

# Relacyjne bazy danych

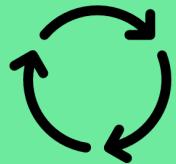
Wykład

# Artur Radomski

*Intel Technology Poland*



# Wprowadzenie



BHP



ZALICZENIE  
PRZEDMIOTU

# Zaliczenie

**Wykład** - kończy się egzaminem teoretycznym (treść wykładu)

# Architektura DBMS

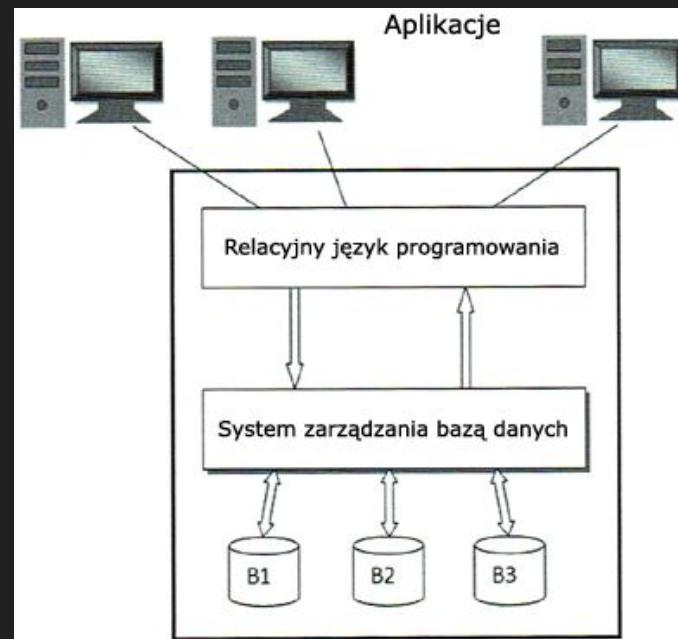
Do obsługi baz danych tworzone są złożone systemy zawierające zbiory gotowych narzędzi zapewniających dostęp do danych. Umożliwiają one manipulowanie danymi zgromadzonymi w systemach komputerowych i aktualizowanie tych danych.

Do najważniejszych cech SZBD można zaliczyć:

- operowanie na dużych i bardzo dużych zbiorach danych
- zarządzanie złożonymi strukturami.

# Architektura DBMS

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.**



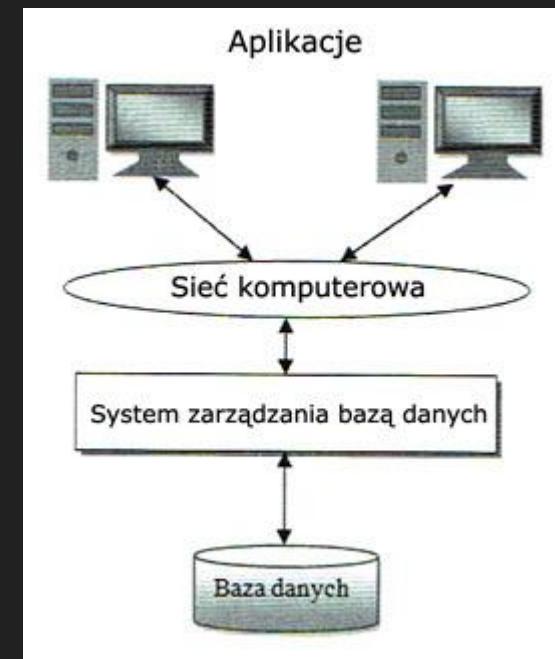
# Architektura DBMS

W praktyce stosuje się dwa sposoby komunikacji z bazą danych:

- architektura klient-serwer
- architektura 3-warstwowa

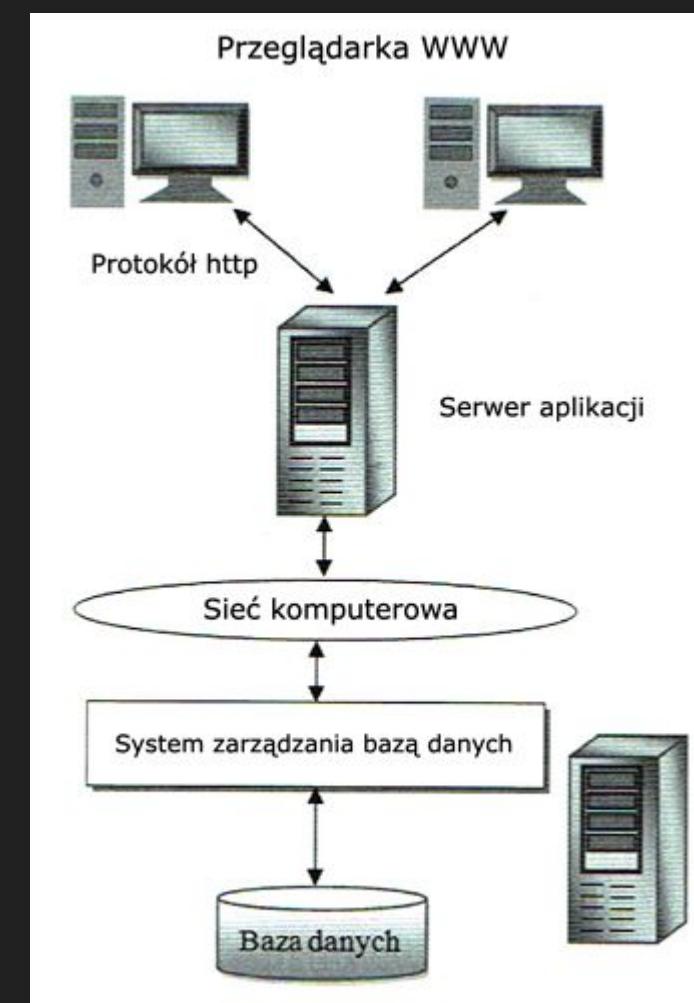
# Architektura klient-serwer

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.



# Baza danych

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

Baza danych powinna charakteryzować się następującymi cechami:

- trwałość danych
- integralność danych
- bezpieczeństwo danych
- współdzielenie danych
- abstrakcja danych
- niezależność danych (logiczna i fizyczna)
- integracja danych

# Język SQL

# 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

# Inne

- 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

- Jest wiele typów wbudowanych, najważniejsze z nich:
  - **CHAR(\_)**, **VARCHAR(\_)**
  - **INTEGER**, **SMALLINT**
  - **DATE**, **TIME**, **TIMESTAMP**, obsługa czasu i daty
  - **BOOLEAN**, wartości np. 't', TRUE, '1','y', 'yes', ( SQL/99 )
  - **NUMERIC(\_,\_)**, np. **NUMERIC** (7,2), 7 cyfr, w tym 2 po przecinku
  - **FLOAT(\_)**, np. **FLOAT**(15), 15 cyfr znaczących
  - **BIT**, **VARBIT**, wartości np. B'10011101'
  - **MONEY**, to samo co **NUMERIC** (9,2)

# Tabele

- **CREATE TABLE** nazwa\_tabeli  
lista-( **definicja\_kolumny** [ wartość\_domyślna ]  
| [ definicja\_klucza\_kandydującego ]  
| [ definicja\_klucza\_obcego ]  
| [ definicja\_warunku\_poprawności ] ) ;
- **definicja\_kolumny** ::= nazwa\_kolumny nazwa\_dziedziny
- **nazwa\_dziedziny** ::= typ\_wbudowany | nazwa\_zdefiniowana  
::= "to jest"  
| "albo"  
[ ] "alternatywnie"
- 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

- definicja\_klucza\_kandydujacego ::=  
**UNIQUE** ( lista\_kolumn ) |  
**PRIMARY KEY** ( lista\_kolumn )
  - lista kolumn w obu przypadkach jest niepusta
  - najwyżej jeden klucz może być określony jako główny (**PRIMARY KEY**)
  - jeśli występuje klucz główny, to wszystkie atrybuty tego klucza zyskują warunek poprawności **NOT NULL**
  - klucz alternatywny dopuszcza wartości **NULL**,
- PostgreSQL – nie naruszają one warunku
- SQL92 (np.. MS SQL Server) – naruszają warunek
- Warunki poprawności można opcjonalnie nazwać:  
CONSTRAINT nazwa\_definicja\_klucza\_kand.

# Klucze obce

- definicja\_klucza\_obcego ::=  
**FOREIGN KEY** ( lista\_kolumn )  
**REFERENCES** tabela\_bazowa [ ( lista\_kolumn ) ]  
[ **ON DELETE** opcja ]  
[ **ON UPDATE** opcja ]  
– 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**
- Warunki poprawności można opcjonalnie nazwać:  
**CONSTRAINT** nazwa definicja\_klucza\_obcego

# Warunki poprawności

- **definicja\_warunku\_poprawności ::=**  
**CHECK** ( wyrażenie\_warunkowe )
  - wyrażenie\_warunkowe może być dowolnie skomplikowane, nie musi ograniczać się do danej tabeli, musi być określone dla każdego wiersza tabeli
  - więzy poprawności są spełnione, jeśli powyższe wyrażenie\_warunkowe ma wartość "true" dla każdego wiersza tabeli
  - system zarządzania bazą danych nie zezwoli na wprowadzenie danych czy aktualizację danych takie, że więzy poprawności nie są spełnione
  - kolejność sprawdzania warunków jest nieokreślona
- Warunki poprawności można opcjonalnie nazwać:  
**CONSTRAINT** nazwa **definicja\_warunku\_poprawności**

# Zmiana definicji tabeli podstawowej

Zmiana definicji tabeli podstawowej

- **ALTER TABLE** nazwa\_tabeli operacja;

- Przykład

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

- operacja może oznaczać
    - dodanie/usunięcie/zmiana nazwy kolumny,
    - zmiana dotychczasowej wartości domyślnej w kolumnie,
    - dodanie/usunięcie warunku poprawności

```
ALTER TABLE klient
```

```
ALTER COLUMN telefon DROP NOT NULL
```

# Usuwanie tabeli podstawowej

## Usuwanie tabeli podstawowej

- **DROP TABLE** nazwa\_tabeli [ **RESTRICT** | **CASCADE** ];
  - jeżeli wybrano **RESTRICT** i tabela podstawowa występuje w jakiejkolwiek definicji perspektywy, to instrukcja **DROP TABLE** nie powiedzie się
  - jeżeli wybrano **CASCADE**, to instrukcja **DROP TABLE** powiedzie się i usunie daną tabelę wraz ze wszystkimi perspektywami bazującymi na tej tabeli oraz więzami poprawności

# CRUD

**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

**INSERT INTO** cel ( lista\_elementów ) źródło;

- **cel** jest nazwą tabeli, do której wstawiamy dane
- **lista\_elementów** zawiera listę nazw atrybutów, którym chcemy nadać wartość
- **źródło** ma jedną z dwu postaci

**VALUES** ( lista\_wartości )

# Przykład

**INSERT INTO** kod\_kreskowy **VALUES** ('4892840112975', 17)

- wstawia jeden wiersz
- nadaje wartości atrybutom zadeklarowanym w definicji tabeli, w kolejności deklaracji
- nie można opuścić żadnego z atrybutów

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

- wstawia dwa wiersze

# “Chwilówki”

**INSERT INTO** chwilowa

```
SELECT imie, nazwisko, ulica_dom  
FROM klient  
WHERE miasto = 'Gdańsk'
```

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

- taka tabela jest usuwana po zakończeniu sesji

**INSERT INTO** towar ( opis, koszt, cena )

```
VALUES ( E'ramka do fotografii 3\'x4\'", 13.36, NULL )
```

- znak ukośnika jest niezbędny, gdy wprowadzana wartość zawiera znak zastrzeżony, np. apostrof czy ukośnik
- można wprowadzić w jawny sposób wartość nieokreślona

# UPDATE

**UPDATE** cel **SET** element = wartość **WHERE** warunek

- **cel** jest nazwą tabeli, w której aktualizujemy dane
  - **element** jest nazwą atrybutu, któremu przypisujemy wartość
  - klauzula **WHERE** wyznacza wiersze, w których będzie dokonana aktualizacja
  - ma ona identyczne znaczenie jak w instrukcji SELECT, w szczególności jej brak oznacza, że wszystkie wiersze będą aktualizowane
- 
- *SQL nie przewiduje możliwości aktualizacji kilku atrybutów w jednym poleceniu*
  - *niektóre implementacje dopuszczają taką możliwość*

# Przykład

**UPDATE** towar **SET** cena = 1.15 **WHERE** nr=5

- aktualizacja pojedynczego wiersza (klucz główny)

**UPDATE** towar **SET** cena = cena\*1.15 **WHERE** opis **LIKE** '%układanka%'

- aktualizacja wielu wierszy jednocześnie

**UPDATE** towar **SET** cena =

( **SELECT** cena **FROM** towar **WHERE** nr=5 )

- tabela 1x1 występuje w roli pojedynczej wartości (gdyby warunek WHERE w zagnieżdżonym zapytaniu nie odwoływał się do wartości kluczowej, polecenie UPDATE mogłoby produkować błąd)
- brak warunku **WHERE** w poleceniu **UPDATE** oznacza, że jest globalne – dotyczy całej tabeli

# DELETE

**DELETE FROM** cel **WHERE** warunek

- cel jest nazwą tabeli, z której usuwamy dane
- klauzula **WHERE** wyznacza wiersze, w których będzie dokonana aktualizacja
- ma ona identyczne znaczenie jak w instrukcji **SELECT**, w szczególności jej brak oznacza, że wszystkie wiersze są usuwane

PostgreSQL i inne implementacje pozwalają na nieodwołalne usunięcie całej zawartości tabeli: **TRUNCATE TABLE** cel

Uwaga: usuwanie wszystkich danych z tabeli, to nie jest to samo co usuwanie tabeli:  
**DROP TABLE** cel

# Przykład

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

Usuń wszelkie informacje o zamówieniach składanych przez klientów z Gdyni:

**DELETE FROM** zamówienie Z

**WHERE** (

**SELECT** miasto

**FROM** klient K

**WHERE** K.nr = Z.klient\_nr

) = 'Gdynia'

# SELECT

**SELECT** [ ALL | DISTINCT ] lista\_atrybutów\_wynikowych [ lista\_klauzul ];

- lista\_atrybutów\_wynikowych realizuje m.in. rzut i zmianę nazwy kolumny, nie może być pusta
- lista\_klauzul realizuje m.in. obcięcie i złączenie
- klauzule: **FROM, WHERE, ORDER BY, GROUP BY, HAVING**

**SELECT DISTINCT** imie, nazwisko

-- rzut na atrybuty

**FROM** klient

**WHERE** miasto = 'Gdańsk'

-- obcięcie do wierszy spełniających warunek

# Lista atrybutów

Atrybut wynikowy jest albo gwiazdką \* albo postaci: **wyrażenie\_skalarne [ [ AS ] nazwa\_kolumny ]**

- \* oznacza wszystkie atrybuty  
**SELECT \* FROM** towar  
wyświetla całą tabelę towarów
- **wyrażenie\_skalarne** będzie najczęściej nazwą pojedynczego atrybutu  
**SELECT imie, nazwisko FROM** klient

**DISTINCT** usuwa powtarzające się wiersze w tabeli wynikowej, domyślnie jest **ALL**

- niektóre implementacje porządkują wynik, nie jest to standard

# FROM

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

**SELECT \* FROM** towar, kod\_kreskowy

- iloczyn kartezjański dwóch tabel

**SELECT \* FROM** klient, towar

- w obu tabelach występuje atrybut „nr”, czysto przypadkowa zbieżność
- podając nazwę atrybutu, w przypadku takiej zbieżności, trzeba dodać nazwę tabeli

# WHERE

## Klauzula **WHERE**

**WHERE** wyrażenie\_warunkowe

- występuje po klauzuli FROM
- wynikiem jest wybór tych wierszy, które spełniają warunek  
**SELECT \* FROM klient WHERE miasto = 'Gdańsk'**
- obcięcie relacji w/g warunku miasto = 'Gdańsk'

## 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*

# SELECT - złączenie

Złączenie jest wyborem pasujących wierszy w iloczynie kartezjańskim:

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

- bez warunku **WHERE** byłyby wszystkie pary wierszy
- czyli iloczyn kartezjański
- w obu tabelach występuje atrybut „nr”, trzeba wyjaśnić, o który chodzi

Wygodne może być stosowanie aliasów dla nazw tabel

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

- w złączeniach wielokrotnie powtarzamy nazwę tabeli
- ale jeśli alias jest zadeklarowany, musi być koniecznie używany

# SELECT - łączenie 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 - złączenie INNER JOIN

Inna składnia na złączenie:

**SELECT** K.nr, nazwisko, imie, data\_zlozenia

**FROM** klient K **INNER JOIN** zamówienie **ON** K.nr = klient\_nr

- 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

- wyrażenie\_skalarne może odwoływać się do nazw atrybutów, ale zawierać dodatkowe obliczenia
- nazwa\_kolumny będzie nazwą kolumny w tabeli wynikowej

**SELECT \* , cena – koszt AS zysk FROM towar**

- dodaje nową kolumnę w wyświetlanym wyniku
- zawiera ona wyniki obliczeń

<b>nr</b>	<b>opis</b>	<b>koszt</b>	<b>cena</b>	<b>zysk</b>
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

# SELECT - atrybuty wynikowe

Bardziej wymyślne wyrażenie:

```
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
```

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

# SELECT - atrybuty wynikowe

- Możliwość wykonania obliczeń wykraczających poza proste operacje algebra relacji (rzut uogólniony)
- Dodatkowe obliczenia w wyrażeniu skalarnym nie muszą ograniczać się do atrybutów z tabel:
  - **SELECT** 2+2
  - **SELECT** now()
  - **SELECT** version()  
version

---

PostgreSQL 10.12 (Ubuntu 10.12-0ubuntu0.18.04.1) on x86\_64-pc-linux-gnu,  
compiled by gcc (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0, 64-bit  
(1 row)

- tabela wynikowa w ogóle nie odwołuje się do żadnej relacji

# SELECT - warunki WHERE

Podaj nazwiska klientów spoza Trójmiasta:

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

- warunek należenia do zbioru

Podaj opis wszystkich ramek do fotografii, które mają podany wymiar w calach (tj. znak prim na końcu opisu):

```
SELECT opis FROM towar  
WHERE opis LIKE 'ramka%' and opis LIKE E '%\'
```

- dopasowanie wzorca tekstowego

Wyświetl szczegóły zamówień złożonych w lutym 2021:

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

- warunek dla zakresu dat

# **SELECT - wyrażenia warunkowe dla WHERE**

Pojedyncze wartości: **WHERE** cena > 3.14

Relacja pomiędzy wartością a zbiorem wartości:

**WHERE** miasto **NOT IN** ('Gdańsk', 'Gdynia', 'Sopot')

**WHERE** koszt >= ALL ( **SELECT** koszt **FROM** towar)

Istnienie elementów: **WHERE NOT EXISTS** ( **SELECT** \* ...

Jednoznaczność elementów:

**SELECT** \* **FROM** zamówienie

**WHERE NOT**

klient\_nr MATCH UNIQUE ( **SELECT** nr **FROM** klient )

(to się nie powinno zdarzyć, jeśli nr jest kluczem w tabeli klientów)

# SELECT - klauzula ORDER BY

**ORDER BY** lista\_kolumn [ DESC | ASC ]

- Występuje po klauzulach **FROM** i **WHERE**
- Wynikiem jest tabela, w której wiersze uporządkowano według atrybutów z listy kolumn, kolejność rosnąca (ASC, domyślnie) lub malejąca (DESC)

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

- wyświetla tabelę towarów uporządkowaną według kosztów, zaczynając od największych

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

- dodatkowa opcja pozwalająca ograniczyć wyświetlanie

# **SELECT - funkcje agregujące**

- 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

**SELECT** count(\*) **FROM** klient

- zwraca liczbę klientów
- tylko jedna kolumna, o nazwie „count”, i jeden wiersz
- wynik może być użyty jako pojedyncza liczba

**SELECT** count (DISTINCT nazwisko) **FROM** klient

- usuwa powtórzenia przed podjęciem zliczania

**SELECT** max(koszt), min(koszt), avg(koszt) **AS** średni

**FROM** towar

- wyświetla tabelę o jednym wierszu i trzech kolumnach

# **SELECT - klauzula GROUP BY**

## **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

**SELECT** towar\_nr, count(zamowienie\_nr), sum(ilosc)

**FROM** pozycja

**GROUP BY** towar\_nr

**ORDER BY** count(zamowienie\_nr) DESC

# **SELECT - klauzula GROUP BY cd.**

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:

```
SELECT towar.nr, opis, count(zamowienie_nr), sum(ilosc)  
FROM pozycja INNER JOIN towar ON towar_nr=twarz.nr  
GROUP BY towar.nr, opis  
ORDER BY count(zamowienie_nr) DESC
```

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

# SELECT - klauzula HAVING

## **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

```
SELECT towar_nr, count(zamowienie_nr) FROM pozycja
```

```
GROUP BY towar_nr
```

```
HAVING count(zamowienie_nr) > 1
```

```
ORDER BY count(zamowienie_nr) DESC
```

# **SELECT - klauzula HAVING cd.**

```
SELECT towar.nr, opis, count(zamowienie_nr)  
FROM pozycja INNER JOIN towar on towar_nr=towar.nr  
GROUP BY towar.nr, opis  
HAVING opis LIKE '%układanka%'
```

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

```
SELECT towar.nr, opis, count(zamowienie_nr)  
FROM pozycja INNER JOIN towar on towar_nr=towar.nr  
WHERE opis LIKE '%układanka%'  
GROUP BY towar.nr, opis
```

# SELECT - zagnieżdżenia

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

```
SELECT DISTINCT nazwisko  
FROM klient K, zamówienie  
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)**

# **SELECT** - zagnieżdżenia cd.

Właściwe rozwiązanie:

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

- 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

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

```
SELECT nazwisko  
FROM klient  
WHERE nr IN (  
    SELECT klient_nr  
    FROM zamówienie  
    WHERE nr IN (  
        SELECT zamówienie_nr  
        FROM pozycja  
    )  
)
```

- wielokrotne zagnieżdżenia, trzeba rozpatrywać od wewnętrz

# **SELECT - zagnieżdżenie w atrybucie wynikowym**

Podaj numery towarów wraz z ich całkowitymi wielkościami zamówień:

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

Podobne rozwiążanie:

```
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
- wyświetlane są wszystkie towary, nawet te niezamawiane

# **SELECT - zagnieżdżenie w klauzuli FROM i alias**

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) QQ
```

- 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.

# Wartość NULL

Wartość nieznana w tej chwili

- np. klienci, których telefon jest nieznany

Wartość nie mogąca mieć sensu w danym kontekście

- 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

# Wartość NULL

Konieczność, gdy jedna tabela realizuje dwie encje połączone związkiem jedno-jednoznaczny, np.

```
CREATE TABLE przedmiotTermin (
    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)
)
```

- przy odrębnych tabelach przedmiot mógł nie być adresatem klucza obcego z tabeli terminów
- ale w jednej tabeli przedmiot występuje i termin musi być zastąpiony NULlem

# Wartość NULL - własności

## Klucz kandydujący:

- SQL/92: taka sama wartość jak inne, a więc może wystąpić w tabeli najwyżej jeden raz
- mało sensowne podejście – tylko jeden klient może być bez telefonu, tylko jedna książka nie wypożyczona
- PostgreSQL, i wiele innych: wartość nieznana, a więc wiele wystąpień NULL nie narusza warunku na klucz kandydujący

## Klucz główny: wartość NULL nie jest dozwolona wcale

- Można sprawdzać tę wartość:

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

# Wartość NULL - własności

Można jawnie wprowadzać tę wartość:

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

Wartość NULL nie pasuje do żadnego wzorca

- założymy, że tabela klientów ma atrybut logiczny „zaległość”  
**SELECT \* FROM klient**  
**WHERE zaleglosc = TRUE OR zaleglosc = FALSE**
- nie wykaże wszystkich klientów, jedynie tych z określoną wartością tego atrybutu  
**SELECT \* FROM klient**
- wykaże wszystkich klientów, również z nieokreślona wartością atrybutu  
**SELECT \* FROM klient**  
**WHERE zaleglosc != NULL**
- jest absolutnie błędne, działania z NULL nigdy nie zwrócią wartości

# Wartość NULL - złączanie

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			

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.

# Ochrona danych w bazie danych

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

Ochrona danych w bazie danych jest realizowana przez politykę ochrony danych oraz 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.

# Ochrona danych w bazie 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

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

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 *firewalliem* 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 przypadku długich identyfikatorów obiektów (np. specyfikacje sieciowe) – własne synonimy.

```
CREATE SYNONYM nazwa_synonimu  
FOR nazwa_tabeli_lub_perspektywy;  
  
CREATE SYNONYM Dept  
FOR Kadry.Dept@mojafirma.com.pl;  
  
DROP SYNONYM nazwa_synonimu;
```

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*.

Np. 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;
```

# Transakcje

Załóżmy, że pierwsza instrukcja wykonała się, a druga nie może zostać wykonana na przykład z powodu tego, że 9999 jest błędnym identyfikatorem klienta albo z powodu awarii komputera. Z punktu widzenia aplikacji dane znalazły się w stanie niespójnym i pozostaje tylko jedna możliwość - wycofać wynik pierwszej instrukcji UPDATE. Do tego celu służy instrukcja ROLLBACK.

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

```
UPDATE Konta SET Saldo = Saldo - 1000
```

```
WHERE Id_klienta = 1001;
```

```
UPDATE Konta SET Saldo = Saldo + 1000
```

```
WHERE Id_klienta = 9999;
```

# Dziedziny (domeny) (Standard)

Zdefiniujmy dziedzinę numerów departamentów:

```
CREATE DOMAIN Dept# CHAR(3)
CHECK (VALUE IN ('A00', 'B01', 'D01', 'D11', 'D21', 'XXX'))
DEFAULT 'XXX';
```

Następnie, możemy jej użyć przy określaniu typu danych kolumn w tabelach:

```
CREATE TABLE Dept
(DeptNo DOMAIN Dept# PRIMARY KEY,
...);
```

# Asercje (Standard)

- Więzy spójności definiowane poza instrukcjami CREATE TABLE i ALTER TABLE dotyczące całej tabeli.

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

```
DROP ASSERTION nazwa_asercji;
```