

Relacyjne Bazy Danych

Andrzej M. Borzyszkowski
PJATK/ Gdańsk

materiały dostępne elektronicznie
<http://szuflandia.pjwstk.edu.pl/~amb>

Architektura SZBD

Trzy poziomy architektury

1. wewnętrzny
 - fizyczne przechowywanie danych
 - typy rekordów, indeksy, reprezentacja pól, kolejność przechowywania
2. pojęciowy (konceptyjny)
 - reprezentacja całej zawartości informacyjnej bazy
 - również reguły spójności
3. zewnętrzny
 - perspektywa konkretnego użytkownika
 - typy, pola, rekordy widziane przez pewnego użytkownika mogą być różne dla różnych użytkowników

Relacyjne Bazy Danych © Andrzej M. Borzyszkowski

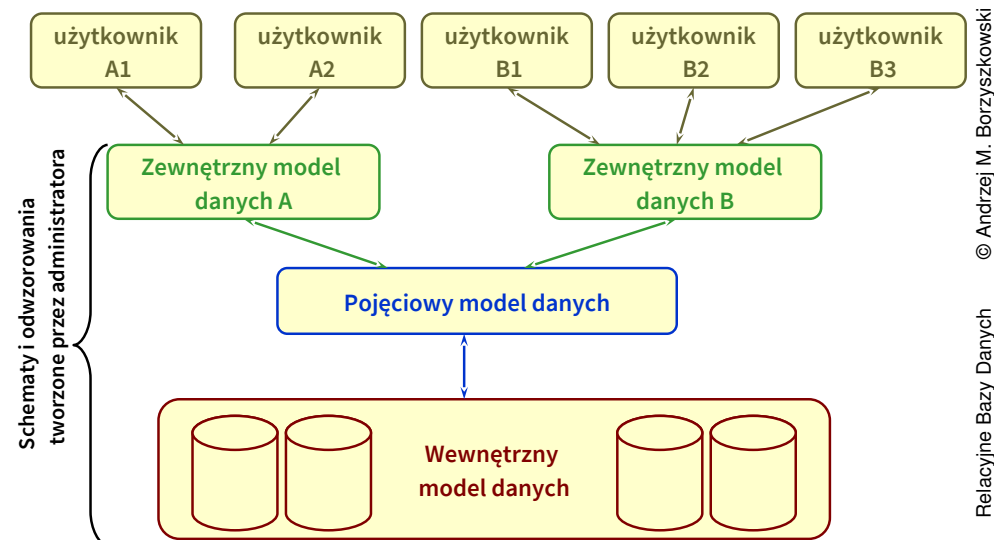
3

Architektura systemów zarządzania bazą danych

Relacyjne Bazy Danych © Andrzej M. Borzyszkowski

2

Architektura SZBD – schemat

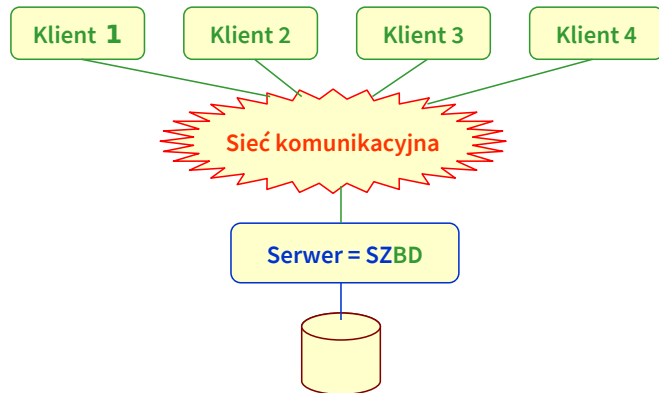


Relacyjne Bazy Danych © Andrzej M. Borzyszkowski

4

Architektura SZBD – klient-serwer

- Serwer jest systemem zarządzania bazą danych
- Klientami są aplikacje poziomu zewnętrznego



© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

5

Programowanie po stronie serwera

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

7

Konieczność rozszerzenia języka zapytań

- Rozszerzenia możliwości standardowych zapytań
 - zależą od konkretnej implementacji SZBD
 - większość (wszystkie?) implementacje mają jakieś rozszerzenia
 - PostgreSQL też ma, nawet dużo możliwości
- Czego SQL nie zapewni:
**CREATE TABLE BoM (id int PRIMARY KEY,
name varchar(22), part_of int REFERENCES BoM(id))**
 - jest tabelą z *rekursywnym* kluczem obcym**SELECT Sub.id
FROM BoM M INNER JOIN BoM Sub ON M.id=Sub.part_of
WHERE M.id=1**
 - wyświetla numery podzespółów, ale tylko bezpośrednich

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

8

Rekursja

- W standardowym SQL *nie ma* możliwości zażądania wyświetlenia wszystkich podzespółów na nieograniczonej głębokości
 - uwaga: standard SQL3 przewiduje taką możliwość
- W dowolnym języku programowania bez problemu można napisać pętlę (lub procedurę rekursywną) przeglądającą drzewo na całej głębokości
**wyświetl(int mat){
 print mat;
 for Sub in BoM if Sub.part_of=mat wyświetl(Sub.id);
}**
wyświetl(1);

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

9

Rekursja w Postgresie

```
WITH RECURSIVE Sub(id, name, part_of) AS (  
  SELECT id, name, part_of FROM BoM WHERE id = 1  
  UNION ALL  
    SELECT M.id, M.name, M.part_of  
    FROM Sub, BoM M  
    WHERE Sub.id = M.part_of  
)  
SELECT id, name  
FROM Sub
```

- tworzy na bieżąco (wirtualnie) tabelę podzespołów zespołu o podanym numerze i wyświetla wszystkie te podzespoły

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

10

Operatory

```
SELECT * FROM towar WHERE (cena*100)%100=99  
SELECT * FROM towar WHERE opis ~'^[IL].*x'
```

- operatory arytmetyczne (+ * / % ^ @ | /), logiczne, napisowe (||), binarne (>> << & | #)
- relacje arytmetyczne, napisowe (~ ~*)
- **ISNULL, LIKE,**
- operatory dotyczące czasu, adresów IP, ...
- można sprawdzić w powłoce psql
 - \do, \df

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

12

Operatory/funkcje/procedury

- Powód rozszerzeń
 - niewystarczalność SQL
 - wydajność, wygoda, etc.
- Rodzaje rozszerzeń
 - operatory
 - funkcje
 - procedury uruchamiane podczas startu bazy danych
 - procedury wyzwalane
- Możliwe języki programowania
 - SQL
 - PL/pgSQL
 - C
 - PL/Tcl, PL/Perl, PL/Python, i wiele innych

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

11

Funkcje

- Operują na liczbach, napisach, datach, adresach IP,
- Wiele funkcji ma wspólną nazwę, ale działa na innych typach
 - czasami działają tak samo, użytkownik nie zauważa typowania
 - czasami są to zupełnie różne funkcje
 - dodawanie liczb vs. konkatencja napisów
 - dzielenie liczb rzeczywistych vs. dzielenie liczb całkowitych
- Przykłady funkcji wbudowanych

matematyczne: log(x) pi() random()

napisowe: char_length(s) lower(s) trim(trailing ' ' from s)

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

13

Funkcje, c.d.

- Pożyteczne funkcje wbudowane:
 - ascii(s), chr(n) – zamiana liter i liczb wg kodu ASCII
 - ltrim(s), rtrim(s), btrim(s) – obcina spacje w końcach napisu
 - lpad(s,n), rpad(s,n) – wypełnia spacjami na końcu napisu
 - char_length(s), bit_length(x) – długości
 - lower(s), upper(s), initcap(s) – zamiana wielkości liter
 - substr(s,n,len), position(s1 IN s2) – podnapisy
 - translate (s, wzorzec, zamiennik) – zamiana liter
- date_part('jednostka',czas)
 - year, month, day, hour, minute, second
 - dow (dzień tygodnia), doy (dzień w roku), week
 - epoch (sekundy od 1.I.1970)
- Zamiana typu: to_char, to_date, to_number, to_timestamp

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

14

Definiowanie funkcji c.d.

- Język
 - musi być znany postgresowi, tzn. musi być uruchomiony do działania
 - od wersji 10 domyślnie uruchamiany jest język PL/pgSQL
 - CREATE EXTENSION SQL**
 - SELECT * FROM pg_language**
 - DROP language 'plpgsql'**
 - w bazie danych przechowywany jest kod funkcji, kompilacja nastąpi przy pierwszym wywołaniu
 - wniosek: dopiero wówczas ujawnią się błędy

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

16

Definiowanie własnych funkcji

```
CREATE FUNCTION nazwa ([typ [...]])  
RETURNS typ_wyniku  
AS definicja funkcji w jakimś języku  
LANGUAGE nazwa_języka
```

```
CREATE FUNCTION plus_raz(int4)  
RETURNS int4  
AS '  
BEGIN  
RETURN $1+$2; -- można też dać nazwę dla argumentu  
-- ALIAS liczba FOR $1  
END  
' LANGUAGE 'plpgsql'
```

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

15

Funkcje c.d.

- Sprawdzanie funkcji
SELECT prosrc FROM pg_proc WHERE proname='plus_raz';
- Usuwanie funkcji
DROP FUNCTION plus_raz(int4)
 - być może są inne funkcje plus_raz; nie zostaną one usunięte
 - “prawdziwa” nazwa funkcji zawiera jej typ
- Apostrof
 - może być potrzebny w definicji funkcji, wówczas podwójny
 - albo definicję objąć \$\$

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

17

Definiowanie funkcji c.d.

- Można wprowadzać nazwy dla parametrów formalnych (wcześniejsze wersje Postgresa nie pozwalały na to)
- Zamiast CREATE można użyć REPLACE albo CREATE OR REPLACE
 - ale nie zmieni się w ten sposób typów argumentu/ wyniku
- Parametr może być zadeklarowany jako
 - wejściowy IN (wartość)
 - wyjściowy OUT (zapis), INOUT
 - jeśli występują parametry wyjściowe, to można zrezygnować z RETURNS

```
CREATE FUNCTION pisz(IN int, OUT int, OUT text) AS $$  
    SELECT $1, CAST($1 AS text)||' jest też tekstem'  
    $$ LANGUAGE 'SQL'  
SELECT pisz(44)  
SELECT * FROM pisz(44)
```

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

18

Zmienne wierszowe

```
DECLARE nowy_k, stary_k klient%ROWTYPE;  
BEGIN  
    nowy_k.miesto := 'Gdynia';  
    nowy_k.ulica_dom := 'Tatrzańska 2';  
    nowy_k.kod_pocztowy := '81-111';  
    SELECT * INTO stary_k FROM klient WHERE nazwisko='Miszke';  
    IF NOT FOUND THEN -----  
    END IF;  
END  
  
– SELECT powinien zwrócić najwyżej jeden wiersz, dalsze  
  zostaną zignorowane  
– chyba, że użyto SELECT * INTO STRICT _____, wówczas błąd  
– istnieją sterowania FOR, LOOP, CONDITIONAL, RETURN  
  (obowiązkowy), RAISE
```

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

20

Język PL/pgSQL

- Program składa się z bloków, każdy ma swe lokalne deklaracje
DECLARE deklaracje BEGIN instrukcje END
 - komentarze identyczne jak w SQL -- /* */
 - zakres deklaracji zmiennych oczywisty
 - zmienna może być inicjalizowana
 - zmienna może być zadeklarowana jako stała (constant), wówczas musi być inicjalizowana
 - typ zmiennej może odwołać się do innego typu
 - jeden integer :=1;
 - pi constant float8 := pi();
 - mójopis towar.opis%TYPE := 'jakiś tekst'
 - nowy_klient klient%ROWTYPE;
 - wiersz record (typ ujawni się w momencie użycia)

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

19

Sterowanie

- Instrukcje warunkowe
 - IF () THEN ____
ELSEIF () THEN ____
ELSE ____
END IF
 - Wyrażenia warunkowe
 - NULLIF (wejście, wartość)
- zamienia określoną wartość na NULL
- CASE
WHEN ____ THEN ____
WHEN ____ THEN ____
ELSE ____
END

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

21

Pętle

- Pętle

```
LOOP n:=n+1; EXIT już WHEN n>1000; END LOOP;  
WHILE n<=1000 LOOP n:=n+1 END LOOP;  
FOR i IN 1..1000 LOOP _____ END LOOP;  
FOR wiersz IN SELECT ____ LOOP ____ END LOOP;  
EXIT
```

- albo warunkowo: EXIT WHEN (coś się stało)
- EXIT z_miejsca, opuszcza nie tylko bieżącą pętlę, ale i pętle wyżej położone, aż do etykiety z_miejsca

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

22

Nazwy dynamiczne

- EXECUTE "UPDATE" ||quote_ident (tu zmienna) || "SET"...
pozwała ułożyć zapytanie z elementów nieznanych w momencie pisania programu

nazwy tabel czy kolumn mogą zależeć od innych wartości

- Jest to bardzo nieefektywne, PostgreSQL nie może optymalizować zapytania przed wykonaniem
- FOR wiersz in EXECUTE "SELECT" _____
LOOP _____ END LOOP;

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

24

Wynik działania procedury

- RETURN, normalne zakończenie działania

- oznacza koniec obliczeń, nawet przed końcem bloku
- musi wystąpić, brak RETURN jest błędem
- RETURN NEXT nie kończy obliczeń, dodaje tylko kolejny wynik gdy spodziewamy się wyniku SETOF typ

- Wyjątki/komunikaty

RAISE DEBUG, zapisuje komunikat do pliku logów

RAISE NOTICE, wyświetla komunikat na ekran

RAISE EXCEPTION, j.w. + przerywa działanie procedury

RAISE NOTICE "wartość = %", zmienna

- po zdefiniowaniu w funkcji i wywołaniu wyświetli na ekranie komunikat o wartości zmiennej
- można podstawiać za % wyłączenie napisy

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

23

Kursory

- Nazwa dla zbioru wierszy wynikowych

DECLARE kursor CURSOR FOR SELECT _____;

BEGIN OPEN kursor;

LOOP FETCH kursor INTO wiersz

EXIT WHEN NOT FOUND;

PERFORM _____;

END LOOP;

CLOSE kursor;

- wiersz musi być odpowiedniego typu, albo RECORD, albo %ROWTYPE, albo ciąg pojedynczych zmiennych dla każdego atrybutu

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

25

Kursory, c.d.

- Kursor może mieć parametr

```
DECLARE kursor CURSOR (parametr typu) FOR SELECT _____  
parametr ____;
```

```
BEGIN OPEN kursor (parametr);
```

- parametr może być wartością, np. użytą w WHERE
- ale nie może być np. nazwą tabeli

- Można zdefiniować wskaźnik na kursor

```
DECLARE ten_kursor REFCURSOR; -- i inne kursory też
```

```
BEGIN OPEN kursor; -- otwiera konkretny kursor
```

```
ten_kursor := kursor;
```

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

26

Definiowanie operatorów

- W zasadzie operatory są funkcjami
 - tyle, że wygodna składnia do wywołania
- Definicja operatora wymaga definicji funkcji, która ma policzyć wartość operatora

```
CREATE OPERATOR + (  
  leftarg = <typ_lewego>,  
  rightarg = <typ_prawego>,  
  procedure = <nazwa funkcji, która policzy wynik>  
);  
-- jeśli operator jest unarny, to należy opuścić jeden  
-- z argumentów
```

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

28

SQL też jest językiem

- Nie ma zmiennych ani sterowania (pętli, warunkowych)
- Nie ma RETURN (zwracane są dane z *ostatniego* SELECT-a wewnątrz definicji)
 - ale można zadeklarować typ wyjściowy jako void, wówczas stosuje się polecenia SQL INSERT czy UPDATE
- Są parametry, parametry aktualne zastępują \$1, itd z definicji
CREATE FUNCTION przykład (text)
RETURNS SETOF klient AS'
SELECT * FROM klient WHERE miasto=\$1
' LANGUAGE SQL
- Typem danych wejściowych może być nazwa tabeli (tzn. na wejściu znajdzie się wiersz z tej tabeli)
 - również dane wyjściowe mogą utworzyć wiersz takiego typu

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

27

Procedury wyzwalane

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

29

Procedury wyzwalane

- Nazwa: *trigger*, trygier, wyzwalacz
- Jak?
 - procedury są wyzwalane “automatycznie” przez zdarzenia w bazie danych
- Dlaczego?
 - poprawność danych (pojedynczych, zależnych od innych)
 - śledzenie zmian, audyt, raport, zapis zmian
 - naruszenie postaci normalnej, kopie danych, dane wynikowe
 - dane bieżące vs. archiwalne
 - spowodowane ergonomią, wydajnością, specjalny format danych dla innych aplikacji

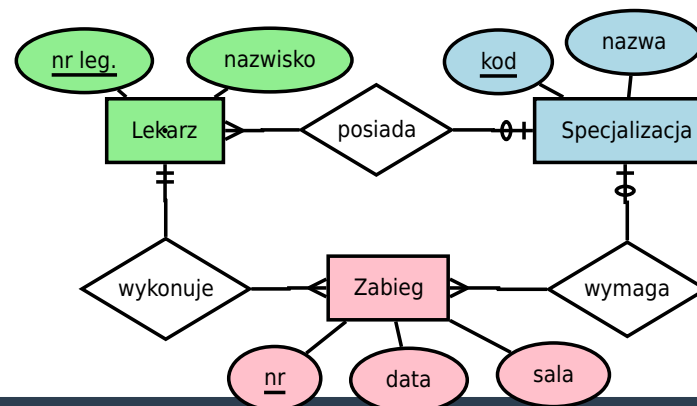
© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

30

Przykłady „dlaczego” – poprawność

- Przykłady pętli w diagramach związków i encji w projektach:
 - lekarz <wykonuje> zabieg
- lekarz <ma> specjalizację
- zabieg <wymaga> specjalizacji



© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

31

Przykłady „dlaczego” – poprawność c.d.

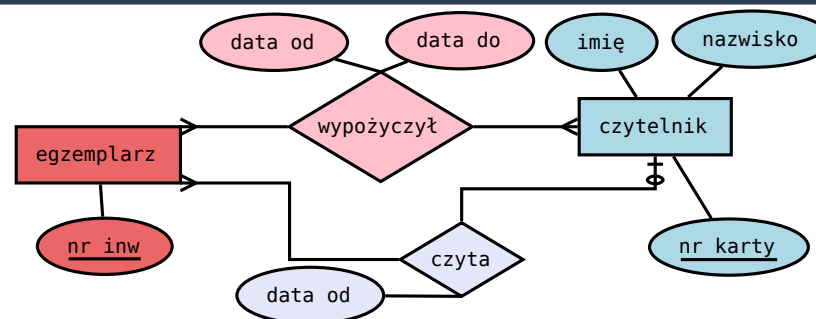
- Przykłady pętli c.d.:
 - nauczyciel <uczy> klasa, przedmiot (związek 3 encji)
- nauczyciel <jest wychowawcą> klasa
 - tylko jeśli uczy jakiegoś przedmiotu w klasie
- klub <jest gościem> w meczu
- klub <jest gospodarzem> w meczu
 - ale nie może być jednym i drugim
- Formalna poprawność danych
 - pesel ma 11 cyfr

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

32

Przykłady „dlaczego” – archiwizacja



© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

33

- W projekcie biblioteki używa się związku <czyta>
 - zapisując czas automatycznie
 - po zwrocie książki zapisywane jest wypożyczenie
 - po zmianie ceny towaru zapisywana jest odrębnie dawna cena
 - po usunięciu faktury dane są archiwizowane

Przykłady „dlaczego” – dane wynikowe, kopie

- Dane o towarach wprowadzane w centrali firmy
 - są automatycznie kopiowane w oddziałach
- Zamówienie złożone przez klienta
 - zmienia atrybut „łącznie suma zamówień”
 - warto mieć pod ręką dane zbiorcze by łatwiej obliczyć rabat dla kolejnego zamówienia
 - gol w meczu zmienia wynik meczu, zmienia pozycję klubu w lidze, zmienia statystyki i ranking piłkarza

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

34

Przykład: odnowienie zapasów

```
CREATE TRIGGER uzupełnij_trig
AFTER INSERT OR UPDATE ON zapas
FOR EACH ROW EXECUTE PROCEDURE uzupełnij_trig_proc(13);
```

- jeśli zostanie dokonana zmiana w tabeli zapasów, to zostanie wywołana procedura uzupełnij_trig_proc, która bada, czy zapasy nie są zbyt małe i być może trzeba złożyć zamówienie (do specjalnej tabeli nowych zamówień)

```
CREATE FUNCTION uzupełnij_trig_proc()
RETURNS TRIGGER AS $$
DECLARE
```

```
    prog INTEGER;
```

```
wiersz RECORD; ciąg dalszy na następnym slajdzie
```

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

36

Definiowanie procedur wyzwalanych

- **CREATE TRIGGER** nazwa BEFORE|AFTER
INSERT|DELETE|UPDATE ON nazwa_tablicy
FOR EACH ROW|STATEMENT
EXECUTE PROCEDURE nazwa_funkcji(arg)
- procedura używana w definicji wyzwalacza musi być wcześniej zdefiniowana
 - typ wynikowy musi być TRIGGER
 - zwraca albo NULL, albo wiersz pasujący do typu tabeli występującej w wyzwalaczu
 - formalnie nie ma argumentów, naprawdę argumenty odczytuje z tablicy tg_argv[] o wielkości tg_nargs
 - może odwoływać się do **new** i **old**, nowa i stara wartość zmienianego wiersza (dla wyzwalaczy FOR EACH ROW)

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

35

Przykład, c.d.

```
BEGIN
    prog := tg_argv[0];
    RAISE NOTICE "próg wynosi %", prog;
    IF new.ilosc < prog
    THEN
        SELECT * INTO wiersz FROM towar
        WHERE nr = new.towar_nr;
        INSERT INTO nowe_zamowienie
        VALUES (wiersz.nr, wiersz.opis, prog-new.ilosc, now());
        RAISE NOTICE 'trzeba zamówić towar: %', wiersz.opis;
    END IF;
    RETURN NULL;
END; $$ LANGUAGE plpgsql
```

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

37

Reguły

- Mogą mieć podobne skutki jak wyzwalacze

```
CREATE RULE name AS ON [UPDATE | INSERT | DELETE]
  TO table [ WHERE condition ]
  DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ;
  command ... ) }
```

 - np.: archiwizacja danych

```
CREATE RULE archiwizuj AS ON UPDATE
  TO towar WHERE old.cena<>new.cena
  DO INSERT INTO towar_log
    VALUES (old.nr, old.cena, now())
```
 - gdzie tabela towar_log musiała być przedtem odpowiednio zdefiniowana
 - ALSO jest domyślne, INSTEAD musi być jawnie wyrażone

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

38

Reguły dla perspektyw

- Reguły są jedyną możliwością aktualizacji perspektyw
 - założmy, że mamy zadeklarowaną perspektywę

```
CREATE VIEW towar_zysk AS
  SELECT *, cena - koszt AS zysk FROM towar
```
 - w niektórych implementacjach nie ma możliwości usuwania wierszy z perspektywy
 - ale można zdefiniować regułę

```
CREATE RULE towar_zysk_del AS
  ON DELETE TO towar_zysk
  DO INSTEAD DELETE FROM towar WHERE nr=old.nr
```
 - wówczas polecenie usuwania z perspektywy usunie odpowiadający wiersz w tabeli

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

39

Reguły vs. wyzwalacze

- Reguła powoduje (może spowodować) wykonanie kolejnego polecenia SQL
 - polecenie dotyczące wielu wierszy może być dobrze zoptymalizowane przez SZBD
 - reguła jest analizowana przed rzeczywistym wykonaniem i może prowadzić do błędu rekursywnego wywołania
- Wyzwalacze używają funkcji, a te mogą więcej niż polecenia SQL
 - np. integralność referencyjna wymaga sprawdzenia istnienia danych i ew. zgłoszenie błędu

© Andrzej M. Borzyszkowski

Relacyjne Bazy Danych

40