Bazy Danych

Andrzej M. Borzyszkowski

Instytut Informatyki
Uniwersytetu Gdańskiego

materiały dostępne elektronicznie
http://inf.ug.edu.pl/~amb

Język SQL, operowanie na danych (*data manipulation language*) c.d.

Cztery główne operacje / słowa kluczowe

- **SELECT** – główna operacja wyszukiwania danych
- **SELECT [ALL | DISTINCT]** ***lista_atrybutów_wynikowych***
  [***lista_klauzul*** **];**
  - ***lista_atrybutów_wynikowych*** rzut i zmiana nazwy kolumny
  - ***lista_klauzul*** obcięcie, złączenie, funkcje agregujące, porządkowanie
  - klauzule: **FROM WHERE ORDER BY GROUP BY HAVING**
- **INSERT** – realizuje aktualizację/wstawianie danych
- **UPDATE** – realizuje aktualizację/zmianę wartości danych
- **DELETE** – realizuje aktualizację/usuwanie danych

Instrukcja SELECT – zagnieżdżenie

- **SELECT** zwraca jako wynik zbiór krotek
  - może być on użyty jako źródło kolejnego wyszukiwania
    SELECT … FROM (SELECT …
  - może być użyty w klauzuli WHERE to testowania niepustości
    … WHERE [NOT] EXISTS (SELECT …
  - albo należenia
    … WHERE … [NOT] IN (SELECT …
  - tabela z jednym wierszem jest utożsamiana z tym wierszem i może być użyta do porównania
    … WHERE … = (SELECT …
  - tabela jednokolumnowa z jednym wierszem może być użyta jako pojedynczy element wyniku
    SELECT …, (SELECT …) FROM
  - albo jako wartość przy aktualizacji

Instrukcja SELECT

- Podaj nazwiska klientów, którzy złożyli zamówienie po 1 lutego 2025:

```
SELECT DISTINCT nazwisko
FROM klient K, zamowienie
WHERE K.nr = klient_nr AND data_zlozenia > '2025-2-1'
```

  - rozwiązanie to jest niezbyt szczęśliwe
  - jeśli dwóch 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 (**GROUP BY**)

Instrukcja SELECT – zagnieżdżenie, przykład

- Właściwe rozwiązanie:

```
SELECT nazwisko FROM klient
WHERE nr IN ( SELECT klient_nr
              FROM zamowienie
              WHERE data_zlozenia > '2025-2-1'
            )
```

  - zagnieżdżona tabela użyta w warunku, tabela jednokolumnowa służy jako zbiór wartości
  - 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

Instrukcja SELECT – zagnieżdżenie c.d.

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

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

  - wielokrotne zagnieżdżenia, trzeba rozpatrywać od wewnątrz

Instrukcja SELECT – zagnieżdżenie w klauzuli FROM, alias dla wyniku

- Oblicz i zanalizuj zysk:

```
SELECT *,
      case when zysk/koszt < 0 then 'ujemny'
           when zysk/koszt < 0.4  then 'za mało'
           when cena is NULL then 'brak danych'
           else 'ok'
      end as opinia
FROM (SELECT *, cena - koszt AS zysk FROM towar) AS
  QQ
```

  - tabela w zagnieżdżeniu ma dodatkową kolumnę, o którą można pytać
  - tabela ta musi być nazwana i wówczas może być użyta jako źródło dla kolejnego wyszukiwania
  - Postgres od pewnej wersji nie wymaga tej nazwy

Zagnieżdżenie: korelacja

- Podaj nazwiska klientów, którzy złożyli zamówienie po 1 lutego 2025:

```
SELECT nazwisko
FROM klient
WHERE EXISTS (
    SELECT *
    FROM zamowienie
    WHERE klient.nr = klient_nr AND data_zlozenia >
   '2025-2-1'
)
```

- wewnętrzny **SELECT** odwołuje się do tabeli zewnętrznej
  - jest to bardzo bliskie kwantyfikatora egzystencjalnego w rachunku krotek
  - *nie możemy* wykonać wewnętrznego zapytania w oderwaniu od reszty, *występuje* korelacja

Zagnieżdżenie: brak korelacji

- Podaj dane klientów którzy złożyli zamówienie po 1 lutego 2025

```
SELECT nazwisko FROM klient
WHERE nr IN ( SELECT klient_nr
                FROM zamowienie
                WHERE data_zlozenia > '2025-2-1'
             )
```

  - zagnieżdżona tabela użyta w warunku służy jako zbiór
  - możemy najpierw wykonać wewnętrzne zapytanie, a potem zewnętrzne
  - w zapytaniu podrzędnym nie ma odwołania do wiersza z tabeli zewnętrznej – *brak korelacji*

Zagnieżdżenie: możliwość błędów

-
```
SELECT nazwisko
FROM klient
WHERE EXISTS (
    SELECT * FROM zamowienie
    WHERE nr = klient_nr AND data_zlozenia > '2025-2-1'
)
```

- Atrybuty w wewnętrznym **SELECT** pochodzą domyślnie z wewnętrzej tabeli
  - w podanym przykładzie atrybut „nr” występuje i w tabeli klientów i zamówień
  - atrybut będzie błędnie uznany za wartość z tabeli zamówień, brak korelacji,
  - gdyby występował wyłącznie w tabeli klientów, byłaby korelacja i nie byłoby błędu

Zagnieżdżenie: korelacja c.d.

- Podaj dane klientów, których nazwiska się powtarzają:

```
SELECT imie, nazwisko, miasto
FROM klient K
WHERE EXISTS (
    SELECT *
    FROM klient
    WHERE  nazwisko=K.nazwisko AND nr != K.nr
    )
```

  - *występuje* korelacja, wewnętrzne pytanie zawiera odwołanie do wiersza z tabeli zewnętrznej
  - dodatkowo, tutaj tabela przeglądana w podrzędnym zapytaniu jest ta sama co w zewnętrznym, występuje konieczność nazwania zewnętrznej tabeli
  - możliwość błędu w określeniu tabeli źródłowej

Zagnieżdżenie: brak korelacji c.d.

- Podaj dane klientów, których nazwiska się powtarzają:

```
SELECT imie, nazwisko, miasto
FROM klient
WHERE nazwisko IN (
    SELECT nazwisko
    FROM klient
    GROUP BY nazwisko HAVING count (nazwisko) > 1
    )
```

  - *brak* korelacji, w zapytaniu podrzędnym nie ma odwołania do wiersza z tabeli zewnętrznej
  - mimo, że ta sama tabela przeglądana jest dwukrotnie, brak korelacji powoduje brak potrzeby zmiany nazwy
  - wiadomo w każdym miejscu o czyje nazwisko chodzi

Zagnieżdżenie w atrybucie wynikowym: korelacja III

- Podaj numery towarów wraz z ich całkowitymi wielkościami zamówień:

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

- Inne rozwiązanie:

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

  - wyświetlone są wszystkie towary, nawet te niezamawiane
  - zagnieżdżona tabela 1x1 użyta jako pojedyncza wartość
  - wewnętrzny **SELECT** odwołuje się do tabeli zewnętrznej
  - nawet jeśli wskazanie nie wystąpi bo nie jest konieczne
  - *występuje* korelacja

Zagnieżdżenie jako wartość: korelacja raz jeszcze

- Podaj dane o zamówieniach składanych przez klientów z Gdańska

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

  - klient_nr jest kluczem obcym w tabeli zamówień, jest więc dokładnie jeden klient dla tego zamówienia, wynikiem instrukcji **SELECT** jest tabela 1x1, czyli pojedyncza wartość
  - *występuje* korelacja
  - klient_nr jest jednoznaczną nazwą, można nawet nie wspomnieć, że pochodzi z tabeli zamówień

Negatywne zapytanie

- Podaj nazwiska klientów, którzy złożyli zamówienie po 1 lutego 2025:

```
SELECT DISTINCT nazwisko
FROM klient K, zamowienie
WHERE K.nr = klient_nr AND data_zlozenia > '2025-2-1'
```

- Podaj nazwiska klientów, którzy nie złożyli zamówienia po 1 lutego 2025 ?
  - nie wiadomo, któremu warunkowi zaprzeczyć

```
data_zlozenia <= '2025-2-1'
```

  - oznacza zamówienie złożone wcześniej, ale jednak złożone
  - klient mógł złożyć zamówienia i przed i po podanej dacie

```
K.nr != klient_nr
```

  - jest totalnym nieporozumieniem, wyświetla klientów z cudzymi zamówieniami

Negatywne zapytanie 2

- Podaj nazwiska klientów, którzy nie złożyli zamówienia po 1 lutego 2025:

```
SELECT nazwisko FROM klient
WHERE nr NOT IN ( SELECT klient_nr FROM zamowienie
                  WHERE data_zlozenia > '2025-2-1' )
```

  - albo

```
SELECT nazwisko FROM klient K
WHERE NOT EXISTS (
   SELECT * FROM zamowienie
   WHERE K.nr = klient_nr AND data_zlozenia > '2025-2-1' )
```


Instrukcja SELECT – operacje teoriomnogościowe

- Podaj nazwiska klientów, którzy nie złożyli zamówienia po 1 lutego 2025

```
SELECT nazwisko FROM klient
EXCEPT
SELECT nazwisko FROM klient
WHERE nr IN ( SELECT klient_nr FROM zamowienie
              WHERE data_zlozenia > '2025-2-1' )
```

  - operacja różnicy relacji
  - w tym przypadku rozwiązanie jest *nieprawidłowe*
  - może być dwóch klientów o tym samym nazwisku, jeden złożył zamówienie w badanym okresie, a drugi nie złożył
  - byłoby inaczej, gdyby wyświetlać nr klienta (wartość klucza)

Instrukcja SELECT – operacje teoriomnogościowe, c.d.

- Podaj dane klientów z Gdańska, którzy złożyli zamówienie

```
SELECT k.nr,imie,nazwisko FROM klient k JOIN
  zamowienie ON klient_nr=k.nr
INTERSECT
SELECT nr,imie,nazwisko FROM klient WHERE
  miasto='Gdańsk';
```

  - operacja przekroju relacji
  - i znowu bez klucza głównego rozwiązanie jest nieprawidłowe, jeden klient może być z Gdańska, inny złożyć zamówienie, imię i nazwisko te same

Instrukcja SELECT – operacje teoriomnogościowe, c.d.

- Podaj informacje o towarach, których w magazynie jest mało lub wcale

```
SELECT t.* FROM towar t JOIN zapas ON nr = towar_nr
WHERE ilosc < 10
UNION
SELECT * FROM towar
WHERE nr NOT IN ( SELECT towar_nr FROM zapas )
```

  - operacja sumy teoriomnogościowej relacji
  - w wersji z **UNION ALL** powtórzenia krotek są zachowane

Instrukcja INSERT – składnia

- **INSERT INTO *cel* [(*lista_elementów*)] *źródło;***
  - ***cel*** jest nazwą tabeli, do której wstawiamy dane
  - ***lista_elementów*** zawiera listę nazw atrybutów, którym chcemy nadać wartość, może być mniejsza niż pełna lista atrybutów tabeli ***cel***
  - ***źródło*** ma jedną z dwu postaci

    VALUES (*lista_wartości*)

    albo tabela otrzymana w wyniku operacji **SELECT**
- Od pewnej wersji PostgreSQL dopuszcza wygodniejszą formę wstawiania wielu wierszy:

  VALUES (*lista_wartości*) [,(*lista_wartości*)]*

  tzn. wymienienie wielu wierszy pod jednym słowem **VALUES**

Instrukcja INSERT – 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
- atrybuty „nr” oraz „cena” nie zostały wymienione
- będą miały wartość domyślną (kolejny numer / NULL)
- kolejność atrybutów nie musi być zgodna z kolejnością deklaracji w tabeli

Instrukcja INSERT – przykład, c.d.

- **INSERT INTO chwilowa**
  SELECT imie, nazwisko, ulica_dom
  FROM klient
  WHERE miasto = 'Gdańsk'
  - wstawia do utworzonej wcześniej tabeli 'chwilowa' całą tabelę otrzymaną w wyniku obliczenia operacji **SELECT**
- Postgres: celowe może być zdefiniowanie tabeli jako

  CREATE TEMP TABLE chwilowa (imię
  varchar(11), ...
  - taka tabela jest usuwana po zakończeniu sesji
- **INSERT INTO towar (opis, koszt, cena)**
  VALUES ('ramka do fotografii 3''x4''', 13.36, NULL)
  - podwójny apostrof służy do wprowadzenia znaku apostrofu
  - można wprowadzić w jawny sposób wartość nieokreśloną

Instrukcja UPDATE – składnia

- **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

Instrukcja UPDATE – 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

Instrukcja 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*

Instrukcja DELETE – przykład

- Usuń dane o klientach z Gdańska

```
DELETE FROM klient
    WHERE miasto = ‘Gdańsk’
```

- Usuń dane o zamówieniach składanych przez klientów z Gdańska

```
DELETE FROM zamowienie Z
WHERE EXISTS ( SELECT *
      FROM klient K
      WHERE K.nr = Z.klient_nr AND miasto = 'Gdańsk' )
```

  - klient_nr jest kluczem obcym w tabeli zamówień, jest jeden klient dla tego zamówienia, z Gdańska lub nie