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

- **INSERT** – realizuje aktualizację/wstawianie danych
- **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**
- **UPDATE** – realizuje aktualizację/zmianę wartości danych
- **DELETE** – realizuje aktualizację/usuwanie danych

Instrukcja SELECT

- Podaj nazwiska klientów, którzy złożyli zamówienie po 1 marca 2024:

```
SELECT DISTINCT nazwisko
FROM klient K, zamowienie
WHERE K.nr = klient_nr AND data_zlozenia > '2024-3-1'
```

  - 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

- Właściwe rozwiązanie:

```
SELECT nazwisko FROM klient
WHERE nr IN ( SELECT klient_nr
                FROM zamowienie
                WHERE data_zlozenia > '2024-3-1'
              )
```

  - 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

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
  - tabela ta musi być nazwana i wówczas może być użyta jako źródło dla kolejnego wyszukiwania

Zagnieżdżenie: korelacja

- Podaj dane klientów, którzy złożyli zamówienie po 1 marca 2024

```
SELECT nazwisko
FROM klient
WHERE EXISTS (
   SELECT *
   FROM zamowienie
   WHERE klient.nr = klient_nr AND data_zlozenia >
  '2024-3-1'
)
```

- 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 zamówienia po 1 marca 2024

```
SELECT nazwisko FROM klient
WHERE nr IN ( SELECT klient_nr
                FROM zamowienie
                WHERE data_zlozenia > '2024-3-1'
              )
```

  - 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*

Zagnieżdżenia: 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

Zagnieżdżenia: 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
  - *występuje* korelacja

Negatywne zapytanie

- Podaj nazwiska klientów, którzy złożyli zamówienie po 1 marca 2024:

```
SELECT DISTINCT nazwisko
FROM klient K, zamowienie
WHERE K.nr = klient_nr AND data_zlozenia > '2024-3-1'
```

- Podaj nazwiska klientów, którzy nie złożyli zamówienia po 1 marca 2024 ?
  - nie wiadomo, któremu warunkowi zaprzeczyć

```
data_zlozenia <= '2024-3-1'
```

  - 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 marca 2024:

```
SELECT nazwisko FROM klient
WHERE nr NOT IN ( SELECT klient_nr FROM zamowienie
                  WHERE data_zlozenia > '2024-3-1' )
```

  - albo

```
SELECT nazwisko FROM klient K
WHERE NOT EXISTS (
   SELECT * FROM zamowienie
   WHERE K.nr = klient_nr AND data_zlozenia > '2024-
  3-1' )
```

- Będą jeszcze inne rozwiązania tego problemu

Instrukcja SELECT – operacje algebry relacji

- Podaj nazwiska klientów, którzy nie złożyli zamówienia po 1 marca 2024 ?

```
SELECT nazwisko FROM klient
EXCEPT
SELECT nazwisko FROM klient
WHERE nr IN ( SELECT klient_nr FROM zamowienie
                  WHERE data_zlozenia > '2024-3-1' )
```

  - 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)
- Istnieją też **UNION** oraz **INTERSECT**
  - w wersji z **UNION ALL** powtórzenia krotek są zachowane

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 wszelkie informacje o zamówieniach składanych przez klientów z Gdańska

```
DELETE 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

Perspektywy

- Bardziej skomplikowane zapytanie może zostać zapamiętane

```
CREATE VIEW towar_zysk AS
    SELECT *, cena – koszt AS zysk FROM towar
```

  - zapamiętuje pytanie
  - późniejsze użycie odnosi się do treści tabeli z momentu tego użycia

```
SELECT * FROM towar_zysk
```

  - jest zawsze równoważne zapytaniu

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

Perspektywy a tabele tymczasowe

- Perspektywa jest czym innym niż tabela tymczasowa

```
CREATE TEMP TABLE towar_zysk (
    nr      int PRIMARY KEY,
    opis    varchar(64)  ,  koszt   numeric(7,2) ,
    cena    numeric(7,2) ,  zysk    numeric(7,2)  )
INSERT INTO towar_zysk
    SELECT *, cena – koszt AS zysk FROM towar
```

  - wstawia do utworzonej wcześniej tabeli wynik obliczenia operacji SELECT
  - po zmianie zawartości tabeli towarów pytania

```
SELECT *, cena – koszt AS zysk FROM towar
SELECT * FROM towar_zysk
```

  - dadzą *różne* odpowiedzi, nową i starą wartość zysku

Perspektywy c.d.

- Tabela tymczasowa może być używana tak jak każda tabela, w szczególności można do niej wstawiać i z niej usuwać krotki
- Operacje UPDATE i DELETE dla perspektyw nie są oczywiste

```
DELETE from towar_zysk
WHERE zysk/koszt<0.05
```

  - można sobie wyobrazić realizację powyższego polecenia jako

```
DELETE from towar
WHERE (cena-koszt)/koszt<0.05
```

  - ale jak miałaby działać poniższa operacja na tabeli towar?

```
UPDATE towar_zysk
SET zysk=zysk*1.1
```


Perspektywy 3.

- PostgreSQL do wersji 9.2 nie przewidywał operacji UPDATE i DELETE dla perspektyw

```
amb=> create view test as select * from towar;
CREATE VIEW
amb=> delete from test where nr=6;
ERROR:  cannot delete from view "test"
PODPOWIEDŹ:  You need an unconditional ON DELETE DO
INSTEAD rule or an INSTEAD OF DELETE trigger.
```


Perspektywy 4.

- Od Postgres wersji 9.3 dla szczególnie prostych perspektyw, definiowanych w oparciu o pojedynczą tabelę, bez grupowania, można używać operacji INSERT, DELETE i UPDATE
  - od wersji 9.4 perspektywa można dopuszczać pewne kolumny bez możliwości aktualizacji podczas gdy inne z możliwością
  - inne systemy zarządzania bazami danych mają/mogą mieć pewne możliwości operowania na perspektywach