM** towar;

wyświetla wszystkie atrybuty tabeli towarów.

- **Wyrażenie skalarne:**

**SELECT** imie, nazwisko **FROM** klient;

wybiera tylko atrybuty imie i nazwisko z tabeli klient.

- **DISTINCT:**
    - Klauzula **DISTINCT**

usuwa powtarzające się wiersze w tabeli wynikowej. Domyślnie jest stosowana opcja ALL, która zwraca wszystkie wyniki, w tym duplikaty.

    - **SELECT DISTINCT** imie **FROM** klient;

zwraca unikalne wartości atrybutu imie z tabeli klient.

- **Klauzula AS:**

**SELECT** imie **AS** first_name, nazwisko **AS** last_name **FROM** klient;

nadaje nowe nazwy kolumnom w wynikach zapytania.

- **Porządkowanie wyników:**

Niektóre implementacje SQL automatycznie sortują wyniki, ale nie jest to standardowe zachowanie.

*Przykład:*

```
-- Wybór wszystkich atrybutów z tabeli towar
SELECT * FROM towar;
```

-- Wybór unikalnych imion z tabeli klient
SELECT DISTINCT imie FROM klient;

-- Wybór imion i nazwisk z tabeli klient z nadaniem nowych nazw kolumn
SELECT imie AS first_name, nazwisko AS last_name FROM klient;

# FROM

Klauzula FROM w poleceniu SELECT służy do określania źródła danych, czyli tabel lub wyników zapytań, z których pobierane są dane.

Klauzula **FROM:**

**FROM** lista_tabel

- Lista_tabel nie może być pusta.
- Wynikiem jest iloczyn kartezjański tabel.

**SELECT** * **FROM** klient

- Jedna tabela, iloczyn równy tej tabeli.
- Wynikiem jest cała tabela klient.

**SELECT * FROM** towar, kod_kreskowy

- Iloczyn kartezjański dwóch tabel.
- Wynikiem jest iloczyn kartezjański tabel towar i kod_kreskowy.

**SELECT * FROM** klient, towar
– w obu tabelach występuje atrybut „nr”, czysto przypadkowa zbieżność
– podając nazwę atrybutu, w przypadku takiej zbieżności, trzeba dodać nazwę tabeli

**SELECT * FROM** klient, towar **WHERE** klient.nr = towar.nr;
- W przypadku zbieżności nazw atrybutów "nr" w tabelach klient i towar, konieczne jest jednoznaczne określenie, z której tabeli pochodzi dany atrybut.

# WHERE

Klauzula WHERE w poleceniu SELECT służy do filtrowania wyników na podstawie określonego warunku.

Klauzula **WHERE**:

**WHERE** wyrażenie_warunkowe

- występuje po klauzuli FROM
- wynikiem jest wybór tych wierszy, które spełniają warunek

Warunek:

- równość, nierówność itp. na atrybutach
- należenie atrybutu do zbioru (tabela 1 kolumnowa)
- operacje logiczne na prostszych warunkach

*Klauzula nie musi występować, wówczas wybrane są wszystkie wiersze tabeli*

1. **Podstawowe użycie:**

**SELECT * FROM** klient **WHERE** miasto = 'Gdańsk';

- Wynikiem jest wybór wszystkich wierszy z tabeli klient, gdzie wartość atrybutu "miasto" jest równa 'Gdańsk'.
- obcięcie relacji w/g warunku miasto = 'Gdańsk'

2. **Warunki logiczne:**

**SELECT * FROM** zamowienie **WHERE** status = 'Złożone' **AND** cena > 100;

- Wyszukuje zamówienia, które mają status 'Złożone' i cena większa niż 100.

3. **Negacja:**

**SELECT * FROM** klient **WHERE NOT** miasto = 'Gdańsk';

- Wybiera wiersze, gdzie miasto nie jest 'Gdańsk'.

4. **Warunki na podstawie nierówności:**

**SELECT * FROM** towar **WHERE** cena **BETWEEN** 10 **AND** 50;

- Wybiera towary, których cena mieści się w zakresie od 10 do 50.

# SELECT – złączanie (JOIN)

Złączenie w SQL jest procesem wyboru pasujących wierszy w iloczynie kartezjańskim dwóch tabel.

Kilka kluczowych punktów dotyczących złączeń to:

1. **Podstawowe złączenie:**

**SELECT** klient.nr, nazwisko, imie, data_zlozenia
**FROM** klient, zamowienie **WHERE** klient.nr = klient_nr;

- Wynikiem jest iloczyn kartezjański tabel **klient** i **zamowienie**, gdzie numery klientów są sobie równe.

2. **Unikanie niejednoznaczności:**

- Bez warunku **WHERE** byłby iloczyn kartezjański, obejmujący wszystkie pary wierszy z obu tabel. Warunek **WHERE** jest kluczowy do sprecyzowania, które wiersze mają zostać połączone.

3. **Użycie aliasów dla nazw tabel:**

   **SELECT** K.nr, nazwisko, imie, data_zlozenia
   **FROM** klient K, zamowienie **WHERE** K.nr = klient_nr;

   - Przy użyciu aliasu **K** dla tabeli **klient**, unikamy niejednoznaczności w przypadku, gdy obie tabele mają atrybut o tej samej nazwie, takiej jak "nr". Alias pozwala jednoznacznie odwołać się do atrybutu z konkretnej tabeli.

4. **Stosowanie aliasów w złączeniach wielokrotnych:**

   - W przypadku złączeń wielokrotnych, gdzie jedna tabela jest używana wielokrotnie, aliasy ułatwiają czytelność zapytania.

Zastosowanie aliasów dla nazw tabel jest praktyczne, zwłaszcza gdy w zapytaniu używane są złączenia lub gdy wiele tabel ma atrybuty o tych samych nazwach. Aliasy pomagają uniknąć niejednoznaczności i sprawiają, że zapytania są bardziej czytelne.

## *SELECT - złączanie przykład*

| nr | nazwisko | imie |
|---|---|---|
| 3 | Szczęsna | Jadwiga |
| 4 | Łukowski | Bernard |
| 5 | Soroczyński | Jan |
| 6 | Niezabitowska-Nasiadko | Marzena |
| 7 | Kołak | Agnieszka |
| 8 | Kołak | Agnieszka |

| klient_nr | data_zlozenia |
|---|---|
| 3 | 21.02.2021 |
| 3 | 23.03.2021 |
| 3 | 13.03.2021 |
| 5 | 4.05.2021 |
| 6 | 1.02.2021 |
| 6 | 22.03.2021 |
| 8 | 7.04.2021 |
| 8 | 12.01.2021 |

| nr | nazwisko | imie | data_zlozenia |
|---|---|---|---|
| 3 | Szczęsna | Jadwiga | 21.02.2021 |
| 3 | Szczęsna | Jadwiga | 23.03.2021 |
| 3 | Szczęsna | Jadwiga | 13.03.2021 |
| 5 | Soroczyński | Jan | 4.05.2021 |
| 6 | Niezabitowska-Nasiadko | Marzena | 1.02.2021 |
| 6 | Niezabitowska-Nasiadko | Marzena | 22.03.2021 |
| 8 | Kołak | Agnieszka | 7.04.2021 |
| 8 | Kołak | Agnieszka | 12.01.2021 |

# SELECT - złączanie INNER JOIN

Składnia INNER JOIN w poleceniu SELECT umożliwia bardziej czytelne i bezpośrednie zapisanie operacji złączenia.

Inna składnia na złączenie:

**SELECT** K.nr, nazwisko, imie, data_zlozenia
**FROM** klient K **INNER JOIN** zamowienie **ON** K.nr = klient_nr;

- W rezultacie zapytanie zwróci kolumny **nr**, **nazwisko**, **imie** oraz **data_zlozenia** dla wierszy, które spełniają warunek złączenia między tabelami **klient** i **zamowienie**. Zastosowanie zaliasowanej tabeli (**K**) ułatwia odwoływanie się do kolumn z konkretnej tabeli, szczególnie w przypadku, gdy obie tabele mają atrybuty o tej samej nazwie.

Poniżej znajduje się opis poszczególnych elementów zapytania:

Zapytanie SQL **SELECT K.nr, nazwisko, imie, data_zlozenia FROM klient K INNER JOIN zamowienie ON K.nr = klient_nr;** wykonuje operację złączenia (**INNER JOIN**) między tabelą **klient** (zaliasowaną jako **K**) a tabelą **zamowienie**. Poniżej znajduje się opis poszczególnych elementów zapytania:

1. **SELECT K.nr, nazwisko, imie, data_zlozenia:**
    - Określa, które kolumny mają być wyświetlone w wynikach zapytania. Wybierane są kolumny z zaliasowanej tabeli klient (**K**), takie jak **nr**, **nazwisko**, **imie** i **data_zlozenia**.
2. **FROM klient K INNER JOIN zamowienie ON K.nr = klient_nr:**
    - Określa źródło danych, z którego pobierane są informacje. Zaliasowana tabela klient (**K**) jest używana jako pierwsza tabela.
    - Klauzula **INNER JOIN** służy do połączenia tabeli **klient** z tabelą **zamowienie**.
    - **ON K.nr = klient_nr** to warunek złączenia, który określa, jakie kolumny w obu tabelach mają mieć równoważne wartości. W tym przypadku chodzi o równość między atrybutem **nr** w tabeli **klient** a atrybutem **klient_nr** w tabeli **zamowienie**.

SELECT - złączanie INNER JOIN:
- bezpośrednie odwołanie się do operacji złączenia w algebrze relacyjnej
- deklaracja atrybutu klient_nr jako klucza obcego wskazującego na klient(nr) nie zwalnia z obowiązku napisania jawnego warunku dla złączenia
- słowo kluczowe **INNER** jest domyślne (będą inne złączenia)

# SELECT - atrybuty wynikowe

W zapytaniu SQL, klauzula SELECT służy do określenia atrybutów wynikowych i ewentualnych dodatkowych obliczeń, które mają być uwzględnione w wynikach.

Poniżej znajduje się opis i przykłady zastosowania tej klauzuli:

1. **Atrybuty Wynikowe:**
    - W zwykłych przypadkach, atrybuty wynikowe wskazują, które kolumny z tabeli mają być uwzględnione w wynikach zapytania.
    - Przykład: **SELECT** *, cena - koszt **AS** zysk **FROM** towar

- Dodaje nową kolumnę o nazwie **zysk**, która zawiera wynik obliczenia różnicy między kolumnami **cena** i **koszt**.
- Wszystkie pozostałe kolumny z tabeli **towar** są również uwzględnione w wynikach.

| **nr** | **opis** | | | **koszt** | **cena** | **zysk** |
|---|---|---|---|---|---|---|
| 1 | układanka drewniana | 15,23 | 21,95 | 6,72 | | |
| 2 | układanka typu puzzle | 16,43 | 19,99 | 3,56 | | |
| 3 | kostka Rubika | | 7,45 | 11,49 | 4,04 | |
| 4 | Linux CD | | 1,99 | 2,49 | 0,50 | |
| 5 | chusteczki higieniczne | 2,11 | 3,99 | 1,88 | | |
| 6 | ramka do fotografii 4'x6' | 7,54 | 9,95 | 2,41 | | |

- wyrażenie_skalarne może odwoływać się do nazw atrybutów, ale zawierać dodatkowe obliczenia
- nazwa_kolumny będzie nazwą kolumny w tabeli wynikowej

2. **Przykład bardziej złożonego wyrażenia:**

```
SELECT *, cena - koszt AS zysk,
CASE WHEN (cena - koszt) / koszt < 0 THEN 'ujemny'
WHEN (cena - koszt) / koszt < 0.4 THEN 'za mało'
WHEN cena IS NULL THEN 'brak danych'
ELSE 'ok'
END AS opinia
FROM towar
```

- Dodaje kolumnę **opinia**, która na podstawie różnicy między ceną a kosztem przypisuje opinie, takie jak 'ujemny', 'za mało', 'brak danych' lub 'ok'.

| **nr** | **opis** | **koszt** | **cena** | **zysk** | **opinia** |
|---|---|---|---|---|---|
| 8 | ramka do fotografii 3'x4' | 13,36 | 19,95 | 6,59 | ok |
| 9 | szczotka do zębów | 0,75 | 1,45 | 0,70 | ok |
| 10 | moneta srebrna z Papieżem | 20,00 | 20,00 | 0,00 | za mało |
| 11 | torba plastikowa | 0,01 | 0,00 | -0,01 | ujemny |
| 12 | głośniki | 19,73 | 25,32 | 5,59 | za mało |
| 13 | nożyczki drewniane | 8,18 | | | brak danych |
| 14 | kompas wielofunkcyjny | 22,10 | | | brak danych |

3. **Dodatkowe Obliczenia Bezpośrednio w Zapytaniu:**
   - Możliwość wykonania obliczeń w wyrażeniach skalarnych nie jest ograniczona do atrybutów z tabel. Można wykonywać bardziej zaawansowane operacje matematyczne, funkcje czy uzyskiwać informacje o systemie bazodanowym.

- Przykłady:

**SELECT** 2 + 2; -- Zwraca 4
**SELECT** NOW(); -- Zwraca aktualną datę i godzinę
**SELECT** VERSION(); -- Zwraca informacje o wersji systemu zarządzania bazą danych

version
---
PostgreSQL 10.12 (Ubuntu 10.12-0ubuntu0.18.04.1) on x86_64-pc-linuxgnu, compiled by gcc (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0, 64-bit
(1 row)

Warto zauważyć, że tabela wynikowa w ostatnim przypadku w ogóle nie odwołuje się do żadnej relacji, co pozwala na wykonanie ogólnych obliczeń, które nie są związane z konkretną tabelą.

# SELECT - warunki WHERE

Klauzula WHERE w poleceniu SELECT służy do filtrowania wyników na podstawie określonych warunków.

Poniżej przedstawione są przykłady zastosowania tej klauzuli:

1. **Przykład z Warunkiem Należenia do Zbioru:**

   - Podaj nazwiska klientów spoza Trójmiasta:

**SELECT** nazwisko **FROM** klient
**WHERE** miasto **NOT IN** ('Gdańsk', 'Gdynia', 'Sopot');

   - Warunek **NOT IN** sprawdza, czy wartość atrybutu **miasto** nie należy do podanego zbioru.

2. **Przykład z Dopasowaniem Wzorca Tekstowego:**

   - Znajdź opisy wszystkich ramek do fotografii o podanym wymiarze w calach:

**SELECT** opis **FROM** towar
**WHERE** opis **LIKE** 'ramka%' **AND** opis **LIKE E** '%\'';

   - Warunek **LIKE** zastosowany jest do dopasowania wzorca tekstowego, a **%** oznacza dowolny ciąg znaków.

3. **Przykład z Warunkiem Dla Zakresu Dat:**

   - Wyświetl szczegóły zamówień złożonych w lutym 2021:

**SELECT * FROM** zamowienie
**WHERE** data_zlozenia **BETWEEN** '2021-02-01' **AND** '2021-02-29';

   - Warunek **BETWEEN** używany jest do określenia zakresu dat.

Przykłady:

- **WHERE cena > 3.14** - warunek dla pojedynczej wartości.
- **WHERE koszt >= ALL (SELECT koszt FROM towar)** - warunek dla relacji zbioru.
- **WHERE NOT EXISTS (SELECT * ...)** - warunek dla istnienia elementów.
- **WHERE NOT klient_nr MATCH UNIQUE (SELECT nr FROM klient)** - warunek dla jednoznaczności elementów.

W ostatnim przykładzie, warunek sprawdza, czy atrybut **klient_nr** w tabeli **zamowienie** nie ma jednoznaczności w stosunku do atrybutu **nr** w tabeli **klient**.

## SELECT - wyrażenia warunkowe dla WHERE

Możliwość stosowania różnych wyrażeń warunkowych, takich jak pojedyncze wartości, relacje, istnienie elementów, jednoznaczność itp.

Przykłady:

1. **Porównanie z Pojedynczymi Wartościami:**

   **WHERE** cena > 3.14;

   - Wyrażenie warunkowe porównujące cenę do określonej wartości.

2. **Relacja z Zbiorem Wartości:**

   **WHERE** miasto **NOT IN** ('Gdańsk', 'Gdynia', 'Sopot')
   **WHERE** koszt **>= ALL** (**SELECT** koszt **FROM** towar);

   - Zastosowanie operatora NOT IN do sprawdzenia, czy miasto klienta nie należy do określonego zbioru.
   - Użycie >= ALL do porównania, czy koszt jest większy lub równy każdej wartości z podzapytania.

3. **Istnienie Elementów:**

   **WHERE NOT EXISTS (SELECT * FROM** tabela **WHERE** warunek);

   - Warunek sprawdzający, czy nie istnieją żadne rekordy spełniające warunki w podzapytaniu.

4. **Jednoznaczność Elementów:**

   **WHERE NOT** klient_nr **MATCH UNIQUE (SELECT** nr **FROM** klient);

   - Wykorzystanie MATCH UNIQUE do sprawdzenia jednoznaczności wartości w kolumnie klient_nr w porównaniu do wartości w podzapytaniu.

- W przypadku, gdy nr jest kluczem w tabeli klientów, takie wyrażenie nie powinno się zdarzyć.

# SELECT - klauzula ORDER BY

Klauzula ORDER BY w poleceniu SELECT służy do sortowania wyników zapytania według określonych kolumn.

Oto kilka kluczowych punktów dotyczących tej klauzuli:

- **Składnia:**

**ORDER BY** lista_kolumn [ **DESC** | **ASC** ];

- **Występowanie:**

Klauzula **ORDER BY** pojawia się po klauzulach **FROM** i **WHERE**.

- **Domyślna Kolejność:**

Kolejność sortowania jest domyślnie rosnąca (**ASC**), ale można ją zmienić na malejącą (**DESC**).

- **Przykład:**

**SELECT * FROM** towar **ORDER BY** koszt **DESC**;

To zapytanie wyświetla tabelę **towar** uporządkowaną według atrybutu **koszt** malejąco, zaczynając od największych wartości.

- **Ograniczenie Wyników:**

**SELECT * FROM** towar **ORDER BY** koszt **DESC LIMIT** 3;

Dodatkowa opcja **LIMIT** pozwala ograniczyć wyświetlanie do określonej liczby wierszy. W tym przypadku, zwrócone zostaną tylko trzy najdroższe produkty.
Klauzula **ORDER BY** jest użyteczna do manipulowania kolejnością wyników zapytania, co umożliwia bardziej czytelne prezentowanie danych.

# SELECT - funkcje agregujące

Funkcje agregujące w poleceniu SELECT pozwalają na obliczenia dla wielu lub wszystkich wierszy tabeli. Funkcje agregujące są używane do przetwarzania danych w tabelach i generowania statystyk, co umożliwia analizę informacji w bardziej zaawansowany sposób.

Oto kilka kluczowych punktów dotyczących funkcji agregujących:

- **Składnia:**

**SELECT** funkcja_agregująca(wyrażenie) **FROM** tabela;

- wyrażenie_skalarne w części **SELECT** może być funkcją obliczaną dla wielu/wszystkich wierszy tabeli
- jeśli nie wystąpi zmiana nazwy **AS** nazwa_kolumny to nazwa funkcji będzie nazwą w tabeli wynikowej

- **Przykład - Zliczanie Wierszy:**

**SELECT COUNT(*) FROM** klient;

To zapytanie zwraca liczbę klientów. Wynik jest jedną kolumną o nazwie "count" i jednym wierszem, co czyni go pojedynczą liczbą, którą można użyć w dalszych obliczeniach.

- **Usuwanie Powtórzeń Przed Zliczaniem:**

**SELECT COUNT(DISTINCT** nazwisko) **FROM** klient;

Funkcja **DISTINCT** usuwa powtórzenia przed podjęciem zliczania, co jest przydatne w przypadku, gdy chcemy zliczyć unikalne wartości.

- **Inne Funkcje Agregujące:**

**SELECT MAX**(koszt), **MIN**(koszt), **AVG**(koszt) **AS** średni **FROM** towar;

To zapytanie wyświetla tabelę o jednym wierszu i trzech kolumnach, zawierających maksymalny, minimalny i średni koszt produktów w tabeli **towar**.

# SELECT - klauzula GROUP BY

Klauzula **GROUP BY** w poleceniu **SELECT** pozwala grupować wiersze o identycznych atrybutach z listy kolumn.

Oto kilka kluczowych punktów dotyczących tej klauzuli:

- **Składnia:**

**SELECT** lista_kolumn, funkcja_agregująca(wyrażenie)
**FROM** tabela
**WHERE** warunek
**GROUP BY** lista_kolumn;

- **Przykład - Grupowanie i Zliczanie:**

**SELECT** towar_nr, **COUNT**(zamowienie_nr), **SUM**(ilosc)
**FROM** pozycja
**GROUP BY** towar_nr

**ORDER BY COUNT**(zamowienie_nr) **DESC**;

To zapytanie grupuje wiersze w tabeli **pozycja** według atrybutu **towar_nr**. Następnie używa funkcji agregujących, takich jak **COUNT** i **SUM**, aby obliczyć liczbę zamówień (**COUNT(zamowienie_nr)**) oraz sumę ilości (**SUM(ilosc)**) dla każdej grupy. Wyniki są uporządkowane malejąco według liczby zamówień.

Grupowanie jest przydatne do analizy danych w kontekście określonych kategorii. Funkcje agregujące, takie jak **COUNT**, **SUM**, **AVG**, itp., mogą być używane, aby uzyskiwać statystyki dla każdej grupy.

**GROUP BY** lista_kolumn

- Występuje po klauzulach FROM i WHERE
- Wynikiem jest tabela, w której zgrupowano wiersze o identycznych atrybutach z listy kolumn
- Elementy wyboru instrukcji SELECT mają obowiązek dawać jednoznaczną wartość dla każdej grupy:
    - albo muszą odwoływać się do atrybutów z listy kolumn, w/g których grupujemy
    - albo do funkcji agregujących

Wymóg jednoznaczności dla wartości atrybutu traktowany jest w SQL formalnie:

- tzn. można odwoływać się do tylko atrybutów, w/g których następuje grupowanie
- nie wystarczy gwarancja jednoznaczności poprzez użycie klucza kandydującego
- w poniższym przykładzie trzeba dodać atrybut opis do grupowania, mimo że nie spowoduje to zmiany grup:

To zapytanie SQL korzysta z klauzuli GROUP BY i łączenia (INNER JOIN) w celu uzyskania informacji o liczbie zamówień i sumie ilości dla każdego towaru.

Oto analiza zapytania:

**SELECT** towar.nr, opis, **COUNT**(zamowienie_nr) **AS** liczba_zamowien,
**SUM**(ilosc) **AS** suma_ilosci
**FROM** pozycja **INNER JOIN** towar **ON** towar_nr = towar.nr
**GROUP BY** towar.nr, opis
**ORDER BY** liczba_zamowien **DESC**;

- **Składnia:**
    - **COUNT(zamowienie_nr) AS liczba_zamowien**: Funkcja agregująca **COUNT** zlicza liczbę zamówień i nadaje wynikowi alias "liczba_zamowien".
    - **SUM(ilosc) AS suma_ilosci**: Funkcja agregująca **SUM** oblicza sumę ilości i nadaje wynikowi alias "suma_ilosci".
- **Analiza:**

- **FROM pozycja INNER JOIN towar ON towar_nr = towar.nr**: Wyrażenie **INNER JOIN** łączy tabelę **pozycja** z tabelą **towar** na podstawie wspólnego atrybutu **towar_nr**.
- **GROUP BY towar.nr, opis**: Klauzula **GROUP BY** grupuje wyniki według numeru towaru (**towar.nr**) i opisu towaru.
- **ORDER BY liczba_zamowien DESC**: Klauzula **ORDER BY** sortuje wyniki malejąco według liczby zamówień.

Wyniki tego zapytania będą zawierać numer towaru, opis towaru, liczbę zamówień dla każdego towaru (**liczba_zamowien**) oraz sumę ilości dla każdego towaru (**suma_ilosci**). Wyniki będą posortowane malejąco według liczby zamówień.

W Postgresie wersji 9 w powyższym przykładzie można opuścić atrybut opis, grupowanie wg klucza gwarantuje jednoznaczność.

# SELECT - klauzula HAVING

Klauzula **HAVING** w zapytaniu SQL służy do filtrowania wyników grup, które zostały zgrupowane za pomocą klauzuli **GROUP BY**.

**HAVING** wyrażenie_warunkowe

- Występuje po innych klauzulach.
- Wynikiem jest tabela taka jak otrzymana poprzez użycie **GROUP BY**, ale dodatkowo z wyeliminowanymi grupami nie spełniającymi wyrażenia warunkowego.
- Brak **GROUP BY** oznacza, że cała tabela jest jedną grupą.
- Wyrażenie warunkowe odwołuje się do wartości, które można wyświetlić legalnie w SELECT.

*Oto analiza zapytania:*

**SELECT** towar_nr, **COUNT**(zamowienie_nr) **AS** liczba_zamowien **FROM** pozycja
**GROUP BY** towar_nr
**HAVING COUNT**(zamowienie_nr) > 1
**ORDER BY** liczba_zamowien **DESC**;

- **Składnia:**
  - **HAVING COUNT(zamowienie_nr) > 1**: Klauzula **HAVING** filtruje grupy, zachowując tylko te, które mają więcej niż jedno zamówienie.

- **Analiza:**
  - **FROM pozycja**: Określa, z którą tabelą będziemy pracować (w tym przypadku **pozycja**).
  - **SELECT towar_nr, COUNT(zamowienie_nr) AS liczba_zamowien**: Wybiera numer towaru i liczbę zamówień dla każdego towaru, nadając wynikowi alias "liczba_zamowien".
  - **GROUP BY towar_nr**: Klauzula **GROUP BY** grupuje wyniki według numeru towaru.

- **HAVING COUNT(zamowienie_nr) > 1**: Klauzula **HAVING** eliminuje grupy, które mają tylko jedno zamówienie.
- **ORDER BY liczba_zamowien DESC**: Klauzula **ORDER BY** sortuje wyniki malejąco według liczby zamówień.

Wyniki tego zapytania będą zawierać numery towarów oraz liczbę zamówień dla tych towarów, ale będą uwzględniać tylko te grupy, które mają więcej niż jedno zamówienie. Wyniki będą posortowane malejąco według liczby zamówień.

**SELECT** towar.nr, opis, **COUNT**(zamowienie_nr) **AS** liczba_zamowien
**FROM** pozycja
**INNER JOIN** towar **ON** towar_nr = towar.nr
**WHERE** opis **LIKE** '%układanka%'
**GROUP BY** towar.nr, opis;

- **HAVING** jest słuszne, gdy odwołuje się do wartości zagregowanych
- wartości pojedynczych krotek powinny być zbadane przed grupowaniem, w klauzuli **WHERE**

- **Składnia:**
  - **WHERE opis LIKE '%układanka%'**: Klauzula **WHERE** filtruje wiersze przed grupowaniem, wybierając tylko te, które zawierają "układanka" w opisie.
  - **GROUP BY towar.nr, opis**: Grupuje wyniki według numeru towaru i opisu.

Dzięki tej poprawce zapytanie będzie bardziej logiczne, a wyniki będą zawierały numery towarów, opisy i liczbę zamówień tylko dla tych towarów, które mają "układanka" w opisie.

# SELECT – zagnieżdżenia

Zagnieżdżone zapytanie, pozwala uniknąć problemu z powtarzającymi się klientami o tym samym nazwisku.

Podaj nazwiska klientów, którzy założyli zamówienie po 1 marca 2021:

**SELECT DISTINCT** nazwisko
**FROM** klient **K**, zamowienie
**WHERE K**.nr = klient_nr **AND** data_zlozenia > '2021-3-1'

- rozwiązanie to jest niezbyt szczęśliwe
- jeśli występuje dwóch klientów o tym samym nazwisku, to tego nie zauważymy
- użycie **DISTINCT** jest konieczne, ponieważ dla danego klienta może być wiele zamówień
- właściwsze byłoby użycie **SELECT DISTINCT** nr, nazwisko
- jeśli nie jesteśmy zainteresowani wyświetlaniem nr, to trzeba stosować grupowanie (**DISTINCT GROUP BY**)

*Oto poprawione zapytanie:*

```
SELECT nazwisko
FROM klient
WHERE nr IN (
        SELECT klient_nr
        FROM zamowienie
        WHERE data_zlozenia > '2021-3-1'
 );
```

- **Opis:**
  - Zewnętrzne zapytanie (SELECT nazwisko FROM klient WHERE nr IN (...);) wybiera nazwiska klientów, którzy spełniają warunek w zagnieżdżonym zapytaniu.
  - Zagnieżdżone zapytanie (SELECT klient_nr FROM zamowienie WHERE data_zlozenia > '2021-03-01';) zwraca numery klientów, którzy złożyli zamówienie po 1 marca 2021.
  - Warunek WHERE nr IN (...) zapewnia, że klient jest uwzględniany tylko raz, nawet jeśli złożył więcej niż jedno zamówienie w określonym okresie.

Dzięki temu rozwiązaniu unikamy problemu z powtarzającymi się klientami o tym samym nazwisku, a wyniki są bardziej zgodne z intencją zapytania.

*Podsumowanie:*

- zagnieżdżona tabela użyta w warunku, tabela jednokolumnowa służy jako zbiór
- nie jest obliczane złączenie
- każdy klient jest wyświetlany co najwyżej raz (tzn. jeśli spełnia warunek)
- jeśli powtarzają się nazwiska klientów spełniających warunek, to będą one uwzględnione

# SELECT - zagnieżdżenia głębokie

Zagnieżdżone zapytanie z głębokimi zagnieżdżeniami pozwala na bardziej złożone operacje, uwzględniając kilka warstw zapytań. Zastosowanie zagnieżdżonych zapytań z głębokimi zagnieżdżeniami jest przydatne w sytuacjach, gdzie warunki selekcji są bardziej skomplikowane i wymagają analizy danych z kilku połączonych tabel.

Podaj nazwiska klientów, którzy cokolwiek zamówili (tzn. złożyli niepuste zamówienie – puste też bywają):

```
SELECT nazwisko
FROM klient
WHERE nr IN (
        SELECT klient_nr
        FROM zamowienie
        WHERE nr IN (
                SELECT zamowienie_nr
                FROM pozycja
```

```
            WHERE zamowienie_nr IS NOT NULL
        )
    );
```

- wielokrotne zagnieżdżenia, trzeba rozpatrywać od wewnątrz

**Opis:**

- Zewnętrzne zapytanie (SELECT nazwisko FROM klient WHERE nr IN (...);) wybiera nazwiska klientów, którzy spełniają warunek w zagnieżdżonym zapytaniu.
- Pierwsze zagnieżdżone zapytanie (SELECT klient_nr FROM zamowienie WHERE nr IN (...);) zwraca numery klientów, którzy złożyli zamówienie.
- Drugie zagnieżdżone zapytanie (SELECT zamowienie_nr FROM pozycja WHERE zamowienie_nr IS NOT NULL;) zwraca numery zamówień, które posiadają przynajmniej jedną pozycję.
- Warunek WHERE nr IN (...) zapewnia, że klient jest uwzględniany tylko raz, nawet jeśli złożył więcej niż jedno zamówienie z niepustymi pozycjami.

To zapytanie pozwala na uzyskanie nazwisk klientów, którzy złożyli zamówienia z przynajmniej jedną niepustą pozycją w zagnieżdżonym zapytaniu głębokim.

## SELECT - zagnieżdżenie w atrybucie wynikowym

Zagnieżdżone zapytanie w atrybucie wynikowym pozwala na uzyskanie informacji z innej tabeli dla każdego wiersza wynikowego.

Podaj numery towarów wraz z ich całkowitymi wielkościami zamówień:

```
SELECT towar_nr, sum(ilosc) AS razem
FROM pozycja
GROUP BY towar_nr;
```

Podobne rozwiązanie:

```
SELECT nr, ( SELECT sum(ilosc) AS razem
        FROM pozycja
        WHERE towar_nr=towar.nr
) AS razem
FROM towar;
```

- zagnieżdżona tabela 1x1 użyta jako pojedyncza wartość
- wyświetlone są wszystkie towary, nawet te niezamawiane

Zadanie polega na wykorzystaniu zagnieżdżonego zapytania w atrybucie wynikowym w celu uzyskania informacji o całkowitej ilości zamówień dla każdego towaru.

*Poniżej znajduje się zapytanie, które to realizuje:*

```
SELECT nr, (
   SELECT COALESCE(SUM(ilosc), 0) AS razem
   FROM pozycja
   WHERE towar_nr = towar.nr
) AS razem
FROM towar;
```

- **Opis:**
  - Zewnętrzne zapytanie (SELECT nr, (...) AS razem FROM towar;) wybiera numery towarów wraz z ich całkowitymi wielkościami zamówień.
  - Wewnętrzne zapytanie (SELECT COALESCE(SUM(ilosc), 0) AS razem FROM pozycja WHERE towar_nr = towar.nr;) zwraca sumę ilości zamówień dla danego towaru. Użyto funkcji COALESCE, aby zwrócić 0, gdy nie ma żadnych zamówień dla danego towaru.

To zapytanie pozwala uzyskać numery towarów wraz z całkowitą ilością zamówień dla każdego z nich, nawet dla tych towarów, które nie były zamawiane.

## SELECT - zagnieżdżenie w klauzuli FROM i alias

Zagnieżdżone zapytanie w klauzuli FROM z aliasem pozwala na używanie rezultatów jednego zapytania jako tabeli tymczasowej w kolejnym zapytaniu.

Poniżej znajduje się zapytanie SQL, które oblicza zysk dla każdego towaru, a następnie przypisuje opinię w zależności od wartości zysku.

*Oblicz i zanalizuj zysk:*

```
SELECT *,
  CASE
    WHEN zysk/koszt < 0 THEN 'ujemny'
    WHEN zysk/koszt < 0.4 THEN 'za mało'
    WHEN cena IS NULL THEN 'brak danych'
    ELSE 'ok'
  END AS opinia
FROM (
  SELECT *, cena - koszt AS zysk
  FROM towar
) AS QQ;
```

- Zewnętrzne zapytanie używa aliasu QQ dla zagnieżdżonego zapytania.
- Zagnieżdżone zapytanie oblicza zysk dla każdego towaru (SELECT *, cena - koszt AS zysk FROM towar).

- Klauzula CASE w zewnętrznym zapytaniu przypisuje opinię w zależności od wartości zysku/kosztu i sprawdza, czy cena jest równa NULL.
- Całość wyświetla dane o towarze, zysku oraz przypisanej opinii.

To zapytanie pozwala na uzyskanie informacji o zysku dla każdego towaru oraz przypisanej opinii zależnej od wartości zysku i ceny.

*Dodatkowo:*
- tabela w zagnieżdżeniu ma dodatkową kolumnę
- tabela ta musi być nazwana i wówczas może być użyta jako źródło dla kolejnego wyszukiwania

# SQL - Wartość nieokreślona NULL

W bazach danych, wartość NULL jest używana do reprezentowania różnych koncepcji braku informacji lub nieokreśloności.

Poniżej przedstawiono kilka kontekstów, w których wartość NULL jest używana:

1. **Wartość nieokreślona NULL:**
    - Oznacza sytuację, w której wartość dla danego atrybutu nie jest znana w danym momencie.
    - Przykładem może być atrybut "telefon" w tabeli klientów, gdzie klienci, których numer telefonu jest nieznany, mogą mieć to pole ustawione na NULL.
2. **Wartość nieznana w tej chwili:**
    - Wskazuje na brak aktualnej informacji na temat pewnej wartości.
    - Przykładem może być data wypożyczenia w tabeli książek, gdzie książki, które nie są aktualnie wypożyczone, mają datę wypożyczenia ustawioną na NULL.
    - np. klienci, których telefon jest nieznany
3. **Wartość nie mogąca mieć sensu w danym kontekście:**
    - W przypadku, gdy jedna tabela realizuje dwie encje połączone związkiem jedno-jednoznacznym, a jedna z tych encji nie ma sensu w danym kontekście, wartość ta jest ustawiana na NULL.
    - Na przykład, tabela przedmiot_termin zawiera terminy związane z przedmiotami. Jeśli przedmiot nie ma przypisanego terminu, to wartości dotyczące dnia tygodnia, godziny i sali są ustawiane na NULL.
    - np. tabela książek z kluczem obcym wskazującym na aktualnego czytelnika i datą wypożyczenia
    - Jeśli książka nie jest wypożyczona, to klucz obcy jest NULL,
    - ale wówczas data wypożyczenia nie ma sensu, też musi być NULL
4. **Klucz kandydujący:**

- W standardzie SQL/92, klucz kandydujący nie może mieć więcej niż jednej wartości NULL. Jednak w niektórych implementacjach, takich jak PostgreSQL, wiele wystąpień NULL nie narusza warunku klucza kandydującego.

5. **Klucz główny:**
    - Wartość NULL nie jest dozwolona jako część klucza głównego.
    - Można sprawdzić i ograniczyć wartość NULL w zapytaniach, takich jak SELECT * FROM klient WHERE telefon IS NOT NULL.
6. **Porównywanie wartości NULL:**
    - Porównywanie wartości NULL zawsze zwraca fałsz. Dlatego WHERE zaleglosc != NULL jest błędne, a poprawne jest WHERE zaleglosc IS NOT NULL.

Wartość NULL pełni istotną rolę w zarządzaniu brakiem danych lub nieokreślonością, a zrozumienie jej kontekstu i zastosowanie jest kluczowe dla skutecznego projektowania i używania baz danych.

*Przykłady:*

- **Przykład CREATE TABLE przedmiot_termin:**

```
CREATE TABLE przedmiot_termin (
  kod serial PRIMARY KEY,
  rodzaj varchar(20) not null,
  nazwa varchar(50) not null,
  dzien_tyg int,
  godzina int,
  sala int,
  CONSTRAINT UNIQUE (dzien_tyg, godzina, sala)
);
```

Tworzy tabelę przedmiot_termin z kluczem głównym "kod" oraz atrybutami takimi jak "rodzaj," "nazwa," "dzien_tyg," "godzina," i "sala."
Klucz UNIQUE (dzien_tyg, godzina, sala) umożliwia unikalność kombinacji tych trzech atrybutów.

- **Przykład SELECT klient z warunkiem WHERE:**

```
SELECT nazwisko, telefon
FROM klient
WHERE telefon IS NOT NULL;
```

Wybiera nazwiska i numery telefonów klientów, dla których numer telefonu nie jest NULL, co oznacza, że są znane.

- **Przykład INSERT INTO towar z wartością NULL:**

**INSERT INTO** towar (opis, koszt, cena)
**VALUES** (E'ramka do fotografii 3\'x4\'', 13.36, **NULL**);

Dodaje nowy towar (ramkę do fotografii) z kosztem 13.36 i ceną ustawioną na NULL, co oznacza, że cena nie jest znana w chwili dodawania rekordu.

- **Przykład SELECT klient z warunkiem WHERE na atrybucie logicznym:**

**SELECT ***
**FROM** klient
**WHERE** zaleglosc = TRUE **OR** zaleglosc = FALSE;

Wybiera klientów, którzy mają zaznaczoną zaległość (TRUE) lub nie mają (FALSE).

- **Przykład SELECT wszystkich klientów:**

**SELECT ***
**FROM** klient;

Wybiera wszystkie dane klientów bez żadnych warunków, co obejmuje również te, dla których wartość atrybutu "zaleglosc" jest NULL.

- **Przykład SELECT klient z warunkiem WHERE na atrybucie z NULL:**

**SELECT ***
**FROM** klient
**WHERE** zaleglosc IS NOT NULL;

Wybiera klientów, którzy mają zdefiniowaną wartość atrybutu "zaleglosc," ale nie uwzględnia klientów, dla których ta wartość jest NULL, ponieważ NULL nie spełnia warunku.

# Wartość NULL – złączanie

| *nr* | *opis* | *koszt* | *cena* |
|---|---|---|---|
| 2 | układanka typu puzzle | 16.43 | 19.99 |
| 5 | chusteczki higieniczne | 2.11 | 3.99 |
| 10 | moneta srebrna z Papieżem | 20.00 | 20.00 |
| 19 | zegarek męski | 26.43 | |
| 11 | torba plastikowa | 0.01 | 0.00 |
| 17 | donica duża | 26.43 | |
| 12 | nożyczki drewniane | 8.18 | |

| *towar_nr* | *ilosc* |
|---|---|
| 2 | 2 |
| 5 | 3 |
| 10 | 1 |
| 19 | 1 |

| *nr* | *opis* | *koszt* | *cena* | *towar_nr* | *ilosc* |
|---|---|---|---|---|---|
| 2 | układanka typu puzzle | 16.43 | 19.99 | 2 | 2 |
| 5 | chusteczki higieniczne | 2.11 | 3.99 | 5 | 3 |
| 10 | moneta srebrna z Papieżem | 20.00 | 20.00 | 10 | 1 |
| 19 | zegarek męski | 26.43 | | 19 | 1 |
| 11 | torba plastikowa | 0.01 | 0.00 | | |
| 17 | donica duża | 26.43 | | | |
| 12 | nożyczki drewniane | 8.18 | | | |

# Ochrona danych

Podstawy kontroli dostępu i ochrony danych w SZBD.

**Podstawowe aspekty związane z ochroną danych w bazie danych są to:**

- ✓ ***Poufność*** (ang. *secrecy*) – użytkownik nie widzi danych, których nie ma prawa oglądać;
- ✓ ***Spójność*** (ang. *integrity*) – użytkownik nie może modyfikować danych, jeśli nie ma do tego odpowiednich uprawnień;
- ✓ ***Dostępność*** (ang. *availability*) – użytkownik ma dostęp do wszystkich danych i może je modyfikować, o ile ma przyznane do tego uprawnienia;

**Ochrona danych w bazie danych jest realizowana przez:**
- politykę ochrony danych;
- mechanizmy ochrony danych.

## *Polityka ochrony danych*

(ang. *security policy*) polega na określeniu jaka część danych ma być chroniona oraz którzy użytkownicy mają mieć dostęp do których części chronionych danych.

## *Mechanizmy ochrony danych*

(ang. *security mechanisms*) są to mechanizmy określone zarówno w systemie operacyjnym, jak i w SZBD, jak i na zewnątrz komputera jak ochrona dostępu do budynków, gdzie znajduje się komputer z bazą danych.
Podstawowe mechanizmy ochrony danych stosowane w SZBD to następujące metody kontroli dostępu do danych w bazie danych:

- ✓ *Uznaniowa kontrola dostępu* (ang. *Discretionary access control*);
- ✓ *Obowiązkowa kontrola dostępu* (ang. *Mandatory access control*);
- ✓ *Statystyczne bazy danych* - dostęp do danych statystycznych przez zapytania sumaryczne;
- ✓ Sporządzanie i analiza *audytu* operacji wykonywanych przez użytkownika na bazie danych.

## *Uznaniowa kontrola dostępu*

Jest to podstawowy mechanizm ochrony danych w bazie danych. Jest oparty na uprawnieniach do wykonywania operacji na obiektach bazy danych.

Właściciel obiektu ma pełne prawa do obiektu. Może część tych praw przekazać innym, wybranym przez siebie użytkownikom.

Osoba przyznająca uprawnienie może to uprawnienie w przyszłości odwołać.

Zarządzanie uprawnieniami i użytkownikami jest wspomagane przez **role**, które odzwierciedlają sposób funkcjonowania organizacji. W trakcie działania aplikacji baz danych role można dynamicznie włączać i wyłączać.

## *Obowiązkowa kontrola dostępu*

Jest mechanizmem stosowanym dodatkowo oprócz uznaniowej kontroli dostępu.
Jej celem jest ochrona bazy danych przed nieuprawnionymi zmianami jakie mogą być dokonane poza bazą danych - w programach aplikacyjnych (w rodzaju kodu wprowadzającego konia trojańskiego bez wiedzy użytkownika aplikacji).

Obowiązkowa kontrola dostępu jest oparta na ***klasach poufności*** (ang. *security class*) przypisanych do poszczególnych obiektów w bazie danych i do użytkowników (lub programów). Przez porównanie klasy obiektu i klasy użytkownika, system podejmuje decyzję czy użytkownik może wykonać określoną operację na obiekcie. Przede wszystkim, zapobiega to aby informacja nie płynęła z wyższego poziomu poufności do niższego.

## *Model Bell-LaPadula*

W zaawansowanych technologicznie bazach danych może być też stosowana metoda obligatoryjna (Mandatory Access Control), w której każdy obiekt oraz każdy użytkownik posiada określoną klasę poufności, co stanowi podstawę do umożliwienia lub zablokowania dostępu odczytu i zapisu obiektu przez danego użytkownika.

W metodzie obligatoryjnej najczęściej stosowany jest model Bell-LaPadula wyróżniający 4 klasy poufności: ściśle tajne, tajne, poufne i nieposiadające klauzuli poufności. Metoda ta jest stosowana głównie w systemach o podwyższonych wymaganiach dotyczących kontroli dostępu do danych.

***Obejmuje:***

- *Obiekty* np. tabele, perspektywy, wiersze.
- *Podmioty* np. użytkownicy, programy użytkowników.
- *Klasy poufności* przypisywane podmiotom class(P) i obiektom class(O); tworzą one liniowy porządek. Np.:
  top secret - TS, secret - S , confidential - C, unclassified - U: TS > S> C > U
- Każdemu obiektowi i każdemu podmiotowi zostaje przypisana klasa poufności.

## *Inne metody ochrony danych*

1. **Macierz *RAID*** (ang. *Redundant Array of Independent Disks*) - redundantny zapis danych w zbiorze dysków; gdy jeden dysk ulega awarii, potrzebne dane są pobierane z drugiego.
2. ***Szyfrowanie*** - dane poufne mogą być przechowywane w postaci zaszyfrowanej.
3. Stawianie serwera bazy danych za *firewall*em lub za serwerem *proxy*.
4. ***Certyfikaty cyfrowe* dołączane do przesyłanych danych** - stwierdzające autentyczność nadawcy danych i/lub umożliwiające odbiorcy zaszyfrowanie swojej zwrotnej odpowiedzi.
5. ***Protokół szyfrowania SSL*** - tworzy bezpieczne połączenie między klientem i serwerem do przesyłania poufnych danych takich jak numer karty kredytowej.

# Tworzenie synonimów nazw tabel i perspektyw

W języku SQL, tworzenie synonimów to sposób na nadanie alternatywnej nazwy tabeli lub perspektywy. Pozwala to na używanie bardziej zwięzłej nazwy w zapytaniach, co może być przydatne szczególnie w przypadku długich lub skomplikowanych identyfikatorów obiektów.

Poniżej przedstawiam przykłady tworzenia i usuwania synonimów w SQL:

- **Tworzenie synonimu:**

  **CREATE SYNONYM** nazwa_synonimu
  **FOR** nazwa_tabeli_lub_perspektywy;

*Przykład:*

**CREATE SYNONYM** Dept
**FOR** Kadry.Dept@mojafirma.com.pl;

W tym przykładzie, nazwa Dept stała się synonimem dla Kadry.Dept@mojafirma.com.pl.

- **Usuwanie synonimu:**

  **DROP** SYNONYM nazwa_synonimu;

*Przykład:*

**DROP** SYNONYM Dept;

Usuwa synonim o nazwie Dept.

Tworzenie synonimów jest szczególnie przydatne w sytuacjach, gdzie chcemy skrócić długie i złożone nazwy tabel lub perspektyw, co poprawia czytelność i zwiększa efektywność pisanego kodu SQL. Ponadto, używanie synonimów może pomóc w utrzymaniu niezależności logicznej danych, zwłaszcza gdy różne poziomy zewnętrzne potrzebują odrębnych nazw dla tych samych obiektów bazodanowych. Wspomaga niezależność logiczną danych. Każdy poziom zewnętrzny może mieć swoje odrębne nazwy.

# Transakcje

Często elementarną operacją na bazie danych nie jest wcale pojedyncza instrukcja SQL, ale ciąg takich instrukcji, nazywany *transakcją*.

W bazach danych, transakcje są używane do grupowania operacji w sposób, który gwarantuje, że wszystkie operacje zostaną wykonane albo żadna. Transakcje są ważne, aby zapewnić spójność i integralność danych.

*Oto jak wyglądałaby transakcja w SQL:*

*-- Rozpoczęcie transakcji*
**BEGIN TRANSACTION**;

*-- Pierwsza operacja*
**UPDATE** Konta SET Saldo = Saldo - 1000 **WHERE** Id_klienta = 1001;

*-- Druga operacja*
**UPDATE** Konta SET Saldo = Saldo + 1000 **WHERE** Id_klienta = 9999;

*-- Sprawdzenie, czy druga operacja może być wykonana (tu można dodać dodatkowe warunki)*

*-- Jeśli druga operacja jest możliwa, zatwierdzenie transakcji*
**COMMIT**;

*-- Jeśli druga operacja nie jest możliwa lub coś się nie powiedzie, wycofanie transakcji*
**ROLLBACK**;

W przypadku powodzenia obu operacji, COMMIT zostanie użyte do zatwierdzenia transakcji, co oznacza, że wszystkie zmiany zostaną utrwalone w bazie danych. W przypadku wystąpienia problemu, ROLLBACK zostanie użyte do cofnięcia wszystkich zmian wprowadzonych w trakcie transakcji, przywracając bazę danych do poprzedniego stanu.

Transakcje są ważne, aby uniknąć problemów związanych z niejednolitością danych i zapewnić spójność w przypadku błędów czy awarii. Działa to na zasadzie "wszystko albo nic". Jeśli którakolwiek operacja w transakcji nie powiedzie się, cała transakcja jest cofana (rollback), co utrzymuje integralność danych.

Przelanie pieniędzy z jednego konta na drugie, jest elementarną operacją z punktu widzenia aplikacji bankowej.

- W SQL używamy w tym celu co najmniej dwóch instrukcji UPDATE:

**UPDATE** Konta **SET** Saldo = Saldo – 1000
**WHERE** Id_klienta = 1001;
**UPDATE** Konta **SET** Saldo = Saldo + 1000
**WHERE** Id_klienta = 9999;

Powyższa sekwencja SQL to dwie instrukcje UPDATE, które mają na celu przelanie kwoty 1000 jednostek waluty z konta o identyfikatorze klienta 1001 na konto o identyfikatorze klienta 9999. Jednak, w przypadku transakcji, ważne jest, aby zapewnić spójność danych i ewentualne wycofanie zmian w razie błędu.

- **COMMIT** - zatwierdza zmiany w bazie danych bez możliwości późniejszego ich wycofania.

# Dziedziny (domeny) (Standard)

Dziedziny (domeny) pozwalają na zdefiniowanie niestandardowych typów danych, które mogą być używane w różnych miejscach w bazie danych. To pomaga w zapewnieniu jednolitości danych i ułatwia zarządzanie typami danych w bazie danych.

Zdefiniujmy dziedzinę numerów departamentów:
**CREATE DOMAIN** Dept# **CHAR**(3)
**CHECK** (**VALUE IN** ('A00','B01','D01','D11','D21', 'XXX'))
**DEFAULT** 'XXX';

W powyższym fragmencie SQL definiujesz dziedzinę (domenę) o nazwie Dept#, która ma typ danych CHAR(3). Domena ta jest ograniczona przy użyciu klauzuli CHECK, która sprawdza, czy wartość zawiera się w podanej liście ('A00', 'B01', 'D01', 'D11', 'D21', 'XXX'). Domyślna wartość tej domeny to 'XXX'.

Następnie, możemy jej użyć przy określaniu typu danych kolumn w tabelach:
**CREATE TABLE** Dept
(DeptNo **DOMAIN** Dept# **PRIMARY KEY**,
...);

Później, używasz tej dziedziny jako typu danych dla kolumny DeptNo w tabeli o nazwie Dept. Oznaczasz tę kolumnę jako klucz główny (PRIMARY KEY), co oznacza, że będzie służyć do jednoznacznej identyfikacji rekordów w tej tabeli. Pozostała część definicji tabeli (...) jest pominięta w Twoim fragmencie, ale zazwyczaj obejmuje definicje innych kolumn.

# Asercje (Standard)

Więzy spójności definiowane poza instrukcjami CREATE TABLE i ALTER TABLE dotyczące całej tabeli.
Asercje są używane do definiowania warunków spójności, które muszą być spełnione przez dane w tabeli na poziomie całej tabeli, a nie tylko dla poszczególnych rekordów.

**CREATE ASSERTION** maxempl
**CHECK** (1000 <= **SELECT COUNT** (*)
**FROM** Emp);

**DROP ASSERTION** *nazwa_asercji*;

W powyższym fragmencie SQL tworzysz asercję (assertion) o nazwie maxempl.

Asercja ta sprawdza, czy liczba rekordów w tabeli Emp (przy użyciu SELECT COUNT(*) FROM Emp) jest większa lub równa 1000. Asercje pozwalają na zdefiniowanie warunków, które muszą być spełnione przez całą tabelę, niezależnie od konkretnych rekordów.

Później, w drugiej części, używasz polecenia DROP ASSERTION, które służy do usunięcia asercji o nazwie nazwa_asercji (w Twoim przypadku maxempl).

W przypadku maxempl zapewnia ona, że liczba rekordów w tabeli Emp będzie zawsze większa lub równa 1000.

***Pytanie kontrolne:***

*Jaką czynność realizuje poniższy mechanizm asercji:*

**CREATE ASSERTION** maxemployee

**CHECK** (1000 <= **SELECT COUNT (*) FROM** Employee);

***Odpowiedź:***

Poniższa asercja ma na celu sprawdzenie, czy liczba pracowników (Employee) nie przekracza 1000. Gdy liczba ta zostanie przekroczona, asercja nie zostanie spełniona, co oznaczałoby naruszenie warunku spójności.

Asercje w SQL są używane do definiowania warunków spójności na poziomie tabeli lub bazy danych. W tym konkretnym przypadku, asercja maxemployee sprawdza, czy liczba pracowników nie przekracza 1000. Jeśli liczba ta jest większa niż 1000, to asercja jest naruszona, co może skutkować odrzuceniem zmian w bazie danych. Jest to narzędzie zapewniające integralność danych na poziomie tabeli.