Style programowania aplikacji bazodanowych

Osadzony SQL — wstawki SQL w języku programowania — styl ten dość „rozwlekły”, ale uniwersalny, bo piszemy w standardowym SQL. Trzeba przy tym ograniczać się do zrozumiałych i dostępnych w SQL-owej bazie poleceń i konstrukcji.

API — biblioteki procedur realizujących odwołania do baz danych (JDBC, ODBC iin.) — jest to styl często preferowany dzięki bogactwu bibliotek i ich integracji z odpowiednim językiem programowania. Istotna (kłopotliwa?) w nim jest specyfika poszczególnych bibliotek i konieczność dobrego zrozumienia działania poszczególnych funkcji/mechanizmów biblioteki w bazie danych.

Programistyczny SQL — pisanie we własnych językach programowania SZBD, jak PL/SQL w Oracle, plpgsql w Postgresie itp. (nazywanych czasami językami czwartej generacji 4GL). Z tego stylu korzystamy często przy programowaniu funkcji, procedur czy wyzwalaczy w bazie danych. Nie daje on takich możliwości, jak wykorzystanie odpowiedniego języka programowania bogatego w potrzebne struktury, funkcje itp.

Włączanie SQL do programu

- Standard SQL daje możliwość umieszczania w treści programu napisanego w **języku macierzystym** (np. C, Pascal, ADA,...) instrukcji SQL wyróżnionych słowami `EXEC SQL`.
- Program ze wstawkami musi zostać poddany **prekompilacji** — np. ecpg prekompiluje program w C z instrukcjami SQL do programu C z wywołaniami procedur.
- Program po prekompilacji używa odwołań do procedur z **biblioteki** dla danego języka programowania — czyli jest praktycznie aplikacją w stylu API.

Komunikacja program ↔ SZBD

Używając wstawek SQL w programie musimy mieć możliwość:

- przekazania z programu do bazy danych treści polecenia;
- ewentualnie wstawienia do treści polecenia wartości zmiennych programowych (**zmienne dzielone**);
- ewentualnie skomponowania treści polecenia na poziomie programu (`PREPARE+EXECUTE` i `EXECUTE IMMEDIATE`);
- odebrania z SZBD informacji o sposobie wykonania instrukcji (**sqlca**, `SQLCODE`, `WHENEVER`);
- ewentualnie odebrania z SZBD wyników wygenerowanych przez instrukcję — mogą być to wyniki o rozmiarze przewidywalnym (jedna liczba czy jedna krotka, `SELECT INTO`) lub o dowolnym rozmiarze (**kursory**).

Proste polecenia SQL

Zacznijmy od prostych poleceń SQL, które nie generują wyników. Może to być polecenie połączenia z bazą, od którego i tak powinniśmy zacząć.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  EXEC SQL  CONNECT TO "zapisy" AS conn1
            USER "admin" IDENTIFIED BY "haslo_admina";
  if (sqlca.sqlcode) {
    printf("Błąd. Nie udało się połączyć z bazą (%s)",sqlca.sqlerrm.sq
    return 0;
  }
  //... możemy działać na bazie z domyślnym połączeniem conn1;
  //... możemy stworzyć inne połączenia i korzystać z wielu
  //    przełączając je;
  EXEC SQL DISCONNECT conn1;
  return 0;
}
end;
```

Błędy bazy danych

Po każdym poleceniu przesłanym do bazy otrzymujemy informację o sukcesie/porażce za pomocą specjalnych zmiennych i/lub struktury.

`SQLSTATE` — zmienna znakowa (6 znaków)

`SQLCODE` — liczba całkowita z kodem błędu lub ostrzeżenia;

`sqlca` (*SQL Communication Area*) — struktura zawierająca takie pola, jak `sqlcode`, `sqlerrm`... opisująca dokładnie błąd ostatniej operacji;

`WHENEVER` — operacja pozwala określić działanie podejmowane w przypadku wystąpienia konkretnego błędu lub zdarzenia:

```
EXEC SQL WHENEVER {SQLWARNING | SQLERROR | NOT FOUND}
         {CONTINUE | DO <procedura> | GOTO <etykieta> | STOP};
```

Deklaracja zmiennej

Zmienne dzielone to zmienne używane zarówno w programie, jak i w poleceniach SQL. Muszą być więc typu obsługiwanego przez oba systemy w sposób analogiczny.

- zmienne dzielone musimy zadeklarować w **sekcji deklaracji**
  ```
  EXEC SQL BEGIN DECLARE SECTION;
  ...
  EXEC SQL END DECLARE SECTION;
  ```
- dla zmiennych określamy typy z języka macierzystego;
- zmiennych używamy ”normalnie” w instrukcjach języka macierzystego;
- w treści poleceń SQL nazwy zmiennych dzielonych poprzedzamy znakiem **:**.

```
EXEC SQL BEGIN DECLARE SECTION;
int kodUz=123;
EXEC SQL END DECLARE SECTION;
EXEC SQL DELETE FROM uzytkownik WHERE kod_uz=:kodUz;
EXEC SQL INSERT INTO uzytkownik VALUES(:kodUz,'Anna','Abacka');
```

Uwaga: *W zapytaniach SQL zmiennych można używać w miejscu parametrów zapytania (wartości atrybutów), a nie w miejscu nazw tabel, kolumn czy poleceń.*

SELECT INTO

Zmienne dzielone mogą posłużyć także do odebrania wyników wyszukanych poleceniem SELECT w bazie danych (właściwie SELECT ... INTO). Możliwe jest to jednak tylko wówczas, gdy wynikiem jest jedna krotka.

```
EXEC SQL BEGIN DECLARE SECTION;
char *nazwaPrz="Bazy danych";
char *nazwaSem="Semestr letni 2009/2010";
int wynik; char wykladowca[30];
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT COUNT(*),nazwisko INTO :wynik,:wykladowca
FROM wybor NATURAL JOIN grupa NATURAL JOIN przedmiot_semestr
NATURAL JOIN przedmiot JOIN semestr USING (semestr_id) JOIN
uzytkownik ON grupa.kod_uz=uzytkownik.kod_uz
WHERE rodzaj_zajec='w' AND przedmiot.nazwa=:nazwaPrz AND
semestr.nazwa=:nazwaSem
GROUP BY nazwisko;
...
```

Indykatory

Wiadomo, że polecenia SQL potrafią zwracać **”wartości” NULL**. NULLe są rozpoznawane i obsługiwane w specjalny sposób przez SZBD, a języki programowania mogą nie odróżniać ich od innych wartości.

Z każdą zmienną dzieloną użytą w zapytaniu SQL możemy użyć **indykatora**. Jest to zmienna typu całkowitego definiowana w sekcji deklaracji, jak pozostałe zmienne dzielone. Użycie zmiennej dzielonej w parze z indykatorem w poleceniu SQL powoduje, że SZBD nadaje wartość indykatorowi:

- ujemny indykator oznacza, że zmienna dzielona jest NULL;
- indykator dodatni oznacza, że SZBD przypisał zmiennej dzielonej wartość, ale musiał ją obciąć z powodu niedopasowania typów;
- indykator równy zero oznacza, że przypisanie wypadło całkowicie pomyślnie.

SELECT INTO — przykład 2

W poniższym przykładzie użyjemy indykatorów, bo nie mamy gwarancji, że poszukiwana osoba istnieje w bazie. Mamy jednak gwarancję, że wynikiem jest jedna krotka, więc możemy zapisać ją do wskazanych zmiennych:

```
EXEC SQL BEGIN DECLARE SECTION;
char nazwisko[30]; char imie[30];
int kodUz=123, nInd, iInd;
EXEC SQL END DECLARE SECTION;

EXEC SQL SELECT nazwisko,imie INTO :nazwisko :nInd, :imie :iInd
FROM uzytkownik WHERE kod_uz=:kodUz;

if (NO_DATA_FOUND) printf("nie ma takiej osoby\n");
if (nInd<0) nazwisko="nazwisko nieznane";
if (iInd<0) imie="brak imienia";
printf("Osoba o kodzie %d to: %s %s \n",kodUz,imie,nazwisko);
```

Zastosowanie kursora

Kursor służy do przeglądania wyniku zapytania SELECT zwracającego relację zawierajcą dowolną liczbę krotek. Kursor wiążemy z pewnym zapytaniem (deklaracja), potem możemy zapytanie wykonać (także wielokrotnie) i po każdym wykonaniu przejrzeć wynik.

DECLARE — operacja wiąże ”obiekt” typu kursor z zapytaniem `SELECT`. Deklaracja nie powoduje (próby) wykonania zapytania.

OPEN — zapytanie zostaje wykonane, wyliczony zostaje wynik, a kursor jest przygotowany do pobrania jego pierwszej krotki.

FETCH — pobieramy krotkę wskazywaną przez kursor i przypisujemy jej pola pod zmienne dzielone wymienione w operacji `FETCH`. Po operacji kursor automatycznie jest przesuwany do kolejnej krotki wyniku.

MOVE — przesunięcie kursora bez pobierania krotki.

CLOSE — zamknięcie kursora kończy dostęp do wyniku zapytania; SZBD może zwolnić zasoby, które były potrzebne do wygenerowania wyniku i (ewentualnie) udostępnić bazę innym użytkownikom.

Przykład kursora

```
EXEC SQL BEGIN DECLARE SECTION;
  char nazwaPrz[30]="Bazy danych",
       nazwaSem[30]="Semestr letni 2009/2010",
       nazwisko[30], imie[30];
  int ind1, ind2;
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE prowadzacy CURSOR FOR  SELECT nazwisko, imie
  FROM uzytkownik NATURAL JOIN grupa NATURAL JOIN przedmiot_semestr
       NATURAL JOIN przedmiot JOIN semestr USING (kod_przed)
  WHERE przedmiot.nazwa=:nazwaPrz AND semestr.nazwa=:nazwaSem;
EXEC SQL OPEN prowadzacy;
while (1) {
  EXEC SQL FETCH prowadzacy INTO :nazwisko :ind1, :imie :ind2;
  if (NO_DATA_FOUND) break;
  if (ind1==0 || ind2==0) printf("%s %s\n",nazwisko,imie);
};

EXEC SQL CLOSE szukajProwadzących;
```

Kursor do modyfikacji

```
EXEC SQL BEGIN DECLARE SECTION;
  char nazwa[30];
  int nr,l;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE FOR UPDATE bledneSemestry CURSOR FOR
  SELECT * FROM SEMESTR WHERE nazwa LIKE '%błąd%';

EXEC SQL OPEN bledneSemestry;
do {
  EXEC SQL FETCH FROM bledneSemestry INTO :nr, :nazwa;
  if (NO_DATA_FOUND) break;
  EXEC SQL SELECT COUNT(*) INTO :l
           FROM przedmiot_semestr WHERE semestr_id=:nr;
  if (l==0)
  EXEC SQL DELETE FROM CURRENT OF bledneSemestry;
} while (!NO_DATA_FOUND);

EXEC SQL CLOSE bledneSemestry;
```

Obsługa kursorów

- Kursory do modyfikacji (`DECLARE FOR UPDATE ...CURSOR FOR ...`) muszą być zdefiniowane dla zapytania, którego krotka jest identyfikowalna z krotką z bazy danych. Mogą służyć do usuwania (`DELETE`) lub modyfikowania (`UPDATE`) danych.
- Kursor do modyfikacji powoduje, że użytkownik trzyma dłużej dane i blokuje dostęp do nich innym aplikacjom i użytkownikom.
- Kursor może być przewijalny (`DECLARE SCROLL ...CURSOR FOR ...`) — wówczas operacje `FETCH` i `MOVE` mogą przechodzić także w tył (`FETCH PRIOR`) czy w dowolne miejsce (`ABSOLUTE`, `RELATIVE`). Kursor przewijalny jest obsługiany zazwyczaj znacznie wolniej niż nieprzewijalny. Wymaga trzymania dużej ilości danych, nie może być wykonywany pewnymi metodami przetwarzającymi dane potokowo itp. Podobnie jak kursor do modyfikacji blokuje bardziej dostęp do danych innym aplikacjom niż kursor nieprzewijalny.

PREPARE i EXECUTE

Polecenia `PREPARE` i `EXECUTE` pozwalają utworzyć szablon zapytania (`PREPARE`) a następnie wykonać go, ewentualnie wstawiając w miejsce stałych wartości.

PREPARE Polecenie to otrzymuje słowo — treść zapytania SQL.

```
EXEC SQL PREPARE noweZapytanie
         FROM [ zmienna | literałZnakowy];
```

W treści zapytania (`[zmienna|literałZnakowy]`), w miejscu stałych, mogą wystąpić *substytuty parametrów* oznaczane znakiem zapytania. Polecenie powoduje przygotowanie **planu** wykonania zapytania i podstawienie odwołania do procedury pod zmienną `noweZapytanie`.

EXECUTE Polecenie otrzymuje jako argument odwołanie do procedury wygenerowane przez `PREPARE`.

```
EXEC SQL EXECUTE noweZapytanie
[USING wartość1, wartość2,...][INTO zm1, zm2,...];
```

Jeśli w zapytaniu występują parametry, to są za nie podstawiane podane wartości. Jeśli zapytanie zwraca wartości, to są one podstawiane pod zmienne.

PREPARE i EXECUTE — przykład

```
EXEC SQL BEGIN DECLARE SECTION;
int kodStud=123, zapisy;
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE zad1 FROM
     "SELECT COUNT(*) FROM wybor WHERE kod_uz=?";
EXEC SQL EXECUTE zad1 INTO :zapisy USING :kodStud;
EXEC SQL DEALLOCATE PREPARE name;
```

EXECUTE IMMEDIATE

Polecenie `EXECUTE IMMEDIATE` przyjmuje jeden argument — zmienną lub stałą zawierającą treść instrukcji do wykonania. Instrukcja ta nie może zwracać wartości i nie zawiera parametrów.

```
EXEC SQL BEGIN DECLARE SECTION;
const char *polecenie = "DROP TABLE uzytkownik CASCADE;";
EXEC SQL END DECLARE SECTION;

EXEC SQL EXECUTE IMMEDIATE :polecenie;
```

Programowanie w stylu API

```
#include <stdio.h>
#include <mysql.h>
int main() {
  MYSQL ms; MYSQL_RES *res; MYSQL_ROW row;

  if (mysql_init(&ms)==NULL)
     {fprintf(stderr,"%s\n",mysql_error(&ms); exit(1);}
  mysql_real_connect(&ms,"localhost","ja","haslo","zapisy",0,NULL,0);

  char *stmt="select * from uzytkownik where kod_uz in (......);"
  if (mysql_query(&ms,stmt))
     {fprintf(stderr,"%s\n",mysql_error(&ms)); exit(1);

  res=mysql_use_result(&ms);
  while ((row=mysql_fetch_result(res)!=NULL)
     printf("%s %s\n",row[1],row[2]);
  mysql_free_result(res);

  mysql_close(&ms);
  return 0;}
```

Funkcje i procedury SQL — prosty przykład

Język SQL pozwala pisać proste funkcje będące zlepkiem kilku poleceń sql bez żadnych dodatkowych poleceń sterujących. Funkcja może przyjmować argumenty, które mogą być użyte w treści w miejscu stałych.

```
CREATE FUNCTION nazwa(argumenty) [RETURNS typ_wyniku]
AS $X$ tresc_funkcji $X$ LANGUAGE sql;
```

```
CREATE FUNCTION usun_grupe(int) AS $X$
DELETE FROM wybor WHERE kod_grupy=$1;
DELETE FROM grupa WHERE kod_grupy=$1;
$X$ LANGUAGE sql;
```

Wywołanie funkcji powoduje wykonanie zawartych w niej poleceń:

```
SELECT usun_grupe(123);
SELECT usun_grupe(kod_grupy) FROM grupa WHERE kod_uz=234;
```

Funkcje i procedury SQL — zwracana wartość

Funkcja SQL może zwracać jedną wartość, rekord lub zbiór rekordów (relację). Wynikiem funkcji jest odpowiedź na ostatnie zapytanie zawarte w treści funkcji.

```
CREATE FUNCTION f1(int) RETURNS text AS $X$
SELECT nazwisko::text FROM uzytkownik WHERE kod_uz=$1;
$X$ LANGUAGE sql;

CREATE FUNCTION f2(uzytkownik.kod_uz%TYPE) RETURNS uzytkownik AS
'SELECT * FROM uzytkownik WHERE kod_uz=$1;'
LANGUAGE sql;

CREATE FUNCTION f3() RETURNS SETOF uzytkownik AS $X$
SELECT * FROM uzytkownik WHERE kod_uz IN (SELECT kod_uz FROM grupa);
$X$ LANGUAGE sql;
```

Funkcję zwracającą relację używamy w zapytaniach SELECT, jak „zwykłe” relacje bazy danych.

```
SELECT nazwisko FROM f3() ORDER BY 1;
```

Funkcje i procedury SQL — usuwanie funkcji

Funcję usuwamy (kasujemy) poleceniem DROP. W poleceniu trzeba podać nazwe funkcji i listę typów jej parametrów, bo dopiero to w pełni pozwala zidentyfikowac funkcję.

```
DROP FUNCTION f3();
DROP FUNCTION f1(INT);
```

Funkcje i procedury SQL — zwracana wartość nowego typu

Wartość zwracana przez funkcję może być rekordem nowego typu, niezgodnego z żadną z instniejących w bazie relacji. Wówczas typ trzeba najpierw zdefiniować.

```
CREATE TYPE grupa_z_liczba AS
(kod_grupy INT, osob INT, max_osob INT, kod_uz INT);

CREATE FUNCTION zapelnienie_grup_w_semestrze(semestr.id_semestr%TYPE)
RETURNS SETOF grupa_z_liczba AS $X$
  SELECT grupa.kod_grupy,count(*) AS "osob",max_osoby, grupa.kod_uz
  FROM grupa JOIN wybor USING(kod_grupy) JOIN
       przedmiot_semestr USING(kod_przed_sem)
  WHERE semestr_id=$1
  GROUP BY grupa.kod_grupy, max_osoby;
$X$ LANGUAGE sql;

SELECT nazwisko,kod_grupy
FROM zapelnienie_grup_w_semestrze(29) NATURAL JOIN uzytkownik
WHERE osob>max_osoby;
```

Struktura funkcji PL/pgSQL

```
CREATE FUNCTION <nazwa>([<arg>] <typ_argumentu>,...)
  [RETURNS [SETOF] <typ_wyniku>]
  AS <ogranicznik> <tresc_funkcji> <ogranicznik>
  LANGUAGE plpgsql;
```

Treść funkcji ma strukturę blokową — bloki mogą być w sobie zagnieżdżane i każdy może być poprzedzony etykietą i/lub sekcją deklaracji:

```
[<<etykieta>>]
  [DECLARE <zmienna> <typ_zmiennej>;,...]
  BEGIN <instrukcja>;,... END
```

Deklaracje

W sekcji deklaracji umieszczamy zmienne używane w bloku i w blokach w nim zagnieżdżonych. Jako typ zmiennej można podać typ bazodanowy, typ zdefiniowany (`CREATE TYPE`), typ zgodny z pewnym atrybutem lub krotką pewnej tabeli (`uzytkownik.kod_uz%TYPE`, `uzytkownik%TYPE`, `uzytkownik`). Możemy także nadać wartości poczatkowe i domyślne zmiennym i zastrzec, czy mogą przyjmować wartosci puste. W sekcji deklaracji można także nadać (zmienić) nazwy atrybutom. Przykłady widzimy poniżej:

```
DECLARE liczba int NOT NULL DEFAULT 100;
        moja_strona text:='http://www.ii.uni.wroc.pl';
        osoba uzytkownik%rowtype; -- typ zgodny z tablicą bazy danych
        arg1 ALIAS FOR $1;        -- przemianowanie argumentu
        wiersz RECORD;            -- zmienna może wskazywać na dowolny
```

Podstawowe instrukcje

Przypisanie:
```
<zmienna> := <wartość_lub_wyrażenie>;
```

Instr. war.:
```
IF <wyrazenie> THEN <instrukcje>
[ELSEIF <wyrazenie> THEN <instrukcje>]
[ ELSE <instrukcje>] END IF;
```

Przełącznik:
```
CASE [<wyrazenie>]
[WHEN <wartosci> THEN <instrukcje>]
END CASE;
```

W powyższym `<instrukcje>` to ciąg instrukcji oddzielonych średnikami a `<wartosci>` to ciąg stałych i/lub wyrażeń oddzielonych przecinkami.

Pętle

- `LOOP <instrukcje> END LOOP;`
- `WHILE <wyr_log> LOOP <instrukcje> END LOOP;`
- `FOR <zmienna> IN [REVERSE] <wyr1>..<wyr2> [BY <wyrazenie>] LOOP <instrukcje> END LOOP;`

Każdą pętlę można poprzedzić etykietą: `<<powrot>> LOOP ... END LOOP;`

Z pętli można wyjść instrukcjami `EXIT WHEN <wyrażenie>;` `CONTINUE WHEN <wyrażenie>;`.

Pętle cd.

Pętla po wynikach zapytania:

```
DECLARE osoba RECORD;
BEGIN
  FOR osoba IN SELECT uzytkownik.*
     FROM uzytkownik NATURAL JOIN grupa
     WHERE rodzaj='w' ORDER BY nazwisko
  LOOP
     UPDATE uzytkownik SET tytul='prof' WHERE kod_uz=osoba.kod_uz;
  END LOOP;
END;
```

RETURN

Instrukcja `RETURN` służy do generowania wyniku i/lub wyjścia z funkcji:

- `RETURN NEXT <wyrażenie>` oraz `RETURN QUERY <zapytanie>` dodają do wyniku funkcji krotkę lub kilka krotek;
- `RETURN <wartość>` powoduje wyjście i zwrócenie podanej wartości jako wartości funkcji;
- `RETURN` powoduje wyjście z funkcji i zwrócenie wyniku obliczonego przez kolejne instrukcje `RETURN NEXT` i `RETURN QUERY` lub wyniku `NULL`.

SQL

W treści funkcji możemy używać bezpośrednio zapytań SQL. Możliwe są następujące ograniczenia:

- instrukcje `COMMIT` i `ROLLBACK` sterują wykonaniem transakcji — zazwyczaj cała funkcja jest jedną transakacją i instrukcji tych nie można używać wewnątrz funkcji;
- zapytania zwracające jedną krotkę zadajemy w postaci `SELECT...INTO...`;
- zapytania zwracające wiele krotek wykonujemy łącznie z pętlą `FOR` przebiegającą wyniki zapytania, w powiązaniu z kursorem (`DECLARE ...CURSOR FOR...` lub poleceniem `RETURN QUERY`;
- zapytania nie zwracające wyniku nie wymagają żadnych dodatkowych konstrukcji.

Wyzwalacze w PostgreSQL

Składnia wyzwalacza w PostgreSQL jest analogiczna do składni standardowej w części nagłówkowej. Natomiast wszystkie instrukcje wyzwalacza są przeniesione do wnętrza procedury, która jest w wyzwalaczu jedynie wywoływana.

```
CREATE TRIGGER odnotuj_wypis AFTER DELETE ON wybor FOR EACH ROW
EXECUTE PROCEDURE odnotuj_wypis_proc();
```

Procedura wyzwalacza musi zwracać typ `TRIGGER` i może wewnątrz odwoływać się do zmiennych `OLD` i `NEW` oznaczających krotkę sprzed zmian (dla `DELETE` i `UPDATE`) oraz po zmianach (dla `INSERT` i `UPDATE`).

```
CREATE FUNCTION odnotuj_wypis() RETURNS TRIGGER AS $X$
BEGIN
INSERT INTO wypisy
       VALUES(OLD.kod_uz,OLD.kod_grupy,OLD.czas_zapisu,now());
RETURN;
END
$X$ LANGUAGE plpgsql;
```

Reguły w PostgreSQL

Reguła to zdefiniowana przez użytkownika zasada „przepisania” podanego zapytania SQL na inne zapytanie/wyrażenie. Zapytanie, dla którego jest zdefiniowana reguła jest przekształcane na etapie parsowania zapytań (w miejsce oryginalnej operacji jest podstawiane wyrażenie podane w regule). Reguły mogą dotyczyć operacji: `SELECT`, `INSERT`, `DELETE`, `UPDATE` wykonywanej na rowolnej tabeli lub perspektywie. Ogólny schemat reguły jest następujący:

```
CREATE [ OR REPLACE ] RULE  nazwa_reguły AS
    ON instrukcja_SELECT_INSERT_DELETE_lub_UPDATE
    TO tabela_lub_perspektywa [ WHERE warunek_dot_krotki ]
    DO [ ALSO | INSTEAD ] { NOTHING | instrukcja |
        ( instrukcja ; instrukcja ... ) }
```

Instrukcje reguły to zapytania `SELECT`, `INSERT`, `DELETE` i `UPDATE`. Możemy w nich odwoływać się do dwóch predefiniowanych zmiennych `OLD` i `NEW` oznaczających odpowiednio krotkę sprzed zmian i po zmianach.

Perspektywy modyfikowalne przez reguły

Jednym z zastosowań reguł jest zdefiniowanie operacji modyfikacji na perspektywach — bezpośrednio na perspektywie w PostgreSQL nie można zrobić operacji `INSERT`, `DELETE` czy `UPDATE`, ale można zdefiniować regułę określającą, czym taką operację zastąpić. Poniżej zakładamy, że w tabeli użytkownik pole `kod_uz` jest typu `SERIAL` i jest wypełniane domyślnie.

```
CREATE VIEW absolwent AS
    SELECT nazwisko,imie FROM uzytkownik WHERE semestr=10;

CREATE RULE wstaw_absolwenta AS ON INSERT TO absolwent DO INSTEAD
    INSERT INTO uzytkownik(imie, nazwisko,semestr)
    VALUES(NEW.imie,NEW.nazwisko,10);

CREATE RULE popraw_absolwenta AS ON UPDATE TO absolwent
WHERE NEW.kod_uz<>OLD.kod_uz DO INSTEAD NOTHING;
```

Dokumentacja PostgreSQL 9.0

Dostępna pod adresem `http://www.postgresql.org/docs/9.0/static/`:

- 35.4 — funkcje SQL;
- 39.1–39.8 — programowanie w PL/pgSQL (deklaracje, instrukcje, SQL, kursory);
- 36,39.9 — wyzwalacze i procedury wyzwalaczy w PL/pgSQL;
- 37.3 — reguły, czyli jak zrobić modyfikowalne perspektywy w PostgreSQL (i nie tylko).

Samodzielne ćwiczenia

1. Napisz funkcję SQL `liczba_osob(int)`, która zwraca liczbę osób zapisanych do grupy o podanym kodzie. Wykorzystaj funkcję, by znaleźć wszystkie grupy w bieżącym semestrze, w których jest zapisanych więcej osób niż pozwala atrybut `max_osoby`.
2. Napisz funkcję SQL `plan_zajec(nr_sali,semestr_id)`, która zwróci uporządkowane według terminu zajęcia w danej sali w danym semestrze. Krotki wynikowe muszą zawierać pola: `termin`, `nazwa` (przedmiotu), `rodzaj` (zajęć) i `nazwisko` (prowadzącego). Zauważ, że dla krotek zwracanych przez funkcję trzeba zdefiniować typ.
3. Napisz funkcję PL/pgSQL `pierwsi_spoznieni(nr_grupy,k)`, która zwraca dane pierwszych *k* studentów, którzy zapisali się do podanej grupy ponad limit. Jeśli nie ma ich aż tylu, to podaj tylu, ilu jest.
4. Napisz wyzwalacz, który każdego studenta zapisującego się na zajęcia do przedmiotu, do którego jest także wykład i/lub repetytorium zapisze także na ten wykład i/lub repetytorium. Wyzwalacz musi najpierw sprawdzić, czy student wcześniej się na ten wykład czy repetytorium nie zapisał, by nie spowodować błędu. Możesz założyc, że w grupach wykładowych i repetytoryjnych jest dość miejsc.
5. Zdefiniuj perspektywę `wykladowcy(kod_uz,nazwisko,nazwa,semestr_id)`, która zwraca kody i nazwiska prowadzących grupy wykładowe wraz z nazwą przedmiotu i numerem semestru. Następnie zdefiniuj regułę pozwalającą usuwać krotki z tej perspektywy. Usunięcie krotki ma oznaczać wykreślenie danych prowadzącego z odpowiedniej grupy wykładowej i pozostawienie w tym miejscu `NULL`.