

# Podstawy SQL

- **SQL - Structured Query Language -strukturalny język zapytań**
- Światowy standard przeznaczony do definiowania, operowania i sterowania danymi w relacyjnych bazach danych
- Powstał w firmie IBM pod koniec lat 70-tych
- Występuje w produktach większości firm produkujących oprogramowanie do zarządzania bazami danych
- Polecenia **SQL** mają postać podobną do zdań w języku angielskim
- Pomimo prób standaryzacji istnieje szereg różnych dialektów **SQL**
- **SQL** używany jest jako standardowe narzędzie umożliwiające dostęp do danych w różnych środowiskach, z różnym sprzętem komputerowym i różnymi systemami operacyjnymi
- Język **SQL** jest niewrażliwy na rejestr czcionki, czyli wielkie i małe litery nie są rozróżniane

# Wprowadzenie do SQL

- SQL zapewnia obsługę:
  - zapytań - wyszukiwanie danych w bazie
  - operowania danymi - wstawianie, modyfikowanie i usuwanie
  - definiowania danych - dodawanie do bazy danych nowych tabel
  - sterowania danymi - ochrona przed niepowołanym dostępem
- Użytkownik określa operacje jakie mają być wykonane nie wnikając w to, jak mają być wykonane
- Najprostsza postać zapytań w **SQL** służy do wybierania rekordów pewnej tabeli, które spełniają określony w zapytaniu warunek
- Taki typ zapytania stanowi odpowiednik operatora selekcji w algebrze relacyjnej
- Takie najprostsze zapytanie, jak zresztą prawie wszystkie zapytania w tym języku, konstruuje się za pomocą trzech słów kluczowych: **SELECT**, **FROM** i **WHERE**

## Podstawowe klauzule w SQL

**SELECT** nazwy\_kolumn

**FROM** nazwa\_tabeli

**WHERE** warunek;

- Pozwalają na wybranie z tabeli określonych kolumn i rekordów spełniających ustalone warunki czyli pozwalają na realizację rzutowania i selekcji
- Warunek formułowany jest jako złożone wyrażenie porównania
- Przykładowa tabela o nazwie **NAZWISKA** zawiera kolumny:
  - NUMER –
  - IMIE
  - NAZWISKO

- STANOWISKO
- PENSJA
- MIASTO

## Klauzule **SELECT** i **FROM**

- **SELECT** - podstawowa klauzula **SQL** - używana do wyszukiwania danych w tabeli • Występuje wraz z klauzulą **FROM**

**SELECT** \*

**FROM** *nazwa-tabeli*;

- Gwiazdka oznacza, że należy wyszukać wszystkie kolumny tabeli
- Jest to przykład instrukcji wybierającej całą tabelę

- W klauzuli **SELECT** zostają określone nazwy kolumn, których wartości, z rekordów spełniających warunek zapytania (formułowany przy pomocy klauzuli **WHERE**), są dołączane do odpowiedzi
- Klauzula **FROM** służy do określenia tabeli, której dotyczy zapytanie

## Klauzula **WHERE**

- W klauzuli **WHERE** formułuje się warunek, który odpowiada warunkowi wyboru (selekcji) w algebrze relacyjnej i który określa ograniczenia, jakie mają spełniać rekordy, aby zostać wybrane w danym zapytaniu
- Jeżeli rekord spełnia te ograniczenia to zostaje dołączony do tabeli wynikowej
- Postać zapytania

**FROMWHERESELECT** *nazwa-tabeli warunki; \**

- Klauzula **WHERE** pozwala na wybranie z tabeli tych wierszy, które spełniają określone warunki

```
SELECT *
```

```
FROM NAZWISKA
```

```
WHERE STANOWISKO = 'URZEDNIK';
```

- Dla podanego przykładu z tabeli zostaną wybrane tylko te rekordy, w których w polu **STANOWISKO** jest wpisane 'URZEDNIK'

## Formułowanie warunku

- Po słowie kluczowym **WHERE** występuje wyrażenie warunkowe
- Do zapisu porównywania wartości w języku SQL służy sześć operatorów:

-równy                   =

-nierówny               <>

-mniejszy              < -

większy               >

-mniejszy lub równy   <= -większy

lub równy             >=

- W wyrażeniu mogą występować stałe oraz nazwy kolumn tabel wymienionych w klauzuli **FROM**
- Dla wartości numerycznych można budować wyrażenia arytmetyczne korzystając z operatorów + - \* / i nawiasów ( )



# Formułowanie warunku

- Stałe tekstowe w **SQL** są ujmowane w pojedyncze cudzysłowy **'Przykład tekstu'**
- W wyniku porównania powstaje wartość logiczna **TRUE** (prawda) lub **FALSE** (fałsz)
- Wartości logiczne można łączyć w wyrażenia logiczne za pomocą operatorów logicznych **AND, OR** i **NOT**
- Priorytet operatorów wykorzystywanych w budowie wyrażień: **operatory porównania, NOT, AND, OR**
- Porównywanie tekstów - dwa teksty są równe, jeśli występują w nich kolejno te same znaki
- Przy teście „nierównościowym” tekstów, tzn. przy wykonywaniu porównań takich jak **<** lub **>=**, o wartości porównania decyduje, czy kolejne znaki z tekstu z lewej strony są alfabetycznie wcześniejsze, czy dalsze w stosunku do znaków z tekstu umieszczonego po prawej stronie wyrażenia
- Przykłady

# Formułowanie warunku

Adamski > Adamowicz

Adam < Adamowicz

- Wartości NULL nie podlegają żadnym operacjom porównania, gdyż jest ona traktowana jako wartość nieznana
- SQL umożliwia testowanie pól w poszukiwaniu wartości NULL
- Użycie w klauzuli **WHERE** zwrotu **IS NULL** jest wykorzystywane do sprawdzania czy pole zawiera tę wartość
- Zamiast standardowego operatora porównania pojawią się słowo **IS**
- Słowo NULL nie jest zawarte w cudzysłowie

## Formułowanie warunku

- Można dokonać przeszukania danych w celu wybrania obiektów posiadających wartości
- W tym celu używa się wyrażenia **IS NOT NULL**

## Przykładowe dane w tabeli NAZWISKA

Numer	Imię	Nazwisko	Stanowisko	Pensja	Miasto
1	Jan	Kowalski	urzędnik	900,00 zł	Gdańsk
2	Waldemar	Pawlak	kierownik	3 000,00 zł	Sopot
3	Marian	Malinowski	urzędnik	1 100,00 zł	Gdynia
4	Adam	Nowak	księgowy	2 000,00 zł	Gdańsk
5	Ewa	Musiał	stażysta		Gdańsk
6	Zenon	Miler	stażysta		Gdynia
7	Paul	Davies	prezes	8 000,00 zł	Londyn
8	Mieczysław	Dobija	kontroler	3 000,00 zł	Warszawa
9	Peter	Norton	informatyk	3 500,00 zł	Gdańsk

### Rzutowanie i selekcja

- Z wybranych rekordów można eliminować składowe, które nie są potrzebne

- Tabelę uzyskaną jako wynik zapytania można rzutować na pewne kolumny, czyli ograniczyć w tabeli wynikowej liczbę kolumn

- Postać zapytania

```
SELECT nazwy-kolumn
```

```
FROM nazwa-tabeli
```

```
WHERE warunek;
```

- Przykład instrukcji wybierającej kolumny zawierające imię i nazwisko (wszystkie rekordy) z tabeli **NAZWISKA**

```
SELECT IMIE, NAZWISKO
```

```
FROM NAZWISKA;
```

- Wybór jak wyżej lecz jedynie rekordów, dla których pole **STANOWISKO** spełnia warunek sformułowany w klauzuli **WHERE**

```
SELECT IMIE, NAZWISKO, MIASTO
```

```
FROM NAZWISKA
```

```
WHERE STANOWISKO = 'PREZES' ;
```

## Rzutowanie i selekcja

- Postać polecenia:

```
SELECT Imię, Nazwisko, Stanowisko, Pensja
FROM NAZWISKA
```

```
WHERE (Stanowisko = 'Urzędnik' OR
Stanowisko = 'Prezes') AND Pensja >= 900;
```

- Z tabeli **NAZWISKA** zostaną wybrane rekordy zawierające kolumny: **Imię, Nazwisko, Stanowisko i Pensja** - pracowników zatrudnionych na stanowiskach Urzędnik i Prezes, których pensja jest równa, bądź większa od 900 zł
- Wynik działania polecenia:

<b>Imię</b>	<b>Nazwisko</b>	<b>Stanowisko</b>	<b>Pensja</b>
Jan	Kowalski	urzędnik	900,00 zł
Marian	Malinowski	urzędnik	1 100,00 zł
Paul	Davies	prezes	8 000,00 zł

## Wykonywanie obliczeń na danych

- Język SQL pozwala na wykonywanie obliczeń na danych i pokazywanie ich wyników w postaci wykonanych zapytań
- Wykonanie obliczeń polega na zastąpieniu pozycji z listy nazw kolumn (w klauzuli SELECT) przez odpowiednie wyrażenia
- Wyrażenie nie musi koniecznie zawierać nazw kolumn, można używać tylko liczb, albo wyrażeń algebraicznych lub łańcuchów znaków
- Postać polecenia:

```
SELECT 'Tekst objaśniający', Stanowisko, Pensja*2  
FROM NAZWISKA WHERE  
Pensja >= 900;
```

- Wynik zapytania  
**Uzycie słowa  
kluczowego AS**
- W zapytaniu można  
uzyc słowa kluczowego  
**AS**, aby przypisac  
nazwy kolumnom i

	<b>Wyr1</b>	<b>Stanowisko</b>	<b>Wyr2</b>
	Tekst objasniajacy	urzednik	1 800,00 zł
	Tekst objasniajacy	kierownik	6 000,00 zł
	Tekst objasniajacy	urzednik	2 200,00 zł
	Tekst objasniajacy	ksiegowy	4 000,00 zł

wyrazeniom (zamiast standardowych Wyr1, Wyr2)

- Nazwy te poprawiają czytelność danych zwracanych przez zapytanie oraz pozwalają odwołać się do nich przez nazwę
- Składnia polecenia wygląda następująco:

```
SELECT 'Tekst objasniajacy' ZKAAS KOMENTARZ,
```

```
Stanowisko, Pensja*2 AS PODWY
```

```
FROM NAZWISKA WHERE
```

```
Pensja >= 900;
```



- Wynik zapytania

<b>KOMENTARZ</b>	<b>Stanowisko</b>	<b>PODWYŻKA</b>
Tekst objasniajacy	urzednik	1 800,00 zł
Tekst objasniajacy	kierownik	6 000,00 zł
Tekst objasniajacy	urzednik	2 200,00 zł
Tekst objasniajacy	ksiegowy	4 000,00 zł

## Wykonywanie obliczen w klauzuli WHERE

- Podobnie jak mozna wykonywac obliczenia na danych wybranych z tabeli, mozna również wykonywac obliczenia w klauzuli WHERE, aby pomóc w filtrowaniu rekordów
- Przykład polecenia

```
Stanowisko, Pensja*2 SELECT 'Tekst objaASsniaj
PODWYacy' ZKAAS KOMENTARZ,
```

**FROM** NAZWISKA

**WHERE** Pensja\*2 >= 2\*900;

- Jest oczywiste, że wyniki polecenia będą takie same jak poprzednio
- Cecha charakterystyczna relacyjnych baz danych jest to, że kolejność kolumn i wierszy nie jest istotna - nie są one traktowane sekwencyjnie
- Można wybierać rekordy z bazy danych w dowolnym porządku
- Domyslnie pojawiają się w kolejności, w jakiej były wprowadzone
- Jednak często przeglądając rekordy chcemy tę kolejność określić, np. względem zawartości jednej z kolumn

## Sortowanie wyników zapytań

- Klauzula **ORDER BY** jest wykorzystywana do sortowania wyników
- Wyniki zapytania będą uporządkowane względem zawartości kolumny (lub kolumn), które określimy w klauzuli **ORDER BY**
- Sortowanie można przeprowadzić zarówno alfabetycznie jak i względem wartości numerycznych oraz kolumn zawierających dane w formacie **Date**

- Kolejność kolumn nie zależy od kolumny używanej do sortowania wyników zapytań - kolumny pozostają zawsze w tym samym porządku, bez względu na kolumnę, której używamy w klauzuli **ORDER BY**
- Dodanie do poprzedniego polecenia: **ORDER BY** Stanowisko;
- spowoduje, że wyniki zostaną posortowane według kolumny Stanowisko (w porządku rosnącym)
- Wyniki zapytań mogą być posortowane zarówno rosnąco (opcja domyślna), jak i malejąco
- Dla sortowania malejącego, używamy w klauzuli **ORDER BY** słowa kluczowego **DESC** (dla rosnącego słowa **ASC** – normalnie jest pomijane)

## Operatory logiczne w klauzuli **WHERE**

- Operacje wykonywane w klauzuli **WHERE** podlegają zasadom logiki boolowskiej - wynik przyjmuje zawsze jedną z wartości: prawda lub fałsz
- W przypadku, gdy wynik wyrażenia to prawda, wiersz jest wybierany, w przeciwnym przypadku – pomijany

- Operator **AND** zwraca wynik prawda, gdy wyrażenia po obu stronach operatora są prawdziwe - jeżeli choć jedno z nich jest nieprawdziwe, wtedy całe wyrażenie zwraca jako wynik wartość fałsz
- Operator **OR** zwraca wynik prawda, gdy jedno z wyrażen po prawej lub po lewej stronie operatora jest prawdziwe - gdy oba wyrażenia są prawdziwe, wynik też przyjmuje wartość prawda
- Operatora **NOT** używamy do zaprzeczenia wartości wyrażenia
- Wielokrotne operatory logiczne mogą być wykorzystywane do utworzenia złożonych instrukcji **WHERE**, w których wykorzystywanych jest kilka wyrażen jednocześnie
- Formułując takie wyrażenia należy pamiętać o priorytecie operatorów w celu zapewnienia poprawności obliczenia wartości wyrażenia

## Przykład złożonych wyrażeń

- Korzystając z danych zawartych w tabeli NAZWISKA wyszukać wszystkich pracowników mieszkających w Gdansk i Gdyni, którzy mają ustalone pensje i posortować wg pola Nazwisko malejąco
- Postać polecenia (polecenie daje nieprawidłowe wyniki):

```
SELECT Imie, Nazwisko, Pensja, Miasto
```

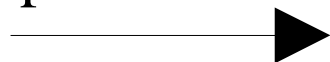
```
FROM NAZWISKA
```

```
WHERE Miasto = 'Gdansk' OR Miasto = 'Gdynia' AND
```

```
Pensja IS NOT NULL
```

```
ORDER BY Nazwisko DESC;
```

- Wynik działania polecenia:



- Poprawnie sformułowany warunek (z nawiasami):

```
WHERE
```

```
(Miasto = 'GdyniaMiasto
```

```
= 'Gdansk' ' ) OR
```

```
AND Pensja IS NOT NULL
```

Imie	Nazwisko	Pensja	Miasto
Adam	Nowak	2 000,00 zł	Gdansk
Peter	Norton	3 500,00 zł	Gdansk
Ewa	Musial		Gdansk
Marian	Malinowski	1 100,00 zł	Gdynia
Jan	Kowalski	900,00 zł	Gdansk

## Klauzula IN

Wzrost złożoności zapytań powoduje trudności z ustaleniem kolejności wykonywanych operacji – konieczne staje się stosowanie nawiasów wykorzystywanych do grupowania wyrażen w klauzuli **WHERE**

- W poprzednim przykładzie nawiasy ustalają kolejność w ten sposób, że najpierw wykonywane są instrukcje połączone operatorem **OR**, a następnie wykonana jest operacja z operatorem **AND**
- Język SQL dysponuje kilkoma dodatkowymi elementami, które znacznie upraszczają zapytania z wieloma operatorami logicznymi
- Klauzula **IN** zastępuje wiele operatorów **OR** w instrukcjach sprawdzających, czy wybrana grupa wartości znajduje się w kolumnie
- Operator **IN** określa, czy wartość testowana jest identyczna z przynajmniej jedną z wartości z listy
- Przykład ilustruje jak można uprościć poprzednie zapytanie:

```
SELECT Imie, Nazwisko, Pensja, Miasto
```

```
FROM NAZWISKA
```

```
WHERE Miasto IN ('Gdansk', 'Gdynia') AND Pensja IS NOT NULL  
ORDER BY Nazwisko DESC;
```

- 

## NOT IN

- Wartość logiczna wyrażenia zawartego wewnątrz klauzuli **IN** można zaprzeczyć operatorem **NOT**
- Klauzula **IN** wybiera wszystkie wiersze, w których wartość testowana jest równa jednej z wartości umieszczonych na liście
- **NOT IN** wybiera te wiersze, w których wartość testowana jest różna od każdej wartości z listy
- Przykład zapytania wybierającego wszystkich pracowników nie mieszkających w Gdańsku ani w Gdyni, którzy mają ustalone pensje:  

```
SELECT Imie, Nazwisko, Pensja, Miasto  
FROM NAZWISKA  
WHERE Miasto NOT IN ('Gdansk', 'Gdynia') AND Pensja IS NOT NULL  
ORDER BY Nazwisko DESC;
```
- Klauzula **NOT IN** może być zastąpiona przez operator **AND**  

```
SELECT Imie, Nazwisko, Pensja, Miasto  
FROM NAZWISKA
```

```

WHERE Miasto <> 'Gdansk' AND Miasto <> 'Gdynia' AND
Pensja IS NOT NULL
ORDER BY Nazwisko DESC;

```

## Klauzula **BETWEEN**

Klauzule **BETWEEN** i jej zaprzeczenie, **NOT BETWEEN**, wykorzystujemy do sprawdzenia, czy wartość należy lub nie należy do określonego przedziału wartości

- Klauzula **BETWEEN** służy do sprawdzenia, czy wartość należy do podanego zakresu z uwzględnieniem wartości granicznych
- Może być zastąpiona przez dwa porównania połączone operatorem **AND**
- Przykład zapytania wyszukiującego wszystkich pracowników których pensje mieszczą się w przedziale 1100-3000 zł, posortowane rosnąco wg pensji:

```

SELECT Imie, Nazwisko, Pensja, Miasto
FROM NAZWISKA
WHERE Pensja BETWEEN 1100 AND 3000

```

Imie	Nazwisko	Pensja	Miasto
------	----------	--------	--------



•

**ORDER BY** Pensja;

• Wynik  
zapytania: 

• Inaczej sformułowany  
warunek:

**WHERE** Pensja >= 1100

**AND** Pensja <= 3000

Marian	Malinowski	1 100,00 zł	Gdynia
Adam	Nowak	2 000,00 zł	Gdansk
Mieczyslaw	Dobija	3 000,00 zł	Warszawa
Waldemar	Pawlak	3 000,00 zł	Sopot

- 

## NOT BETWEEN

- Sprawdza czy podana wartosc znajduje sie poza okreslonym przedzialem
- Dzialanie tej instrukcji moze byc zastapione dwoma porównaniami polaczonymi instrukcja **OR**
- Sprawdzajac czy liczba znajduje sie pomiedzy innymi liczbami, logiczne wydaje sie, ze musi byc ona wieksza od dolnej wartosci i mniejsza od górnej wartosci
- Przyklad zapytania wyszukującego pracowników mających pensje niższe od 1100 i wyższe od 3000 zł:

```
SELECT Imie, Nazwisko, Pensja, Miasto  
FROM NAZWISKA
```

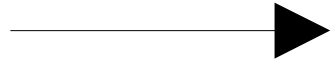
<b>Imie</b>	<b>Nazwisko</b>	<b>Pensja</b>	<b>Miasto</b>
Jan	Kowalski	900,00 zł	Gdansk

---

- ```
WHERE Pensja NOT
BETWEEN 1100 AND 3000
ORDER BY Pensja;
```

---

|              |             |        |
|--------------|-------------|--------|
| Peter Norton | 3 500,00 zł | Gdansk |
| Paul Davies  | 8 000,00 zł | Londyn |

- Wynik zapytania: 
- Inaczej sformułowany warunek:

```
WHERE Pensja < 1100 OR Pensja > 3000
```

## BETWEEN i inne typy danych

**BETWEEN** stosuje się również, żeby sprawdzić czy podana data i czas należą do podanego zakresu

- BETWEEN** można stosować również przy operacjach na łańcuchach, podobnie jak zwykle operatory porównania
- Postać zapytania wybierającego pracowników, których nazwiska zaczynają się od liter między 'D' a 'N':

```
SELECT Imie, Nazwisko, Pensja, Miasto
FROM NAZWISKA
```

**WHERE** Nazwisko **BETWEEN** 'D' **AND** 'N'

**ORDER BY** Pensja;

- Wynik zapytania Zenon Miler Gdynia



| Imie       | Nazwisko   | Pensja                        | Miasto   |
|------------|------------|-------------------------------|----------|
| Zenon      | Miler      |                               | Gdynia   |
| Ewa        | Musial     |                               | Gdansk   |
| Jan        | Kowalski   | 900,00 zł                     | Gdansk   |
|            |            | 2000 z lewej jest             |          |
|            |            | warunek $\geq$ a z prawej $<$ |          |
| Marian     | Malinowski | 1 100,00 zł                   | Gdynia   |
| Mieczyslaw | Dobija     | 3 000,00 zł                   | Warszawa |
| Paul       | Davies     | 8 000,00 zł                   | Londyn   |

## Zlozone klauzule **WHERE** z operatorem **LIKE**

- Działa na kolumnach zawierających wartości lancuchowe.
- Operator **LIKE** sprawdza czy wartość tekstowa odpowiada podanemu wzorcowi, umożliwia więc wykonywanie częściowych porównań, takich

- jak „zaczynający się od tekstu”, „konczący się na tekście”, lub „zawierający tekst”
- Tworząc wzorce stosuje się znaki wieloznaczne:
  - % - zastępuje sekwencję dowolnych znaków o długości  $n$  (gdzie  $n$  może być zerem)
  - \_ - odpowiada jednemu znakowi w przeszukiwanym tekście
- W Accessie
  - \* - zastępuje sekwencję dowolnych znaków o długości  $n$  (gdzie  $n$  może być zerem)
  - ? - odpowiada jednemu znakowi
- Ogólna postać polecenia z operatorem **LIKE** **WHERE** tekst **LIKE**  
wzorzec

## Przykład operatora LIKE

Postać zapytania wyszukującego wszystkie rekordy, w których w polu Nazwisko występuje sekwencja znaków 'no': **SELECT** Imie, Nazwisko, Pensja, Miasto

**FROM** NAZWISKA

**WHERE** Nazwisko **LIKE**

'\*no\*'

**ORDER BY** Nazwisko;

- Wynik zapytania



| Imie   | Nazwisko      | Pensja      | Miasto |
|--------|---------------|-------------|--------|
| Marian | Malinowski    | 1 100,00 zł | Gdynia |
| Peter  | <b>Norton</b> | 3 500,00 zł | Gdansk |
| Adam   | <b>Nowak</b>  | 2 000,00 zł | Gdansk |

- Postać zapytania, które wyszuka wszystkie rekordy, gdzie druga litera nazwiska jest „o”:

**SELECT** Imie, Nazwisko, Pensja, Miasto

**FROM** NAZWISKA

**WHERE** Nazwisko **LIKE** '?o\*'

- **ORDER BY** Nazwisko;
- Operator **LIKE** zmniejsza wydajność realizacji zapytania

## Usuwanie niepotrzebnych spacji

- Funkcja **TRIM** (**nazwa\_kolumny**) służy do odrzucenia spacji znajdujących się przed i za łańcuchem
- Przy założeniu, że niektóre nazwiska są wpisane błędnie z niepotrzebną spacją na początku, nie uzyskamy wszystkich informacji w wyniku działania zapytania
- Sformułowanie zapytania jak poniżej, z zastosowaniem funkcji **TRIM** (**nazwa\_kolumny**) usuwa ten problem
- Przykład polecenia:

```
SELECT Imie, Nazwisko, Pensja, Miasto  
FROM NAZWISKA  
WHERE TRIM(Nazwisko) BETWEEN 'D' AND 'N';
```

# Operator DISTINCT

- Zastosowanie operatora **DISTINCT** pozwala na wybranie unikalnych wartości spośród wszystkich występujących w danej kolumnie

- Postać polecenia z powtarzającymi się rekordami  
**SELECT** Stanowisko

**FROM** NAZWISKA

**ORDER BY** Stanowisko;

*Wynik zapytania*



Stanowisko

|                 |
|-----------------|
| informatyk      |
| kierownik       |
| kontroler       |
| ksiegowy        |
| prezes          |
| stazysta        |
| stazysta        |
| urzednik        |
| urzednik pelnia |

- Przykład – jeżeli chcemy dowiedzieć się jakie występują stanowiska (bez powtórzeń) w tabeli NAZWISKA, które pracownicy, to można sformułować zapytanie: **Stanowisko**

**SELECT DISTINCT** Stanowisko

**FROM** NAZWISKA

**ORDER BY** Stanowisko;

*Wynik zapytania*



|            |
|------------|
| informatyk |
| kierownik  |
| kontroler  |
|            |
|            |



# Funkcje agregujace

- 
- Zastosowanie operatora DISTINCT spowodowało, że na księgowy liście nie ma wartości powtarzających się prezes
- Zastosowanie słowa DISTINCT odnosi się do stazysta powtarzalności kombinacji wszystkich pól, jakie urzędnik wymienione są na liście
- W SQL dostępnych jest kilka funkcji agregujących działających na grupie wartości zwracanych przez zapytanie a nie na pojedynczej wartości pola
- Na przykład możemy w tabeli policzyć liczbę wierszy spełniających określone kryteria lub można wyliczyć wartość średnią dla wszystkich wartości z wybranej kolumny
- Funkcje te działają na wszystkich wierszach w tabeli, na pewnej grupie wierszy wybranej klauzula **WHERE** lub na grupach danych wybranych klauzula **GROUP BY**
- **Funkcja COUNT(nazwa\_kolumny)**
- Funkcja ta zlicza ilość wierszy w zapytaniu

# Funkcje agregujace

- 
- Jeżeli chcemy znać liczbę wierszy zwróconych przez zapytanie, najprościej użyć funkcji w postaci **COUNT(\*)** (gwiazdka - wszystkie kolumny tabeli)
- Są tego dwa powody:
  - po pierwsze pozwalamy optymalizatorowi bazy danych wybrać kolumnę do wykonania obliczeń, co czasem nieznacznie podnosi wydajność zapytania
  - po drugie, nie musimy się martwić o wartości NULL zawarte w kolumnie oraz o to, czy kolumna o podanej nazwie w ogóle istnieje

## **Funkcja COUNT(nazwa\_kolumny) i wartości NULL**

- Funkcja w postaci **COUNT(nazwa\_kolumny)** nie uwzględnia pól z wartościami NULL
- Użycie funkcji w postaci **COUNT(\*)** zlicza wszystkie wiersze bez względu na zawartość
- Fakt, że wiersze z wartością NULL nie są zliczane, może być przydatny, gdy wartość NULL ma jakieś szczególne znaczenie, np. brak ustalonej pensji

# Funkcje agregujace

- Postac zapytania z uwzględnieniem wartosci NULL w kolumnie **Pensja** ile osób z Gdańska pracuje w firmie

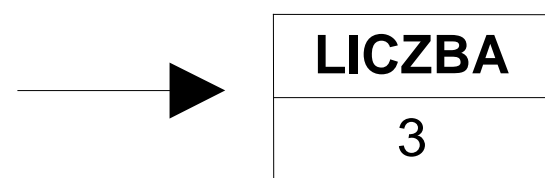
```
SELECT COUNT (*)
FROM NAZWISKA      Wynik zapytania
WHERE Miasto = 'Gdansk';
```



| Expr1000 |
|----------|
| 4        |

- Postac zapytania – zliczanie wg kolumny **Pensja** bez wartosci NULL, ze zmiana nazwy pola wyniku na LICZBA:

```
SELECT COUNT (Pensja) AS LICZBA FROM
NAZWISKA      Wynik zapytania
WHERE Miasto = 'Gdansk';
```



| LICZBA |
|--------|
| 3      |

## Funkcje SUM(nazwa\_kolumny) i AVG(nazwa\_kolumny)

- Funkcja **SUM()** dodaje wszystkie wartosci i zwraca pojedynczy wynik, a funkcja **AVG()** wylicza wartosc srednia dla grupy wartosci
- W przeciwienstwie do funkcji **COUNT()**, która dziala dla wszystkich typów danych, funkcje **SUM()** i **AVG()** dzialaja tylko dla argumentów liczbowych

# Funkcje agregujace

- 
- W przypadku funkcji **SUM()** i **AVG()** wartosci NULL sa ignorowane (nie sa uwzgledniane w obliczeniach)
- Obie funkcje moga byc uzyte z operatorem **DISTINCT** - jezeli go uzyjemy to obliczenia sa przeprowadzane tylko dla wartosci unikalnych
- Postac zapytania o sume do wyplaty:

```
SELECT SUM(Pensja) AS DO_WYPLATY DO_WYPLATY FROM
```

NAZWISKA;      *Wynik zapytania*      21 500,00 zł

|  |
|--|
|  |
|  |

- Postac zapytania o srednia pensje wszystkich pracowników:

```
SELECT AVG(Pensja) AS SREDNIA
```

**FROM** NAZWISKA;      *Wynik zapytania*

|                |
|----------------|
| <b>SREDNIA</b> |
| 3 071,43 zł    |

Postac zapytania o srednia dla pracowników pracujacych poza Trójmiastem

```
SELECT AVG(Pensja) AS SREDNIA
```

**FROM** NAZWISKA

*Wynik zapytania*

|                |
|----------------|
| <b>SREDNIA</b> |
| 5 500,00 zł    |

# Funkcje agregujace

- **WHERE** Miasto **NOT IN** ('Gdansk', 'Sopot', 'Gdynia');
- **Funkcje MIN(nazwa\_kolumny) i MAX(nazwa\_kolumny)**
- Sluza do znajdowania wartosci najmniejszej i największej w zbiorze wartosci
- Obie funkcje moga byc uzyte dla różnych typów danych
- Funkcja **MAX()** znajduje największy lancuch danych (zgodnie z regułami porównywania lancuchów) najnowsza date (lub najodleglejsza w przyszłości) oraz największa liczbe w zestawieniu
- Funkcja **MIN()** znajduje odpowiednio wartosci najmniejsze
- Wartosc **NULL** traktowana jest jako nieokreslona i nie mozna jej porównywac z innymi (wartości te są ignorowane)
- Zarówno funkcja **MAX** jak i **MIN** moga byc stosowane z operatorem **DISTINCT**, ale nie ma to wiekszego znaczenia, gdyż zwracaja i tak tylko jedna wartosc z zestawienia

## Funkcje agregujące – grupowanie wyników

Postać zapytania o maksymalna pensje osoby z Gdanska

```
SELECT MAX(Pensja) AS MAX_PENSJA
```

| MAX_PENSJA  |
|-------------|
| 3 500,00 zł |

```
FROM NAZWISKA WHERE Miasto = 'Gdansk';
```

*Wynik zapytania* →

- Postać zapytania o najnizsza pensje osoby pracujacej w Trójmieście

```
SELECT MIN(Pensja) AS NAJNIZSZA
```

```
FROM NAZWISKA
```

*Wynik zapytania*

```
WHERE Miasto IN ('Gdansk', 'Sopot', 'Gdynia');
```

| NAJNIZSZA |
|-----------|
| 900,00 zł |

- Wykonywanie obliczen z podzialem na kategorie
- Klauzula **GROUP BY** automatycznie dzieli wyniki zapytania na wybrane kategorie
- Umozliwia grupowanie wyników wzgledem zawartosci wybranej kolumny

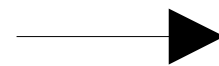
- 
- Jeżeli użyjemy w zapytaniu jednocześnie funkcji agregującej dla innej kolumny, to funkcja ta dokona obliczeń dla kategorii określonych w klauzuli **GROUP BY**
- Jest bardzo ważne, aby kolumna, względem której dokonujemy podziału na kategorie, znajdowała się w części deklarycyjnej wyrażenia **SELECT**

## Wykonywanie obliczeń z podziałem na kategorie

Postać zapytania robiącego zestawienie wypłat pensji dla poszczególnych miast:

```
SELECT Miasto, SUM(Pensja) AS SUMA
FROM NAZWISKA
GROUP BY Miasto;
```

*Wynik zapytania*



| <b>Miasto</b> | <b>SUMA</b> |
|---------------|-------------|
| Gdansk        | 400,00 zł   |
| Gdynia        | 100,00 zł   |
| Londyn        | 000,00 zł   |
| Sopot         | 000,00 zł   |
| Warszawa      | 000,00 zł   |

- Klauzula **GROUP BY** działa ze wszystkimi funkcjami agregującymi.

- Przy pomocy klauzuli **GROUP BY** można tworzyć grupy i podgrupy, w zależności od tego, czy wybrana jest więcej niż jedna kolumna.
- Postać polecenia dająca w wyniku, w jakich miastach występują jakie stanowiska:
 

| Miasto   | Stanowisko         |
|----------|--------------------|
| Gdańsk   | informatyk         |
| Sopot    | informatyk         |
| Warszawa | kontroler miastach |
| Gdańsk   | księgowy           |
| Londyn   | prezes             |
| Gdynia   | stazysta           |
| Gdańsk   | urzędnik           |
| Gdynia   | urzędnik           |

```
SELECT Miasto, Stanowisko
```

```
FROM NAZWISKA
```

*Wynik zapytania*

```
GROUP BY Miasto, Stanowisko
```

```
ORDER BY Stanowisko;
```



## GROUP BY stosowane łącznie z WHERE

Klauzule **WHERE** można użyć łącznie z **GROUP BY**, aby ograniczyć ilość wierszy zanim będą dzielone na grupy i podgrupy

- Można dla poprzedniego zapytania wprowadzić ograniczenie na stanowiska, na których pensja jest większa od 2 000 zł

- Postać zapytania: **Miasto Stanowisko**

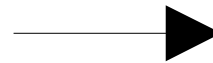
```
SELECT Miasto, Stanowisko
```

```
FROM NAZWISKA
```

```
WHERE Pensja > 2000 Wynik zapytania
```

```
GROUP BY Miasto, Stanowisko
```

```
ORDER BY Stanowisko;
```



|          |            |
|----------|------------|
|          |            |
| Gdansk   | informatyk |
| Sopot    | kierownik  |
| Warszawa | kontroler  |
| Londyn   | prezes     |

- 
- Przykład zapytania o sume do wypłaty w poszczególnych miastach:

```
SELECT Miasto, SUM(Pensja) AS WYPLATA  
FROM NAZWISKA  
GROUP BY Miasto  
ORDER BY Miasto;
```

*Wynik zapytania* →

| <b>Miasto</b> | <b>WYPLATA</b> |
|---------------|----------------|
| Gdańsk        | 6 400,00 zł    |
| Gdynia        | 1 100,00 zł    |
| Londyn        | 8 000,00 zł    |
| Sopot         | 3 000,00 zł    |
| Warszawa      | 3 000,00 zł    |

## Filtrowanie wyników zapytan z uzyciem HAVING

- Język SQL dostarcza jeszcze jedną metodę filtrowania wyników zapytania w połączeniu z klauzulą **GROUP BY**
- Klauzula **WHERE** filtruje wyniki zapytania zanim są one grupowane, natomiast klauzula **HAVING** filtruje wyniki po wykonaniu grupowania
- Wyrażenia zawarte w tej klauzuli wykonywane są na całych grupach, a nie na pojedynczych rekordach
- Postać polecenia – wybierającego te miasta, dla których suma wyplac jest wyższa od 3 000 zł

```
SELECT Miasto, SUM(Pensja) AS SUMA
```

```
FROM NAZWISKA
```

```
GROUP BY HAVING SUM Miasto (Pensja) > 3000;
```

*zapytania*

Londyn

| Miasto | SUMA        |
|--------|-------------|
| Gdansk | 6 400,00 zł |
|        |             |
| Wynik  |             |
|        | 8 000,00 zł |

- Funkcje agregujące są użyte w dwóch miejscach, w klauzuli **SELECT** oraz **HAVING**
- W **HAVING** musi się znajdować takie samo wyrażenie jak na liście klauzuli **SELECT**

## **Filtrowanie wyników zapytań z użyciem HAVING**

- Nazwy kolumn, które nie pojawiają się na liście klauzuli **SELECT**, nie mogą być w ogóle użyte w klauzuli **GROUP BY**
- Klauzula **HAVING** pojawia się przed **ORDER BY** ale za **GROUP BY**
- W obrębie klauzuli **HAVING**, można używać złożonych wyrażeń
- Jedyne ograniczenie polega na tym, że wszystkie wyrażenia w części **HAVING** muszą mieć swój odpowiednik na liście klauzuli **SELECT**
- **HAVING** i **WHERE** mogą być stosowane w jednym zapytaniu
- Wynikiem poniższego zapytania będzie lista stanowisk, na których zatrudnionych jest więcej niż jedna osoba, wraz z podaniem średniej pensji dla danego stanowiska

```
SELECT Stanowisko, COUNT(Stanowisko), AVG(Pensja)
```

```
FROM NAZWISKA
```

*Wy* *nik zapytania*  

```
GROUP BY
```

```
Stanowisko
```

```
HAVING COUNT(Stanowisko) > 1;
```



| Stanowisko | Expr1001 | Expr1002    |
|------------|----------|-------------|
| stażysta   | 2        |             |
| urzędnik   | 2        | 1 000,00 zł |

## Tworzenie nowej tabeli

- Do zdefiniowania nowej tabeli używamy instrukcji **CREATE TABLE**, której najprostsza instrukcja wygląda następująco: **CREATE TABLE**

```
Nazwa_tabeli
```

```
(nazwa_kolumny typ_danych[(rozmiar)],
```

```
 nazwa_kolumny typ_danych[(rozmiar)], ...)
```

- Każda kolumna musi mieć określony typ danych
- Dla większości typów danych wymagane jest także określenie rozmiaru

- W instrukcji **CREATE TABLE** istnieje możliwość zdefiniowania klucza głównego, określenie relacji z innymi tabelami, wprowadzenie ograniczeń na wartości kolumn itp.
- **Typy danych w definiowaniu tabel w SQL**
- Do zdefiniowania tabeli konieczne jest podanie typu danych
- Nie można stosować nazw typów używanych w Accessie, takich jak: Autonumerowanie, Tekst, Nota, Liczba, Data/godzina, Walutowy, Tak/Nie, Obiekt OLE, Hiperlacze

## **Typy danych**

- Typ danych determinuje nie tylko sposób przechowywania danych na dysku, ale co ważniejsze, sposób interpretacji tych danych
- Niemniej ważne są wymagania dotyczące zajmowania pamięci
- Marnotrawstwem byłoby zarezerwowanie 255 bajtów dla pola, które wykorzystuje tylko 2 bajty, a z drugiej strony zarezerwowanie 5 bajtów dla numeru telefonu, może nie być wystarczające
- Relacyjne bazy danych dostarczają bardzo bogaty zestaw typów danych

- Istnieją typy danych tekstowych, liczby, typy określające czas oraz obiekty, dane binarne czy duże teksty
- Każda baza danych posiada swoje własne zestawy typów danych, mogące się różnić pomiędzy sobą nazwami
- Niektóre systemy baz danych udostępniają również podtypy, jak np. dla typu liczbowego, może to być liczba całkowita, zmiennoprzecinkowa czy waluta
- Większość baz danych obsługuje podstawowe typy, choć pomiędzy różnymi produktami nie ma pełnej zgodności

## **Typy danych**

- Cztery kategorie typów: dane łańcuchowe, numeryczne, określające czas i duże obiekty
- Dane łańcuchowe mogą przechowywać właściwie każdy typ danych z zastrzeżeniem, że dane te są traktowane tylko jako łańcuch znaków
- Dane numeryczne i określenia czasu umożliwiają wykonywanie działań matematycznych oraz innych funkcji do przetwarzania danych

- Duże obiekty, służą do gromadzenia dużych ilości informacji - są one traktowane odmiennie od innych typów danych, np. nie można porównywać takich obiektów
- Ważna różnica między typami danych polega na sposobie traktowania ich przez język SQL - dane łańcuchowe, okreslenia czasu i duże obiekty muszą być w instrukcjach SQL zawarte w pojedynczych cudzysłowach, natomiast dane numeryczne nie są zapisywane w cudzysłowach
- W większości baz danych mamy do dyspozycji dwa rodzaje typów łańcuchowych o ustalonej długości i o zmiennej długości
- Ustalona długość powoduje zawsze rezerwację takiej samej ilości pamięci, bez względu na wymagania danych, natomiast zmienna długość zużywa tylko tyle pamięci, ile jest potrzebne dla konkretnej wartości

## Typy danych – dane znakowe

- **Typy łańcuchowe**
- **CHAR** jest typem danych o ustalonej długości - **CHAR(wymiar)**
- W polu typu **CHAR** miejsce nie zużyte przez dane jest automatycznie uzupełniane spacjami



- **VARCHAR** jest typem danych o zmiennej dlugosci – **VARCHAR(wymiar)**
- Przy deklaracji tego typu danych okreslamy maksymalna dlugosc
- Różnica między **VARCHAR(50)** a **CHAR(50)** polega na tym, że pole o zmiennej dlugosci dostosowuje potrzebną pamięć do rzeczywistej dlugosci lancucha danych
- W przypadku, gdy chcemy zapamiętać większą ilość danych znakowych mamy do dyspozycji specjalny typ dla dużych obiektów tekstowych
- W Oracle jest to **CLOB** – Character Large Object a w Microsoft SQL Server jest typ **TEXT**.
- W Accessie jest to typ **MEMO**

## **Typy danych - dane numeryczne**

- Czasami dane numeryczne przechowuje się w polu znakowym, np. kod pocztowy, czy numer telefonu lepiej zapamiętać w polu tekstowym, mimo, że składają się z cyfr

- Większość baz danych dostarcza dwóch typów numerycznych, jeden dla liczb całkowitych, drugi dla zmiennoprzecinkowych
- Czasami mamy jeszcze bardziej szczegółowe jak **MONEY**, który automatycznie przydziela dwa miejsca po przecinku i znak waluty
- Liczba cyfr obsługiwana przez pole numeryczne może się różnić w zależności od bazy danych, a w wielu przypadkach można o tym zdecydować przy definicji, podobnie jak w typie **CHAR**

### Typ danych

### Definicja

**DECIMAL** Liczba zmiennoprzecinkowa **FLOAT** Liczba zmiennoprzecinkowa

**INTEGER(rozmiar)** Liczba całkowita o określonej długości

**MONEY** Liczba posiadająca dwie pozycje dziesiętne

**NUMBER** Standardowa liczba zmiennoprzecinkowa

Kolejny typ danych określa datę i czas - w Accessie jest to typ **DATE**

## Okreslanie kluczy

- Tworząc tabele, można zdefiniować zarówno klucz główny jak i klucze kandydujące
- Słowo **UNIQUE** służy do określenia, która kolumna (lub grupa kolumn) musi być unikalna i jest przez to kluczem kandydującym
- Użycie ograniczenia **UNIQUE** powoduje, że próba powtórzenia danych w tych kolumnach będzie przez bazy danych powstrzymana
- Definicja klucza głównego znajduje się po definicjach pól, jeżeli klucz główny składa się z kilku pól podaje się listę nazw pól oddzieloną przecinkami
- Zdefiniowanie klucza głównego wymaga użycia klauzuli **PRIMARY KEY**
- Oczywiście w tabeli może być zidentyfikowany jeden klucz główny
- Kolejny przykład przedstawia polecenie tworzące tabelę o nazwie **NOWA** zawierającą osiem pól różnych typów oraz zdefiniowany klucz główny

## Przykład tworzenia nowej tabeli

- Postać polecenia, tworzącego tabelę o nazwie **NOWA**, w której kluczem głównym jest pole **Nr\_ident**, a kluczem kandydującym jest pole **Telefon**:

**CREATE TABLE** NOWA definicja nazwy tabeli

(Nr\_ident **INTEGER**, pole typu całkowitego Zawód

**CHAR(20)**, pole znakowe o stałej długości

Telefon **VARCHAR(15)**, pole znakowe o zmiennej długości Data\_rozp

**DATE**, pole zapamiętujące datę i czas

Premia **MONEY**, pole walutowe

Prawo\_jazdy **LOGICAL**, pole typu logicznego Uwagi **MEMO**,

pole dużego obiektu znakowego

**UNIQUE** (Telefon), definicja klucza kandydującego

**PRIMARY KEY** (Nr\_ident)) definicja klucza głównego

- Można definiować klucze również w linii definiującej kolumnę
- np.: (Nr\_ident **INTEGER PRIMARY KEY**,

- Klucze obce - klauzula **REFERENCES** służy do ustalenia relacji między tabelami

## Przykład tworzenia nowej tabeli

- **Odrzucanie wartości NULL** - zapobiega wprowadzaniu wartości NULL do kolumny. Użycie **NOT NULL** w definicji kolumny wymusza podanie wartości dla takiej kolumny przy każdym wprowadzaniu nowego wiersza
  - Zapobiega to zmianie wartości na NULL przy aktualizacji danych w tabeli
- Taki sam efekt daje zdefiniowanie klucza głównego.
- Postać polecenia tworzącego tabelę z ustaleniem relacji między polem **Nr\_ident** z tabeli **NOWA** z polem **Numer** z tabeli **NAZWISKA** oraz zabezpieczeniem przed wartościami NULL dla pól **Zawód** i **Data\_rozp**:  
**CREATE TABLE** NOWA

```
(Nr_ident INTEGER PRIMARY KEY REFERENCES Nazwiska (Numer) ,
Zawód CHAR(20) NOT NULL ,
Telefon VARCHAR(15) ,
```

Data\_rozp     **DATE**     **NOT**     **NULL,**  
Premia **MONEY,**  
Prawo\_jazdy **LOGICAL,**  
Uwagi **MEMO)**

- Dane wprowadza się przy pomocy instrukcji **INSERT**
- Do wprowadzania zmian służą instrukcje **UPDATE** i **DELETE** (do kasowania)
- Do usuwania tabeli z bazy danych służy instrukcja **DROP**
- **Instrukcja INSERT** - jest to jedyna instrukcja języka SQL służąca do dopisywania nowych rekordów do tabel
- Podstawowa struktura instrukcji **INSERT** jest następująca: **INSERT INTO**  
nazwa\_tabeli  
[(lista kolumn)]  
**VALUES**  
(lista wartosci)
- **Nazwa\_tabeli** określa tabelę, do której wprowadza się nowy rekord
- W przypadku, gdy wprowadza się wartości tylko dla niektórych kolumn, należy podać nazwy kolumn, do których mają być wprowadzone wartości

- 
- Pominiecie listy kolumn w instrukcji **INSERT** wymusza podanie wartości dla wszystkich kolumn w tabeli

Postać polecenia wprowadzającego pełny rekord danych

```
INSERT INTO NOWA  
VALUES (3, 'prawnik', '345 89 98', '1999-08-05', 1200, 1,  
'wyjazd w grudniu');
```

- Postać polecenia wprowadzającego dane do wybranych kolumn **INSERT INTO** NOWA

```
(Nr_ident, Zawód, Data_rozp)  
VALUES (4, 'ekonomista', '2002-01-01');
```

- Musza być wypełnione te pola, które są NOT NULL i klucz główny • Pole **Zawód** jest dopełniane spacjami do długości 20 znaków
- **Instrukcja DELETE** - służy do usuwania rekordów z tabeli.
- Podstawowa struktura instrukcji **DELETE**: **DELETE FROM** tabela  
[**WHERE** warunek]



- 
- Opcjonalna część z klauzula **WHERE** jest wykorzystywana do ograniczania rekordów, które zostaną usunięte
- Pominięcie tej części powoduje, że wszystkie rekordy są usuwane. Postać polecenia usuwającego z tabeli **NOWA**, wszystkie rekordy pracowników nie będących ekonomistami:  

```
DELETE FROM NOWA  
WHERE Zawód <> 'ekonomista';
```
- Postać polecenia usuwającego wszystkie rekordy z tabeli **NOWA**: **DELETE FROM NOWA**
- **Instrukcja UPDATE** - jest wykorzystywana do wprowadzania zmian w istniejących rekordach
- Struktura instrukcji jest następująca: **UPDATE** tabela  

```
SET kolumna = wartosc, ...  
[WHERE warunek]
```
- Instrukcja składa się z trzech części:
  - W pierwszej części określa się, jaka tabela będzie aktualizowana

- - Druga część – klauzula **SET** – służy do podania listy kolumn, które będą zmieniane i nowych wartości, które zostaną przypisane tym kolumnom
  - W ostatniej części za pomocą klauzuli **WHERE** określa się wiersze tabeli, w których nastąpi zmiana

# Tworzenie, zmienianie i usuwanie rekordów

59

- Postać polecenia zmieniającego zawartość pola **Premia** (było 1200) na 500 dla pracownika o **Nr\_ident** równym 3:  

```
UPDATE NOWA  
SET Premia = 500  
WHERE Nr_ident = 3;
```
- **Instrukcja DROP** - służy do usuwania tabel z bazy danych
- Przy ustalaniu nowych wartości określonego pola można zastosować wyrażenia arytmetyczne
- Przykładowe polecenie spowoduje zwiększenie wszystkim pracownikom premii o 100 zł  

```
UPDATE NOWA  
SET Premia = Premia+100;
```
- Postać polecenia usuwającego tabelę z bazy:  

```
DROP TABLE Nazwa_tabeli
```
- W wielu przypadkach w trakcie wyszukiwania informacji z bazy danych okazuje się, że potrzebne dane przechowywane są w kilku tabelach

# Laczenie tabel

- W celu polaczenia danych z wielu tabel w jednym zapytaniu wymagane jest zlaczenie
- **Polaczenia i normalizacja**
- Efektem normalizacji jest rozbicie bazy danych na wiele tabel
- Uzywajac zlaczen miedzy tabelami mozna wybierac informacje z wielu tabel za pomoca pojedynczej instrukcji **SELECT**
- Daje to efekt ponownego polaczenia danych, które zostały rozdzielone do wielu tabel w trakcie normalizacji
- Zlaczenie to zapytanie, które laczy dane z wielu tabel
- Struktura standardowego zapytania jest nastepujaca: **SELECT**

```
lista_kolumn
```

```
    FROM tabela1, [tabela2, ...]
```

```
    WHERE warunek;
```

- W czesci **FROM** pojawiaja sie deklaracje kilku tabel, reszta nie różni się od polecenia działającego na jednej tabeli

# Laczenie tabel

- Problem z instrukcją SELECT polega na tym, że zwraca ona każdą kombinację wierszy z dwóch tabel - jeżeli jedna tabela zawiera 8 wierszy a druga 10 wierszy to zapytanie zwróci 80 wierszy
- Zapytanie poniżej, przy założeniu 9 wierszy w pierwszej i 9 w drugiej tabeli da w wyniku tabelę z 81 wierszami

```
SELECT *
```

```
FROM NAZWISKA, NOWA;
```

- Tworzenie sensownych złączeń wymaga spełnienia dwóch warunków
  - Należy wybrać w każdej tabeli kolumny, które są ze sobą w logiczny sposób powiązane z kolumnami z drugiej tabeli
  - Musi być zdefiniowane kryterium określające warunki złączenia dwóch tabel
- **Zgodne kolumny** - aby połączenie dwóch tabel miało sens, muszą one mieć jakieś wspólne dane
- W przypadku tabel **Nazwiska** i **Nowa** mogą to być kolumny określające numer identyfikacyjny pracownika **Numer** i **Nr\_ident** odpowiednio
- Klauzula **WHERE** określa drugi warunek wymagany w złączeniu

# Laczenie tabel

- Zwykle najefektywniejsze polaczenia osiaga się poprzez kolumny bedace kluczami w laczonych tabelach, np. zawsze mozna dokonac polaczenia tabel, jesli klucz główny jednej tabeli jest kluczem obcym w drugiej
- W przypadku, gdy laczymy tabele, w której klucz główny stanowi kilka kolumn, nalezy uzyc wszystkich kolumn klucza przy okreslaniu warunków polaczenia
- Warunki zwykle okresla sie w klauzuli **WHERE**, inaczej nalezy okreslic, jakie wiersze tabeli pierwszej maja byc polaczone z wierszami z tabeli drugiej
- Wartosci NULL nigdy nie sa traktowane jako spelniajace warunek zlaczenia - wiersze, dla których w obu kolumnach laczacych znajduja sie wartosci NULL sa pomijane w wyniku zapytania.
- Wynika to stad, ze NULL traktowane sa jak wartosci nieokreslone i w zwiazku z tym nie moga podlegac operacjom porównania
- **Wybieranie kolumn** - tworzac zapytanie laczace kilka tabel rzadko wybieramy wszystkie kolumny przy pomocy szablonu \*

# Laczenie tabel

- Użycie go powoduje, że wszystkie kolumny ze wszystkich tabel pojawiają się w zestawieniu wynikowym
- **Skracanie nazw tabel – aliasy** - zamiast używać pełnych nazw tabel można utworzyć aliasy dla nazw
- Polega to na podaniu zaraz za nazwą tabeli jej skróconej nazwy poprzedzonej słowem **AS** (jeżeli słowo kluczowe **AS** nie zostanie wpisane, system doda je automatycznie)
- Zadaniem jest sformułowanie zapytania wybierającego z tabeli **NAZWISKA** kolumn **Numer, Imie, Nazwisko i Pensja** a z tabeli **NOWA** kolumny **Premia** z dodaniem pola wyliczającego sumę do wypłaty (**Pensja** z tabeli **NAZWISKA** i **Premia** z tabeli **NOWA**) – pole to zostało nazwane **WYPŁATA**
- Zastosowano aliasy **N** dla tabeli **NAZWISKA** i **P** dla tabeli **NOWA**

# Laczenie tabel

Postać zapytania:

```
SELECT N.Numer, N.Imie, N.Nazwisko, N.Pensja, P.Premia,
N.Pensja+P.Premia AS WYPLATA
FROM NAZWISKA AS N, NOWA AS P
WHERE N.Numer = P.Nr_ident;
```

- Wynik działania polecenia:

| Numer | Imie       | Nazwisko   | Pensja      | Premia      | WYPLATA      |
|-------|------------|------------|-------------|-------------|--------------|
| 1     | Jan        | Kowalski   | 900,00 zł   | 600,00 zł   | 1 500,00 zł  |
| 2     | Waldemar   | Pawlak     | 3 000,00 zł | 200,00 zł   | 3 200,00 zł  |
| 3     | Marian     | Malinowski | 1 100,00 zł | 1 200,00 zł | 2 300,00 zł  |
| 4     | Adam       | Nowak      | 2 000,00 zł | 900,00 zł   | 2 900,00 zł  |
| 5     | Ewa        | Musial     |             |             |              |
| 6     | Zenon      | Miler      |             |             |              |
| 7     | Paul       | Davies     | 8 000,00 zł | 4 000,00 zł | 12 000,00 zł |
| 8     | Mieczyslaw | Dobija     | 3 000,00 zł | 5 000,00 zł | 8 000,00 zł  |



- |   |       |        |             |             |             |
|---|-------|--------|-------------|-------------|-------------|
| 9 | Peter | Norton | 3 500,00 zł | 1 000,00 zł | 4 500,00 zł |
|---|-------|--------|-------------|-------------|-------------|
- **Zlaczania i relacje** - relacja jeden do wiele zachodzi, gdy jednemu z dwóch obiektów relacji odpowiada wiele pozycji drugiego obiektu, ale każdej pozycji drugiego obiektu odpowiada tylko jedna pozycja obiektu pierwszego
- Przykładem jest tabela zawierająca listę nazwisk **NAZWISKA** oraz tabela **ZLECENIA**, w której rejestrowane są dane o zleceniach realizowanych przez poszczególnych pracowników
- Zawartość tabeli **ZLECENIA**:

| Nr_zlec | Nr_prac | Kod_zlecenia | Wartosc_zlecenia |
|---------|---------|--------------|------------------|
| 1       | 1       | Z-001        | 500,00 zł        |
| 2       | 2       | Z-002        | 3 000,00 zł      |
| 3       | 3       | Z-003        | 700,00 zł        |
| 4       | 4       | Z-004        | 300,00 zł        |
| 5       | 1       | Z-005        | 400,00 zł        |

# Laczenie tabel

|   |   |       |             |
|---|---|-------|-------------|
| 6 | 3 | Z-006 | 500,00 zł   |
| 7 | 3 | Z-007 | 900,00 zł   |
| 8 | 2 | Z-008 | 1 000,00 zł |

Postac polecenia laczonego tabele **NAZWISKA** i **ZLECENIA**:

```
P.WartoSELECTsc_zlecenia      N.Numer,      N.Imie,      N.Nazwisko,  
P.Kod_zlecenia,
```

```
FROM Nazwiska AS N, Zlecenia AS P WHERE  
N.Numer = P.Nr_prac  
ORDER BY N.Numer;
```

- Wynik dzialania polecenia: • Jest to przyklad

| Numer | Imie     | Nazwisko   | Kod_zlecenia | Wartosc_zlecenia | relacji jeden do wiele:<br>pole <b>Numer</b> jest<br>kluczem głównym<br>tabeli <b>Nazwiska</b> a pole<br><b>Nr_prac</b> w tym<br>przypadku jest kluczem<br>obcym w tabeli<br><b>Zlecenia</b> |
|-------|----------|------------|--------------|------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1     | Jan      | Kowalski   | Z-005        | 400,00 zł        | <ul style="list-style-type: none"> <li>• Na wyniki<br/>koncowe zapytania<br/>sklada sie kilka<br/>etapów przetwarzania</li> </ul>                                                            |
| 1     | Jan      | Kowalski   | Z-001        | 500,00 zł        |                                                                                                                                                                                              |
| 2     | Waldemar | Pawlak     | Z-008        | 1 000,00 zł      |                                                                                                                                                                                              |
| 2     | Waldemar | Pawlak     | Z-002        | 3 000,00 zł      |                                                                                                                                                                                              |
| 3     | Marian   | Malinowski | Z-007        | 900,00 zł        |                                                                                                                                                                                              |
| 3     | Marian   | Malinowski | Z-006        | 500,00 zł        |                                                                                                                                                                                              |
| 3     | Marian   | Malinowski | Z-003        | 700,00 zł        |                                                                                                                                                                                              |
| 4     | Adam     | Nowak      | Z-004        | 300,00 zł        |                                                                                                                                                                                              |

danych

- Poszczególne kroki sa szczególnie wazne w przypadku zapytan laczacych, poniewaz ilustrują problemy zwiazane z wydajnoscia takich zapytan

# Laczenie tabel

- Na początku pojawia sie iloczyn kartezyjanski z wierszy laczonych tabel
- Jest to kombinacja wszystkich wierszy z pierwszej tabeli, z wszystkimi wierszami z drugiej tabeli
- Dla trzech tabel o 50, 100 i 10 wierszach wynikowa tabela ma 50 000 wierszy co jest przyczyna spadku wydajnosci instrukcji **SELECT**
- Iloczyn kartezyjanski stanowi tabele dla dalszego zapytania, która zachowuje kolejnosc wierszy z tabel laczonych
- Kolejny krok polega na wykonaniu ograniczen wynikajacych z klauzuli **WHERE** - wszystkie wiersze, dla których wynik wyrażenia w klauzuli **WHERE** jest prawdziwy, sa wybierane
- Do tej pory nie byla wykonana selekcja kolumn z tabeli wiec w klauzuli **WHERE** moga znajdowac sie odwołania do dowolnej kolumny tabeli Kolejny krok, jesli w zapytaniu obecna jest klauzula **GROUP BY**, polega na sortowaniu pozostalych wierszy w tabeli wedlug wybranych kolumn
- Jesli na liscie **SELECT** znajduja sie funkcje agregujace, to w tym momencie sa one wykonywane, a tabela zostaje zastapiona przez nowa,

- zawierająca wyniki funkcji agregujących dla grup (jesli **GROUP BY** jest uzyte)
- Następnie klauzula **HAVING** jest stosowana dla tabeli podzielonej na grupy, wiersze nie spełniające warunków określonych w tej części są odrzucane
- Na końcu wybrane są z tabeli kolumny zawarte na liście **SELECT**, wyliczone odpowiednie wyrażenia i tak powstaje wynik końcowy
- Przykład zapytania z funkcją agregującą i grupowaniem – suma zleceń poszczególnych pracowników

```
SELECT    N.Numer,    N.Imie_zlecenia)    AS    SUMA_ZLECE,  
    N.Nazwisko, N  
SUM(P.Warto  
FROM Nazwiska AS N, Zlecenia AS P WHERE  
    N.Numer=P.Nr_prac  
GROUP BY N.Numer, N.Imie, N.Nazwisko  
ORDER BY N.Numer;
```

# Laczenie tabel

- Wynik działania poprzedniego polecenia:

| Numer | Imie     | Nazwisko   | SUMA_ZLECEN |
|-------|----------|------------|-------------|
| 1     | Jan      | Kowalski   | 900,00 zł   |
| 2     | Waldemar | Pawlak     | 4 000,00 zł |
| 3     | Marian   | Malinowski | 2 100,00 zł |

- Warunki złączenia mogą być formułowane w sposób złożony, np. można wybrać zlecenia poszczególnych pracowników, które przekraczają 900 zł:

```
P.Warto SELECT sc_zlecenia      N.Numer,      N.Imie,      N.Nazwisko,
P.Kod_zlecenia,
```

```
FROM Nazwiska AS N, Zlecenia AS P
```

```
WHERE N.Numer=P.Nr_prac AND P.Wartosc_zlecenia >= 900
```

```
ORDER BY N.Numer;
```

```
Numer      Imie      Nazwisko      Kod_zlecenia      Wartość_zlecenia
```

# Laczenie tabel

- |   |          |            |       |             |
|---|----------|------------|-------|-------------|
| 1 | Waldemar | Pawlak     | Z-008 | 1 000,00 zł |
| 2 | Waldemar | Pawlak     | Z-002 | 3 000,00 zł |
| 3 | Marian   | Malinowski | Z-007 | 900,00 zł   |

---

## Laczenie wiecej niz dwóch tabel

Przykladem jest zapytanie o dane pracownika, wysokosc pensji oraz premii oraz sume zleceń i sume do wypłaty

- Dane te zawarte są w trzech tabelach **NAZWISKA**, **NOWA** i **ZLECENIA**
- Postać zapytania:

```
SUMSELECT (P.Warto  N.Numer,  N.Imisc_zlecenia)  e,  N.Nazwisko,
N.Pensja,  R.Premia, AS SUMA_ZLECEN,
      Suma_zleceń+N.Pensja+R.Premia AS WYPLATA
FROM Nazwiska AS N, Zlecenia AS P, Nowa AS R
WHERE N.Numer = P.Nr_prac AND N.Numer = Nr_ident
GROUP BY N.Numer, N.Imie, N.Nazwisko, N.Pensja, R.Premia ORDER
BY N.Nazwisko;
```

| Numer | Imie | Nazwisko | Pensja | Premia | SUMA_ZLECEN | WYPLATA |
|-------|------|----------|--------|--------|-------------|---------|
|-------|------|----------|--------|--------|-------------|---------|



- |   |          |            |             |             |             |             |
|---|----------|------------|-------------|-------------|-------------|-------------|
| 1 | Jan      | Kowalski   | 900,00 zł   | 600,00 zł   | 900,00 zł   | 2 400,00 zł |
| 3 | Marian   | Malinowski | 1 100,00 zł | 1 200,00 zł | 2 100,00 zł | 4 400,00 zł |
| 4 | Adam     | Nowak      | 2 000,00 zł | 900,00 zł   | 300,00 zł   | 3 200,00 zł |
| 2 | Waldemar | Pawlak     | 3 000,00 zł | 200,00 zł   | 4 000,00 zł | 7 200,00 zł |
- Wynik zapytania:

## Unie

- Unia umożliwia łączenie wyników kilku zapytań w jednym zestawieniu
- Wyniki pojawiają się jakby zostały wybrane z jednej tabeli, podczas gdy pochodzą z wielu tabel
- Działanie operatora **UNION**
- Przykład zapytania, które powinno pobrać z tabeli **NAZWISKA** kolumny **Numer**, **Nazwisko** i **Imie** oraz z tabeli **NOWA** kolumny **Nr\_ident**, **Zawód** i **Premia** – dla trzech pierwszych pracowników (na podstawie numeru identyfikacyjnego pracownika)

- Postać zapytania:

```
SELECT Numer, Nazwisko, Imie FROM
NAZWISKA
```

```
WHERE Numer <= 3 UNION Wynik
zapytania
```

```
SELECT Nr_ident, Zawód, Premia
FROM NOWA
WHERE Nr_ident <= 3;
```

| Numer | Nazwisko   | Imie     |
|-------|------------|----------|
| 1     | ekonomista | 600      |
| 1     | Kowalski   | Jan      |
| 2     | ekonomista | 200      |
| 2     | Pawlak     | Waldemar |
| 3     | Malinowski | Marian   |
| 3     | prawnik    | 1200     |

- Wyniki posortowane według kolumn idac od lewej

## Unie

Aby wyniki były bardziej czytelne można zmienić nazwy kolumn i dodać opcję **ALL** do operatora **UNION**

- Użycie operatora **UNION** do połączenia kilku zapytań powoduje, że żadne powtarzające się wiersze nie są wybierane, a wyniki są automatycznie sortowane według kolumn od lewej do prawej •

Wszystkie wiersze bez sortowania pojawiają się po użyciu opcji **ALL**

- 
- Ilustruje to zapytanie sformułowane poniżej:

```
SELECT Numer, Nazwisko AS Kolumna_1, Imie AS Kolumna_2
```

```
FROM NAZWISKA
```

```
WHERE Numer <= 3 Wynik zapytania
```

```
SELECT FROM WHERE UNION ALL NOWA Nr_ident <= 3; Nr_ident, Zawód,
```

```
Premia 231
```

```
600
```

```
Pawlak Malinowski ekonomista Waldemar Marian
```

| Numer | Kolumna_1 | Kolumna_2 |
|-------|-----------|-----------|
| 1     | Kowalski  | Jan       |
|       |           |           |
|       |           |           |
|       |           |           |
|       |           |           |
|       |           |           |
|       |           |           |
|       |           |           |
|       |           |           |

```
2 ekonomista 200
```

```
3 prawnik 1200
```

## Podzapytania

- W niektórych przypadkach najprostsza metoda osiągnięcia jakiegoś celu w języku SQL jest wykorzystanie wyniku jednego zapytania w drugim
- Zagnieżdzone zapytania nazywane również podzapytaniem, mogą być używane w klauzuli WHERE do filtrowania danych
- Podzapytanie używamy, gdy dane z pewnej tabeli są potrzebne w innym zapytaniu
- **Podzapytanie to, najprościej mówiąc, instrukcja SELECT zagnieżdżona w innej instrukcji SQL, która dostarcza dla tej drugiej danych wejściowych**
- Podzapytanie jest zapytaniem zagnieżdżonym
- Zapytanie otaczające też może być podzapytaniem, ponieważ SQL nie wprowadza ograniczeń w ilości zagnieżdżeń
- Jeśli zapytanie jest podzapytaniem, to kolejne zapytanie może pojawić się w jego klauzuli **WHERE**

- 
- Zapytanie otaczające określa się czasem jako zapytanie zewnętrzne a zapytanie zagnieżdżone jako wewnętrzne

## Wprowadzenie - podzapytanie w wyrażeniu IN

Sformułować zapytanie, które da w wyniku listę nazwisk pracowników, którzy mają zarejestrowaną realizację zleceń na kwotę  $\geq 900$  zł

- Postać zapytania:

```
SELECT Numer, Imie, Nazwisko
```

```
FROM NAZWISKA
```

```
WHERE Numer IN      Wynik działania zapytania
```

```
(SELECT Nr_prac
```

```
FROM ZLECENIA
```

```
WHERE Wartosc_zlecenia  $\geq$  900);
```

| Numer | Imie     | Nazwisko   |
|-------|----------|------------|
| 2     | Waldemar | Pawlak     |
| 3     | Marian   | Malinowski |

- Zapytanie wewnętrzne (zagnieżdżone) dostarcza danych (lista zleceń o wartości  $\geq 900$  zł) do zapytania otaczającego z **IN**

- Na podstawie pola **Nr\_prac** wybiera się z tabeli **NAZWISKA** dane pracownika (**Imie** i **Nazwisko**)
- Ten sam wynik można uzyskać stosując złączenie, co ilustruje kolejne zapytanie:

```
SELECT DISTINCT Numer, Imie, Nazwisko
```

```
FROM NAZWISKA, ZLECENIA
```

```
WHERE Numer = Nr_prac AND Wartosc_zlecenia >= 900;
```

# Typy podzapytan

- 
- Wyróżniamy dwa typy podzapytan: **powiazane** i **niepowiazane**
- **Podzapytanie powiazane** wymaga danych z zapytania otaczajacego, zanim moze byc wykonane - wykonuje sie je wykorzystujac dane z zapytania otaczajacego, a dane przez nie zwrócone sa z powrotem wprowadzane do zapytania do zapytania otaczajacego
- **Podzapytania niepowiazane** wykonuje sie przed zapytaniem otaczajacym, a jego wyniki sa przekazywane do zapytania otaczajacego
- Podzapytanie niepowiazane mozna poznac po tym, ze nie zawiera zadnych odwołan do zapytania otaczajacego - przykladem jest sformulowane poprzednio zapytanie
- Podzapytanie wybiera liste identyfikatorów pracowników, którzy mieli zarejestrowane zlecenia na kwoty powyzej 900 zł, która to lista jest wykorzystywana w zapytaniu otaczajacym w klauzuli **IN**
- Podzapytanie to w zaden sposób nie zalezy od otaczajacego je zapytania
- Podzapytanie jest wykonywane, a wyniki sa porównywane z wartosciami z tabeli określonej w zapytaniu otaczajacym

# Typy podzapytan

Zapytanie powiazane różni sie od zapytania niepowiazanego tym, ze pozycje z listy **SELECT** zapytania otaczajacego sa wykorzystane wewnatrz klauzuli **WHERE** podzapytania

- Zapytanie powiazane przypomina zlaczenia, poniewaz zawartosc tabeli wystepujacej w podzapytaniu bedzie porównywana z zawartoscia tabeli z zapytania otaczajacego, podobnie jak w zapytaniu zlaczajacym
- Różnica polega na tym, ze zamiast warunku zlaczajacego, powiazane podzapytanie odwołuje sie do zapytania zewnetrznego przez klauzule **WHERE** zapytania wewnetrznego
- Przykład zapytania dajacego w wyniku liste nazwisk pracowników bedacych ekonomistami

```
SELECT Numer, Imie, Nazwisko
```

```
FROM NAZWISKA AS N
```

```
WHERE 'ekonomista' IN
```

```
(SELECT Zawód
```

```
FROM NOWA
```

*Wynik dzialania zapytania* ►

| <b>Numer</b> | <b>Imie</b> | <b>Nazwisko</b> |
|--------------|-------------|-----------------|
| 7            | Paul        | Davies          |
| 8            | Mieczyslaw  | Dobija          |
| 1            | Jan         | Kowalski        |
| 4            | Adam        | Nowak           |
| 2            | Waldemar    | Pawlak          |



# Typy podzapytan

- **WHERE** N.Numer = Nr\_ident) **ORDER BY** Nazwisko;
- Przedstawione zapytanie przetwarza każdy wiersz z tabeli **NAZWISKA** w sposób następujący:
  - odczytywana jest zawartość wiersza,
  - wykonuje się podzapytanie, a wartości z aktualnie wybranego wiersza zapytania otaczającego są wykorzystywane w klauzuli **WHERE**, podzapytania
  - wyniki podzapytania są przekazywane do klauzuli **WHERE** zapytania otaczającego,
  - w przypadku, gdy wyrażenie logiczne w warunku klauzuli **WHERE** ma wartość prawdę, wiersz jest pobierany do zestawienia wynikowego, a w przeciwnym przypadku pomijany,
- Aby uzyskać taki sam wynik można zapytanie sformułować inaczej, jak ilustruje to kolejny przykład:  
**SELECT** Numer, Imie, Nazwisko  
**FROM** NAZWISKA **AS** N, NOWA **AS** P

# Typy podzapytan

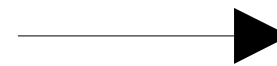
```
WHERE P.Zawód = 'ekonomista' AND N.Numer = P.Nr_ident ORDER  
BY N.Nazwisko
```

## Tworzenie zapytan z IN i NOT IN

- Wyrażenie **IN** jest wykorzystywane do sprawdzenia, czy wartość należy do pewnego zbioru
- Podzapytanie może być wykorzystane do wybrania tego zbioru wartości
- Przykład wyszukujący imiona i nazwiska osób, które realizowały prace na zleceniu

```
SELECT Imie, Nazwisko FROM NAZWISKA
WHERE Numer IN
  (SELECT Nr_prac
   FROM ZLECENIA);
```

*Wynik działania zapytania*



| Imie     | Nazwisko   |
|----------|------------|
| Jan      | Kowalski   |
| Waldemar | Pawlak     |
| Marian   | Malinowski |
| Adam     | Nowak      |

- Podobny efekt można uzyskać przez złączenie

```
SELECT DISTINCT Imie, Nazwisko
FROM NAZWISKA, ZLECENIA
WHERE Nazwiska.Numer = Zlecenia.Nr_prac;
```

- Przykład z **NOT IN** wyszukujący wszystkie osoby, które nie miały zleceń

```
SELECT Imie, Nazwisko
FROM NAZWISKA
WHERE Numer NOT IN
```

```
(SELECT Nr_prac  
FROM ZLECENIA) ;
```

## Wykorzystanie EXISTS

- Słowo kluczowe **EXISTS** zostało zaprojektowane specjalnie do wykorzystania w podzapytaniach
- Składnia instrukcji wykorzystującej słowo **EXISTS** jest następująca:  

```
SELECT lista FROM nazwa_tabeli  
WHERE EXISTS (podzapytanie) ;
```
- W przypadku, gdy podzapytanie zwraca dowolną wartość, to klauzula **EXISTS** zwraca wartość logiczną prawdę
- Klauzula **EXISTS** może być wykorzystana również w zapytaniu niepowiązanym
- W takim przypadku, gdy podzapytanie zwraca jakiegokolwiek wiersz, klauzula daje wynik prawdę, w przeciwnym przypadku fałsz
- Klauzula **EXISTS** jest bardzo przydatna w połączeniu z zapytaniami powiązаныmi

- Wykonywane są one dla każdego wiersza tabeli, a wartości aktualnie wybranego wiersza są przekazywane do klauzuli **WHERE** podzapytania
- Wykorzystując klauzule **WHERE** można porównywać dane z każdego wiersza tabeli z danymi z innych tabel

## Wykorzystanie EXISTS

- Zapytanie wyszukujące dane osób, które realizowały prace na zlecenia

```
SELECT Imie, Nazwisko
```

```
FROM NAZWISKA
```

```
WHERE EXISTS
```

*działania zapytania*

```
(SELECT Nr_prac FROM ZLECENIA
WHERE Nazwiska.Numer = Zlecenia.Nr_prac);
```

*Wynik*



| Imie     | Nazwisko   |
|----------|------------|
| Jan      | Kowalski   |
| Waldemar | Pawlak     |
| Marian   | Malinowski |
| Adam     | Nowak      |

- Klauzula **NOT EXISTS** jest wykorzystywana do znajdowania wierszy, dla których powiązane podzapytanie nie zwraca żadnych wartości
- Przydaje się to do znajdowania wierszy, które nie zawierają powiązanych danych w innych tabelach
- Zapytanie wyszukujące dane osób, które **nie realizowały** prac na zlecenia

```
SELECT Imie, Nazwisko
FROM NAZWISKA
WHERE NOT EXISTS
(SELECT Nr_prac      FROM
ZLECENIA
WHERE Nazwiska.Numer = Zlecenia.Nr_prac);
```