Bazy Danych

Zaawansowane funkcje SQL. Reguły, funkcje użytkownika w PostgreSQL, wyzwalacze

Antoni Ligęza, Marcin Szpyrka
Sławomir Lichoń

Wykorzystano materiały:
`http://www.postgresql.org/docs/8.3/interactive/extend.html`

Drzewo zapytania

Drzewo zapytania — (ang. query tree) to wewnętrzna strukturalna reprezentacja zapytania SQL wygenerowana przez parser na podstawie analizy komendy SQL użytkownika. Komenda SQL jest defragmentowana; wygenerowane fragmenty są przechowywane oddzielnie.

System reguł — (ang. rule system) jest umieszczony pomiędzy *parserem* a *planerem*. Jest to w istocie mechanizm typu *term rewriting*; przetwarza on jedno drzewo zapytania na inne.

Komenda SQL jest dzielona na fragmenty:

- `the command type` — `SELECT, INSERT, UPDATE, DELETE`,
- `the range table` — tablice na których działa zapytanie (po `FROM`),
- `the result relation` — tablica docelowa dla `INSERT, UPDATE, DELETE`,
- `the target list` — lista wyrażeń tworzących wynik zapytania (po `SELECT`),
- `the qualification` — wyrażenie logiczne określające kiedy operację należy wykonać (analogiczne do `WHERE`),
- `the join tree` — struktura drzewa łączenia tablic ()po `FROM`,
- `the others` — inne części zapytania (np. po `ORDER BY`).

Reguły

Widoki w PostgreSQL są definiowane komendą `CREATE VIEW` wraz z wewnętrzną komendą `SELECT`.

Deklaracja `CREATE VIEW` stanowi w istocie makro 'podmieniające' instrukcję tworzenia widoku na tworzenie tabeli i reguły typu `ON SELECT`.

W istocie, deklaracja:

```
CREATE VIEW widok AS SELECT * FROM tabela;
```

jest realizowana poprzez użycie nowej tabeli i reguły:

```
CREATE TABLE widok (<lista kolumn taka jak w tabeli>);

CREATE RULE "_RETURN" AS
     ON SELECT TO widok DO INSTEAD
     SELECT * FROM tabela;
```

Reguły w PostgreSQL dostarczają mechanizmu `podmiany` komendy SQL na inną komendę.

Reguły `ON SELECT` są stosowane w ostatniej fazie przetwarzania kodu zapytania.

Można w nich zastosować jedynie pojedynczą instrukcję `SELECT` z opcją `DO INSTEAD` (podmiana).

Reguły pozwalają na zdefiniowanie instrukcji SQL, które powinny być automatycznie dołączone do podawanych przez użytkownika poleceń lub zamiast nich być wstawione i wykonane.

Reguły realizowane są przez standardową ścieżkę przetwarzania instrukcji SQL. Są jednak szybsze gdyż korzystają z optymalizatora zapytań i mniej obciążają system niż procedury składowane. Mają za to dużo mniejsze możliwości.

Składnia definicji reguł

Składnia definicji reguły wygląda następująco:

```
CREATE [ OR REPLACE ] RULE nazwa
    AS ON <event>
    TO tab [ WHERE warunek ]
    DO [ ALSO | INSTEAD ] { NOTHING | zapytanie |
                ( zapytanie ; zapytanie ... ) }
```

Możliwe wartości dla znacznika `event` to `SELECT` oraz `INSERT, UPDATE,DELETE`.

Dla `INSERT, UPDATE,DELETE` reguły pozwalają na:

- `DO ALSO` — dołączanie pewnych instrukcji do innych instrukcji (kreowanie sekwencji instrukcji),
- `DO INSTEAD` — podmianę instrukcji na inną/inne.

W zapisie definicji zdarzenie wraz z warunkiem określają kiedy reguła powinna być zastosowana.

Polecenie może być pojedynczą instrukcją SQL, bądź ich ciągiem ujętym w nawiasy okrągłe.

Słowo kluczowe `INSTEAD` i `ALSO` określa czy reguła ma być wykonana zamiast, czy też wraz ze wskazaną operacją.

Wewnątrz reguły, można korzystać z pewnych predefiniowanych pseudo-relacji `NEW` i `OLD`, które reprezentują modyfikowane rekordy, przed i po wykonaniu instrukcji modyfikacji.

Reguły możemy stosować zarówno do tabel jak i do widoków.

Aby na widoku były możliwe operacje modyfikowania danych, muszą być utworzone odpowiednie reguły z ustawionym znacznikiem `INSTEAD`. W razie ich braku wystąpi błąd. Podobnie system zachowa się jak zostaną wykryte zapętlenia.

Przykłady reguł

Wyświetlanie kontrolne po zmianach (trigger-agr.sql):

```
CREATE RULE ragr
AS ON UPDATE TO prac
        DO ALSO                 -- INSTEAD
SELECT stanowisko, count(*) AS liczba_prac,
               avg(pobory) as Średnia, sum(pobory) AS Suma
FROM prac
GROUP BY stanowisko;
```

Podmiana (trigger-rule-select-agr.sql)

```
CREATE TABLE agrrule
   (stanowisko    varchar(24),
    liczba_prac   bigint,
    Średnia       numeric,
    Suma          numeric
   );

CREATE RULE "_RETURN"
AS ON SELECT TO agrrule
        DO INSTEAD
SELECT stanowisko, count(*) AS liczba_prac,
        avg(pobory) as Średnia, sum(pobory) AS Suma
FROM prac
GROUP BY stanowisko;
```

Przykłady reguł

(trigger-rule-count-down.sql)

```
-- rejestrowanie  liczby insertów do tablicy pra

DROP TABLE positions_count CASCADE;

CREATE TABLE positions_count (
    places    integer        -- ilosc miejsc do obsadzenia
);

INSERT INTO positions_count VALUES (11);

CREATE OR REPLACE RULE positions_count_rul AS
    ON INSERT TO pra
    DO
    (UPDATE positions_count SET places=places-1;
     SELECT 'No more positions!'
         FROM positions_count WHERE places < 1);
```

Przykład zastosowania reguł do monitorowania zmian wysokości poborów

Zadanie: monitorować i rejestrować zmiany poborów; kto i kiedy dokonał jakich zmian.
(trigger-rule-insert.sql)

```
-- rejestrowanie zmian poborów w tablicy pra

CREATE TABLE pobory_log (
   id_prac     text,          -- id_prac ktorego dotyczy zmiana
   pobory_old  numeric(8,2), -- poprzednie pobory
   pobory_new  numeric(8,2), -- nowe pobory
   log_who     text,          -- who did it
   log_when    timestamp      -- when
);

CREATE OR REPLACE RULE pobory_log_rul AS
    ON UPDATE TO pra
    WHERE NEW.pobory <> OLD.pobory
    DO INSERT INTO pobory_log
          VALUES (
                  NEW.id_prac,
                  OLD.pobory,
                  NEW.pobory,
                  current_user,
                  current_timestamp
                 );
```

Przykłady reguł

Przykład wykorzystania reguł:

```
-- widok, ktory wzbogaca tabele dzial o kolumne caly pobor

DROP VIEW pobory_dzial;
CREATE VIEW pobory_dzial AS
 SELECT dzial.id_dzial, dzial.nazwa, dzial.lokalizacja,
          dzial.kierownik, sum(prac.pobory) AS calypobor
   FROM dzial
   LEFT JOIN prac ON dzial.id_dzial = prac.dzial
   GROUP BY dzial.id_dzial, dzial.nazwa,
                  dzial.lokalizacja, dzial.kierownik;

-- dodanie do pobory_dzial spowoduje dodanie do dzial
CREATE OR REPLACE RULE r1 AS
    ON INSERT TO pobory_dzial DO INSTEAD
         INSERT INTO dzial (id_dzial, nazwa,
                            lokalizacja, kierownik)
  VALUES (new.id_dzial, new.nazwa,
                    new.lokalizacja, new.kierownik);

-- modyfikacja na pobory_dzial spowoduje
-- modyfikacje na dzial
CREATE OR REPLACE RULE r2 AS
    ON UPDATE TO pobory_dzial DO INSTEAD
    (
        UPDATE dzial SET id_dzial = new.id_dzial,
          nazwa = new.nazwa,
          lokalizacja = new.lokalizacja,
          kierownik = new.kierownik
```

```
        WHERE dzial.id_dzial = old.id_dzial;
        --zmiana w calypobor rozkladana
        -- na wszystkich prac
        UPDATE prac SET
          pobory = (
                    prac.pobory +
                    (new.calypobor - old.calypobor)
                        / (( SELECT count(*) AS count
                                FROM prac
                   WHERE prac.dzial = old.id_dzial))
                    )
        WHERE prac.dzial = old.id_dzial;
    );
```

Zastosowanie

Modyfikacja kolumny `calypobor` powoduje zmodyfikowanie wierszy w tabeli `prac`:

```
UPDATE pobory_dzial SET calypobor = 5000
     WHERE id_dzial = 'PK101';

SELECT * FROM pobory_dzial
     WHERE id_dzial = 'PK101';
```

id_dzial	nazwa	lokalizacja	kierownik	calypobor
PK101	Projektowy	Mysieko	1101	5000.00

Reguły mogą również działać na tablicach. Poniższe instrukcje pozwolą na usunięcie rekordów z tabel PRAC i DZIAL:

```
CREATE RULE del_prac AS ON DELETE TO prac DO
  UPDATE dzial SET kierownik=NULL WHERE kierownik=OLD.id_prac;
CREATE RULE del_dzial AS ON DELETE TO dzial DO
  DELETE FROM prac WHERE dzial=OLD.id_dzial;
```

Zastosowanie reguł nie wymaga instalacji w systemie żadnego języka proceduralnego. Można określać szczegółowe uprawnienia do ich tworzenia za pomocą poleceń GRANT RULE oraz REVOKE RULE.

Funkcje

Poza funkcjami wbudowanymi PostgreSQl pozwala na definiowanie funkcji przez użytkownika.

W tym celu PostgreSQL umożliwia korzystanie z języków proceduralnych. Przed wykorzystaniem danego języka, należy wprowadzić w PostgreSQL ustawienia włączające obsługę tego języka. Dla języka PL/pgSQL program obsługi jest dołączony w dystrybucji jako współdzielona biblioteka.

Oferowane są następujące możliwości realizacji funkcji użytkownika:

SQL — funkcje definiowane w oparciu o język SQL,

PL/pgSQL — funkcje definiowane w oparciu o język wewnętrzny proceduralny język programowania PostgreSQL,

C — funkcje definiowane w oparciu o język C,

Tcl — funkcje definiowane w oparciu o język Tcl,

PL/Perl — funkcje definiowane w oparciu o język Perl,

Python — funkcje definiowane w oparciu o język Python.

Funkcje obsługiwane są specjalnym programem (ang. handler) napisanym w języku C. Handler sam obsługuje język lub pośredniczy w przekazaniu wykonania funkcji do zewnętrznej implementacji języka.

Aby zainstalować język PL/pgSQL dla bazy danych, można skorzystać z polecenia `CREATE LANGUAGE`.

```
createlang plpgsql nazwa_bazy
```

Informacje o załadowanych językach przechowywane są w tabeli `pg_language`.

```
select * from pg_language;
drop language 'nazwa_jezyka';
```

Funkcje – podstawy

Istnieje możliwość zdefiniowania własnych funkcji i wykorzystania ich wewnątrz bazy danych PostgreSQL.

```
create function nazwa ([typ_funkcji [,  ...]])
  returns typ_wyniku
  as definicja
  language nazwa_języka
```

Definicję funkcji podaje się jako pojedynczy ciąg znaków, który może obejmować kilka wierszy. Można go zapisać w dowolnym języku programowania obsługiwanym przez PostgreSQL jako ładowalny język proceduralny. Po utworzeniu funkcji, jej definicja jest zapisywana w bazie danych.

Kiedy funkcja jest wywoływana po raz pierwszy, definicja funkcji jest kompilowana przez program obsługi do postaci wykonywalnej, a następnie wykonywana. Dopiero użycie funkcji pozwala na wykrycie jej ewentualnych błędów.

PostgreSQL uznaje funkcje za różne, jeżeli mają one różne nazwy lub różną liczbę parametrów, albo parametry są innego typu. Możemy zatem przeciążać nazwy funkcji.

```
select prosrc from pg_proc where proname='nazwa_funkcji';
drop function nazwa_funkcji;
```

Funkcje w PostgreSQL

Ogólna postać deklaracji funkcji w PostgreSQL:

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ IN | OUT | INOUT ]
            [ argname ] argtype [, ...] ] )
    [ RETURNS [SETOF] rettype ]
  { LANGUAGE langname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT |
      RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER |
      [ EXTERNAL ] SECURITY DEFINER
    | COST execution_cost
    | ROWS result_rows
    | SET configuration_parameter
      { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol' -- funkcje C
  } ...
    [ WITH ( attribute [, ...] ) ]
```

Zastąpienie starej definicji funkcji jest możliwe tylko wtedy, gdy nazwa i typ samej funkcji, oraz jej argumentów są takie same. W przeciwnym wypadku stwarzana jest nowa instancja funkcji.

Znacznik `SETOF` powoduje, że funkcja może zwrócić wynik, który zawiera wiele rekordów. W przypadku jego braku zwracany jest zawsze pojedynczy rekord, lub wartość skalarna.

Znaczniki – znaczenie

Znaczniki IMMUTABLE, STABLE i VOLATILE określają poziom zmienności funkcji. Korzystając z nich mechanizmy optymalizujące:

- VOLATILE - funkcja może wykonywać wszelkie modyfikacje na bazie danych. Jej zwracana wartość może się zmieniać w sposób dowolny (nawet dla tych samych argumentów może zwracać różne wyniki). Funkcje z tym znacznikiem nie podlegają optymalizacji.
- STABLE - funkcja nie może modyfikować bazy danych i dla tych samych argumentów, oraz nie zmodyfikowanej odpytywanej tabeli zwraca te same wyniki. Mechanizmy optymalizujące przetrzymują wyniki działania tej funkcji dla różnych danych wejściowych i zmieniają je jeżeli nastąpiły (nie równolegle) pewne modyfikacje.
- IMMUTABLE - funkcja nie może modyfikować bazy danych i dla tych samych argumentów, niezależnie od stanu bazy danych zwraca te same wyniki. Funkcje z tym znacznikiem wykonywane są tylko raz dla tych samych danych wejściowych.

Znaczniki CALLED ON NULL INPUT, RETURNS NULL ON NULL INPUT, STRICT określają jak funkcja ma się zachować w razie podania jej wartości *NULL* jako argument.

Jeżeli funkcja ma ustawiony znacznik STRICT, to w takim wypadku zostanie wywołany wyjątek.

Jeżeli użyty został znacznik RETURNS NULL ON NULL INPUT to funkcja zakończy swoje działanie i zwróci wartość *NULL*.

W ostatnim wypadku funkcja będzie wykonywana normalnie i to w ciele funkcji powinna znaleźć się obsługa wartości *NULL* któregoś z argumentów.

Znaczniki SECURITY INVOKER i SECURITY DEFINER określają z jakimi prawami ma zostać wywołana funkcja — czy użytkownika który stworzył funkcję (DEFINER), bądź użytkownika, który wykonuje funkcję (INVOKER). Standardowo wywołanie funkcji następuje z prawami użytkownika wywołującego.

Funkcje w SQL – podstawy

Do tworzenia funkcji można w najprostszym przypadku wykorzystać czysty język SQL. W funkcjach takich nie istnieją struktury sterujące, można wykorzystywać jedynie instrukcje SQL bazy danych PostgreSQL. Są dostępne natomiast parametry podobnie jak w PL/pgSQL.

Zaletą wykorzystywania języka SQL dla procedur składowanych jest brak konieczności ładowania do bazy danych procedury obsługi języka PL/pgSQL.

Język SQL pozwala na zwrócenia przez funkcję więcej niż jednego wiersza danych, jeżeli typ zwracanej wartości zadeklarowany został jako typ SETOF, a następnie wykorzystano odpowiednią instrukcję `SELECT`:

```
create or replace function sqlf(nazwa text)
returns setof klienci as
$$
  select * from klienci where miasto = nazwa;
$$
language sql;

select sqlf('Kraków');
```

Funkcje SQL

Funkcje SQL jako języka definicji funkcji używają wyłącznie SQL. Mają zatem ograniczone możliwości przetwarzania danych (np. brak rekurencji, iteracji).

Nie wymagają deklaracji języka.

```
CREATE FUNCTION name([args_data_type])
           RETURNS [SETOF] data_type AS
    $$ [BODY] $$
    LANGUAGE SQL;
```

Funkcje te mogą być wywoływane z poziomu zapytań SQL.

Wewnątrz takiej funkcji można używać komend SQL (również tych modyfikujących dane `INSERT, UPDATE, DELETE`.) Wyjątek stanowią polecenia do obsługi transakcji: `BEGIN, COMMIT, ROLLBACK, SAVEPOINT`.

Ciało funkcji musi być umieszczone pomiędzy znacznikami stringów (' [BODY] ', $ [BODY] $, $BODY$ [BODY] $BODY$ itp.), tak więc ciało funkcji jest przekazywane jako tekst.

Listę argumentów stanowią oddzielone przecinkami kolejne typy tych argumentów podane w nawiasie zaraz po nazwie funkcji. Dostęp do tych argumentów z poziomu ciała funkcji odbywa się przez znacznik $ i numer argumentu. Np.:

```
$1  -- odwolanie do pierwszego argumentu

$2  -- odwolanie do drugiego argumentu
```

Funkcje SQL są najprostsze w definiowaniu; maja jednak ograniczone możliwości.

Funkcje SQL — c.d.

Funkcja zwraca wartość ostatniej komendy `SELECT`.

Jeżeli przy definicji funkcji wyrażenie `SETOF` nie zostało użyte to zwracany jest pierwszy rekord (jeżeli w zapytaniu nie użyto ORDER BY trudno określić który wiersz z odpytywanej tabeli zostanie zwrócony).

W przeciwnym wypadku zwracany jest pełny wynik (function-sql.sql):

```
CREATE FUNCTION f1(char(5),char(5)) RETURNS prac AS
    $$
    SELECT * FROM prac WHERE
                  id_prac = $1 OR id_prac = $2
    $$
    LANGUAGE SQL;
CREATE FUNCTION f2(char(5),char(5)) RETURNS SETOF prac AS
    $$
    SELECT * FROM prac WHERE
                  id_prac = $1 OR id_prac = $2
    $$
    LANGUAGE SQL;
SELECT * FROM f1('110','101');
```

id_prac	nazwisko	imie	data_ur	dzial	stanowisko	pobory
101	Kowal	Adam	1989-12-15	PD303	robotnik	1500.00

```
SELECT * FROM f2('11','101');
```

id_prac	nazwisko	imie	data_ur	dzial	stanowisko	pobory
101	Kowal	Adam	1989-12-15	PD303	robotnik	1500.00
110	Kowalik	Artur	1998-12-13	PR202	majster	1500.00

Funkcje – typy polimorficzne

Funkcje SQL jako argument mogą przyjmować i zwracać następujące typy polimorficzne:

- *anyelement*,
- *anyarray*,
- *anynonarray*, oraz
- *anyenum*.

```
CREATE FUNCTION make_array(anyelement, anyelement)
        RETURNS anyarray AS
    $$
    SELECT ARRAY[$1, $2];
    $$
    LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray,
       make_array('a'::text, 'b') AS textarray;
```

intarray	textarray
1,2	a,b

Funkcje języka proceduralnego PL/pgSQL

Funkcje napisane w PL/pgSQL lub(PL/Tcl). W dalszych punktach tego opracowania zostanie szczegółowo przedstawiony język PL/pgSQL.

`PL/pgSQL` jest językiem proceduralnym, za pomocą którego można łatwo programować serwer bazy danych.

Z poziomu tego języka w sposób bezpośredni możliwy jest dostęp do wszystkich elementów języka SQL (tabele, funkcje itp.).

Funkcje napisane w tym języku mogą pobierać i zwracać te same argumenty co funkcje SQL, w tym również typ *void* i rezultaty zwracane przez komendę SELECT.

Funkcje PL/pgSQL działają w transakcji zewnętrznej, która nie może być odwołana z wnętrza funkcji. Jeżeli wystąpi błąd zostaje ona odwołana. Aby tego uniknąć należy użyć instrukcji EXCEPTION. To jedyna możliwość obsługi transakcji w PL/pgSQL.

Aby używać język `PL/pgSQL` należy go zadeklarować w bazie danych. Można to zrobić poleceniem:

```
CREATE TRUSTED PROCEDURAL LANGUAGE 'plpgsql'
  HANDLER plpgsql_call_handler
  VALIDATOR plpgsql_validator;
```

PL/pgSQL – podstawy

PL/pgSQL to język blokowo-strukturalny, podobny do języka Pascal lub C, z deklaracjami zmiennych i zakresami bloków. Każdy blok ma opcjonalną etykietę, może posiadać kilka deklaracji zmiennych i zamyka instrukcje tworzące blok pomiędzy słowami kluczowymi `BEGIN` oraz `END`.

```
[<<etykieta>>]
declare deklaracje
begin
  instrukcje
end;
```

W języku PL/pgSQL wielkość liter nie ma znaczenia.

Funkcje PL/pgSQL mogą nie mieć argumentów lub mogą mieć ich kilka. Typ parametrów podaje się w nawiasach, po nazwie funkcji. Można stosować wbudowane typy PostgreSQL, takie jak int4 lub float8. Jeżeli nie podamy nazw parametrów, to odwołania do parametrów wewnątrz treści funkcji mają postać $1, $2 itd., w kolejności ich definiowania.

Typ zwracanego wyniku określa się w klauzuli `RETURNS` definicji funkcji.

Bloki instrukcji

PL/pgSQL posługuje się zagnieżdżonymi blokami instrukcji:

```
[ <<label>> ]
[ DECLARE
    deklaracje ]
BEGIN
    instrukcje
[ EXCEPTION --
    WHEN warunek [ OR warunek ... ] THEN
        obsluga wyjatku
    [ WHEN warunek [ OR warunek ... ] THEN
          obsluga wyjatku
      ... ] ]
END [ label ];
```

Instrukcje w bloku mogą odpowiadać za:

- Przypisanie wartości do zmiennej(operator :=)
- Wykonanie zapytania SQL nie zwracających żadnych wartości (INSERT, UPDATE, DELETE)
- Wykonanie zapytania SQL zwracającego pojedynczy wynik i zapisującego go do zmiennej bloku:

  ```
  SELECT wyrazenie-select INTO [STRICT] target FROM ...;
  INSERT ... RETURNING wyrazenie INTO [STRICT] target;
  UPDATE ... RETURNING wyrazenie INTO [STRICT] target;
  DELETE ... RETURNING wyrazenie INTO [STRICT] target;
  ```

  Nie wyspecyfikowanie słowa STRICT powoduje, że do zmiennej *target* zostanie przypisana pierwsza wartość zwrócona przez zapytanie, lub wartość *NULL*. Dodanie tego słowa spowoduje wystąpienie wyjątku jeżeli w

zwróconym wyniku z zapytania będzie więcej niż jeden wiersz, lub wynik będzie pusty:

```
BEGIN
    SELECT * INTO STRICT myrec
         FROM prac WHERE nazwisko = myname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'prac % not found', myname;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'prac % not unique', myname;
END;
```

Ponadto w celu sprawdzenia poprawności wyszukanych danych można sprawdzić zmienną FOUND.

- Wywołanie dynamicznego zapytania:

```
EXECUTE command-string [ INTO [STRICT] target ];
```

- Zagnieżdżony blok instrukcji.

Przykład funkcji – podstawy

```
create function zysk(cena numeric(7,2))
returns numeric(7,2) as
$$
begin
  if cena < 75.00 then return 0.05 * cena;
  elseif cena >= 75.00 and cena <= 110.00
    then return 0.07 * cena;
  else return 0.08 * cena;
  end if;
end;
$$
language plpgsql;

drop function zysk(numeric(7,2));

create or replace function zysk(cena numeric(7,2))
returns numeric(7,2) as
...
```

Przykład 2 – podstawy

```
create or replace function suma_zamowien(klient text)
returns numeric(7,2) as
$$
declare
  w numeric(7,2);

begin
  select into w sum(cena) from zamowienia
    where idnadawcy = klient;
  return w;
end;
$$
language plpgsql;
```

Wewnątrz funkcji PL/pgSQL można deklarować lokalne zmienne. Każda zmienna musi mieć określony typ, który może być jednym z wbudowanych typów PostgreSQL, typem użytkownika lub typem odpowiadającym wierszowi w tabeli.

Deklaracje zmiennych dla funkcji zapisuje się w sekcji `DECLARE` definicji funkcji lub w bloku wewnątrz funkcji. Zmienne zadeklarowane w bloku są widoczne tylko wewnątrz niego oraz w innych blokach, które się w nim znajdują. Zmienna zadeklarowana w bloku wewnętrznym, posiadająca tę samą nazwę jak zmienna na zewnątrz bloku, przesłania zmienną zewnętrzną.

Przypisania – rekordy

Rekord jest rodzajem typu złożonego, który w czasie deklarowania zmiennej nie opiera się na konkretnej tabeli. Zawiera on pola, które są zgodne w czasie wykonania z przypisaną do niego wartością. Rekordy można wykorzystać do zapisywania wyników instrukcji `SELECT`.

Operacje przypisania są wykonywane przez serwer PostgreSQL jako instrukcje `SELECT` nawet, gdy używany jest operator :=. Poprzez dołączenie klauzuli `FROM` można wykorzystać `SELECT` do przypisywania zmiennym wartości z bazy danych.

Należy zapewnić, że instrukcja `SELECT` zwróci tylko jeden wiersz, ponieważ dodatkowe wiersze zostaną pominięte bez wyświetlania jakichkolwiek komunikatów, zaś do zmiennej będzie przypisany tylko pierwszy zwrócony wiersz.

Istnieje możliwość, że instrukcja `SELECT` nie zwróci żadnego wiersza. W takim przypadku operacja przypisanie nie będzie wykonana. W PostgreSQL występuje zmienna logiczna `FOUND`, która jest dostępna natychmiast po wykonaniu operacji przypisania z wykorzystaniem instrukcji `SELECT INTO`.

```
wynik record;
select * into wynik from klienci where idklienta = 9;
if not found then
-- dzialania awaryjne
end if;
```

Wyrażenia

Dostęp z zagnieżdżonego bloku do zmiennych bloku macierzystego odbywa się przez *userblock.param_name*.

Deklaracje zmiennych wyglądają następująco:

```
name [CONSTANT] type [NOT NULL]
        [{ DEFAULT | := } expression];
name tabela_nazwa%ROWTYPE;
name typ_laczony_nazwa;
```

Dostęp do argumentów funkcji odbywa się przez nazwę, jeżeli argument jest nazwany, lub przez alias utworzony w sekcji deklaracji:

```
nazwa ALIAS FOR $n;
```

Wyspecyfikowanie wartości zwracanej przez funkcję odbywa się przez argument o numerze 0 (*$0*), lub polecenie RETURN, przy czym RETURN powoduje wyjście z funkcji.

Instrukcje sterujące

Określenie wartości zwracanej:

```
RETURN wyrazenie;
```

Jeżeli w definicji funkcji został użyty znacznik `SETOF` to konieczne jest zastosowanie jednego z poleceń:

```
RETURN NEXT wyrazenie;
RETURN QUERY zapytanie;
```

Obie komendy działają podobnie. Obie mogą działać na typach skalarnych, łączonych i tablicach. Jedyna różnica polega na tym, że:

`RETURN NEXT` dodaje do wyniku końcowego pojedynczy obiekt, lub całą tablice;

`RETURN QUERY` wykonuje podane zapytanie i dodaje jego wynik do zwracanej kolekcji. Obydwa polecenia mogą być stosowane wielokrotnie w jednej funkcji. Ich wyniki są sumowane i po wywołaniu bezparametrowym komendy RETURN zostają zwrócone z funkcji.

Instrukcje warunkowe

Postać instrukcji warunkowej (if-then-elsif):

```
IF wyrazenie-logiczne-1 THEN
 instrukcje-1
[ ELSIF wyrazenie_logiczne-2 THEN
 instrukcje-2 ]
[ ELSIF wyrazenie-logiczne-3 THEN
 instrukcje-3 ]
[ ELSE
 instrukcje ]
END IF;
```

Pętle – podstawy

Najprostszą pętlą jest pętla niekontrolowana, która wykonuje się nieskończenie, o ile nie zamieścimy wewnątrz instrukcji `EXIT` lub `RETURN`.

```
[<<etykieta>>]
loop
  instrukcje
end loop;
```

Wszystkie konstrukcje pętli mogą posiadać etykiety, które są wykorzystywane w instrukcji `EXIT`. Instrukcja ta powoduje zakończenie odpowiedniej pętli. Etykieta musi odnosić się do bieżącej pętli lub do pętli zewnętrznej.

Jeżeli etykieta nie jest określona, zakończona zostaje pętla bieżąca. Jeżeli występuje klauzula `WHEN`, instrukcja `EXIT` nie wykonuje się, chyba że wartością wyrażenia jest `TRUE`.

Podobnie funkcjonuje instrukcja `CONTINUE`.

```
<<infinite>>
loop
  n:=n+l;
  exit infinite when n>=10;
end loop;
```

Pętle – podstawy c.d.

```
while wyrazenie
loop
  instrukcje
end loop;

for nazwa in [reverse] od .. do
loop
  instrukcje
end loop;

for wiersz in select ...
loop
instrukcje
end loop;
```

Dla każdego z wierszy zwracanego przez instrukcję `SELECT`, zmienna wiersz przyjmuje wartość tego wiersza, po czym wykonywane są instrukcje.

Zmienną wykorzystywaną do zapamiętania wiersza należy wcześniej zadeklarować jako record lub rowtype.

Ostatni przetworzony wiersz będzie dostępny również po zakończeniu wykonywania pętli lub przerwaniu jej działania za pomocą instrukcji `EXIT`.

Pętle

Postać pętli:

```
[ <<label>> ]
LOOP
    instrukcje
    [ CONTINUE [ label ] [ WHEN wyrazenie-logiczne-1 ] ];
    [ EXIT [ label ] [ WHEN wyrazenie-logiczne-2 ] ];
END LOOP [ label ]

[ <<label>> ]
WHILE wyrazenie-logiczne LOOP
    instrukcje
END LOOP [ label ];

[ <<label>> ]
FOR zmienna IN [ REVERSE ]
             wyrazenie-poczatek .. wyrazenie-koniec
                [ BY wyrazenie-krok ] LOOP
    instrukcje
END LOOP [ label ];

[ <<label>> ]
FOR zmienna-record IN zapytanie-select LOOP
    instrukcje
END LOOP [ label ];
```

Wywoływanie wyjątków

```
RAISE level 'format' [, wyrazenie [, ...]];
```

Ta instrukcja powoduje wystąpienie wyjątku, lub przesłanie wiadomości.

Możliwe wartości wyrażenia *level* to: `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, `EXCEPTION`.

Tylko `EXCEPTION` powoduje wystąpienie wyjątku (odwołuje też transakcje). Pozostałe poziomy generują wiadomość o różnym priorytecie. Te wiadomości są przesyłane do klienta bazy danych, lub zapisane do logu serwera.

Sterowanie obsługi tych wiadomości odbywa się przez zmienne konfiguracyjne serwera: *log_min_messages* i *client_min_messages*.
Przykładowe wywołanie:

```
RAISE EXCEPTION 'Nie isteniejące ID --> %', user_id;
```

Przykładowa funkcja

```
CREATE TYPE nadplata_result AS
   (nazwisko varchar(32),
    bnadplata bool);
CREATE FUNCTION pc_nadplata(gr int4)
  RETURNS SETOF nadplata_result AS
$BODY$
DECLARE
    p prac%rowtype;
--zmiena typu prac
BEGIN
    FOR p IN SELECT * FROM prac LOOP
IF p.pobory > gr THEN
            --dodanie rekordu do zwracanej kolekcji
            RETURN QUERY (SELECT p.Nazwisko AS Nazwisko,
                          false AS bNadplata);
        ELSE
            --dodanie rekordu do zwracanej kolekcji
            RETURN QUERY (SELECT p.Nazwisko AS Nazwisko,
                          true AS bNadplata);
        END IF;
    END LOOP;
    RETURN; --zwraca cala kolekcje
END;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE;
```

Przykład wywołania

```
SELECT * FROM pc_nadplata(1500)
    WHERE nazwisko = 'Kowal' OR nazwisko = 'Kowalski'
```

nazwisko	nadplata
Kowal	t
Kowalski	f

Funkcje wewnętrzne, funkcje w C

Funkcje wewnętrzne — są to funkcje napisane w C statycznie podlinkowane do serwera PostgreSQL. „Ciało" deklaracji funkcji zawiera nazwę funkcji języka C. Jeżeli nic nie wyszczególniono to brana jest funkcja która posiada taką samą nazwę jak nazwa funkcji wewnętrznej. Przykład deklaracji:

```
CREATE FUNCTION square_root(double precision)
                RETURNS double precision
    AS 'dsqrt'
    LANGUAGE internal
    STRICT;
```

Podczas uruchamiania bazy wszystkie statycznie linkowane funkcje C są deklarowane jako wewnętrzne (*initdb*).
Funkcje języka C - są to funkcje napisane w języku C, skompilowane i umieszczone w bibliotekach współdzielonych, które mogą być dynamicznie podładowane przez serwer. Właśnie to dynamiczne ładowanie rozróżnia „funkcje wewnętrzne" od „funkcji języka C".

Definicja tej funkcji musi zawierać ścieżkę do modułu dynamicznie ładowanego, oraz nazwę funkcji z tej biblioteki. Przy podawaniu ścieżki, można skorzystać ze zmiennych konfiguracyjnych *$libdir* lub *dynamic_library_path*, lub podać ścieżkę bezpośrednią.

Podczas ładowania modułu wołana jest funkcja *_PG_init*. Umieszczając ją w swoim module użytkownik może dokonać pewnych predefiniowanych ustawień. Podczas odłączania modułu wołana jest funkcja *_PG_fini*, za pomocą której użytkownik może zwolnić używane zasoby.

Funkcje w C

Funkcje pisane w języku C mogą korzystać z typów bazowych po załączeniu odpowiednich plików definiujących (tablica nr. 1).

Dostęp do argumentów funkcji odbywa się za pomocą *PG_GETARG_XXX(nr_arg)*, gdzie *XXX* jest nazwą typu argumentu. Możliwe jest również przekazanie argumentów o typie łączonym za pomocą funkcji *GetAttributeByName* dostępnej po załączeniu „*executor/executor.h*”:

```
#include <postgres.h>
#include <executor/executor.h> /*GetAttributeByName()*/
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
PG_FUNCTION_INFO_V1(c_nadplata);
Datum
c_nadplata(PG_FUNCTION_ARGS)
{
    HeapTupleHeader  t = PG_GETARG_HEAPTUPLEHEADER(0);
        /*Wiersz tabeli*/
    int32            limit = PG_GETARG_INT32(1);
    bool isnull;
    Datum salary;

    salary = GetAttributeByName(t, "pobory", &isnull);
    if (isnull)
        PG_RETURN_BOOL(false);
    PG_RETURN_BOOL(DatumGetInt32(salary) > limit);
}
```

Kompilacja

Po zapisaniu tego kodu do pliku „*foo.c*” można go skompilować i stworzyć bibliotekę współdzieloną:

```
cc -fpic -c foo.c -I /usr/include/postgresql/8.3/server
cc -shared -o foo.so foo.o
sudo cp foo.so /usr/lib/postgresql/8.3/lib
```

Gdzie „/usr/include/postgresql/8.3/server” to ścieżka do plików nagłówkowych interfejsów serwera (pg_config –includedir-server), a „/usr/lib/postgresql/8.3/lib” to miejsce przechowywania obiektów dzielonych serwera (pg_config –pkglibdir).

Funkcje w C - deklaracja i użycie w SQL

Deklaracja i użycie w SQL:

```
CREATE FUNCTION c_nadplata(prac, integer)
                          RETURNS boolean
    AS '$libdir/foo.so', 'c_nadplata'
    LANGUAGE C STRICT;

-- przyklad wolania

SELECT nazwisko, c_nadplata(prac, 1500) AS nadplata
    FROM prac
    WHERE nazwisko = 'Kowal' OR nazwisko = 'Kowalowski';
```

nazwisko	nadplata
Kowal	f
Kowalowski	t

Z poziomu funkcji języka C można również używać typów polimorficznych *anyelement*, *anyarray*, *anynonarray* i *anyenum*. Służą do tego funkcje *get_fn_expr_argtype* i *get_fn_expr_rettype*, które pozwalają na określenie typu argumentu i wartości zwracanej. Funkcje C mogą również dokonywać modyfikacji na bazie danych dzięki interfejsowi serwera[1].

[1] http://www.postgresql.org/docs/8.3/interactive/spi.html

Procedury wyzwalane — idea

Procedury wyzwalane — predefiniowane operacje na danych wyzwalane zdarzeniami. Procedury wyzwalane są definiowane przez użytkownika i uruchamiane automatycznie.

Inne nazwy: wyzwalacze, triggery.

Procedury wyzwalane mogą być definiowane za pomocą SQL (wyłącznie, ale jako funkcje SQL) lub też mogą zawierać definicje funkcji w innych językach (np. PL/pgSQL, C).

Wyzwalacze uruchamiane są zdarzeniami:

- `INSERT`,
- `UPDATE`,
- `DELETE`.

Wyzwalacze mogą być uruchamiane:

- `BEFORE`,
- `AFTER`,

czyli odpowiednio *przed* i `po` modyfikacji danych.

Wyzwalacze mogą być uruchamiane:

- `FOR EACH ROW`,
- `FOR EACH STATEMENT`,

czyli dla każdego wiersza, którego dotyczy działanie lub raz dla całej instrukcji.

Wyzwalacze – podstawy

Wykorzystanie *procedury wyzwalanej* (ang. trigger) pozwala na automatyczne uruchamianie procedury zapisanej w bazie danych, jeżeli dla określonej tabeli będą podjęte takie działania, jak: `INSERT, UPDATE, DELETE`.

Aby wykorzystać procedurę wyzwalaną, najpierw należy zdefiniować procedurę, a następnie utworzyć sam wyzwalacz, który określa, kiedy procedura wyzwalana będzie wykonywana.

```
create trigger nazwa {before  | after}
{zdarzenie [or...]}
on table for each {row | statement}
execute procedure funkcja(argumenty)
```

Wyzwalacz w istocie informuje: *„Uruchom tę procedurę przechowywaną w bazie danych za każdym razem, kiedy dla tej tabeli zachodzi w bazie danych określone zdarzenie”*.

Wyzwalacz może mieć nazwę, którą można wykorzystać do jego usunięcia, kiedy nie będzie już potrzebny, poprzez wykonanie polecenia:

```
drop trigger nazwa on tabela;
```

Wyzwalacze

Funkcje języka PL/pgSQL bardzo często wykorzystuje się do pisania tak zwanych *wyzwalaczy*, czyli elementów aktywnych, które podobnie jak reguły są wykonywane gdy wystąpi odpowiednie zdarzenie.

Deklaracja wyzwalacza wymaga uprzedniego zdefiniowania funkcji!

Wyzwalacze różnią się od reguł językiem w którym są tworzone (reguły wykorzystują składnie SQL). Dzięki zastosowaniu funkcji PL/pgSQL wyzwalacze mogą modyfikować, lub nawet odwołać działanie instrukcji które je uruchamiają.

Jedną funkcję można użyć do deklaracji kilku wyzwalaczy:

```
CREATE TRIGGER nazwa
            { BEFORE | AFTER } { zdarzenie [ OR ... ] }
    ON tabela_nazwa [ FOR [ EACH ] { ROW | STATEMENT } ]
    EXECUTE PROCEDURE funkcja_nazwa ( argumenty_wartosci )
```

Znaczniki BEFORE i AFTER określają czy wyzwalacz ma się wykonać przed czy po modyfikacji.

Zdarzenie może przybierać wartości: `INSERT, UPDATE, DELETE`.

Jeżeli wyzwalacz ma być uruchamiany dla każdego wiersza tabeli, to należy użyć znacznika `FOR EACH ROW`.

Domyślny znacznik `FOR EACH STATEMENT` sprawia, że wyzwalacz jest wołany raz dla całej tabeli.

Argumenty przekazywane do funkcji wyzwalacza muszą mieć postać stringów i mogą zostać użyte przez zmienną TG_ARGV[].

Wyzwalacze – uwagi

Wyzwalacz zadziała, kiedy zajdzie określone zdarzenie. Można zażądać, by zadziałał on po zajściu zdarzenia, kiedy to wywoływana procedura będzie miała dostęp do danych pierwotnych (dla aktualizacji i usunięcia) oraz nowych danych (dla wstawiania i aktualizacji). Można także zażądać, by zadziałał przed zajściem zdarzenia.

Jeśli aktualizacja wielu wierszy spowoduje działanie wyzwalacza, można wybrać, czy wyzwalacz ma działać dla każdego aktualizowanego wiersza (`ROW`), czy też raz dla całej operacji aktualizacji (`STATEMENT`).

Wyzwalacz działa w momencie, kiedy są spełnione określone warunki i wykonuje tzw. *procedurę wyzwalaną*. Procedurę wyzwalaną tworzy się jako funkcję bez parametrów oraz o specjalnym typie wyniku `TRIGGER` Procedura musi zwracać wartość `NULL` lub wiersz odpowiadający strukturze tabeli, dla której ją uruchomiono.

Dla wyzwalaczy typu `AFTER`, które wykonuje się po operacji `UPDATE`, zaleca się, aby procedura wyzwalana zwracała wartość `NULL`.

Dla wyzwalaczy typu `BEFORE`, zwracany wynik wykorzystuje się do sterowania aktualizacją, która ma być wykonana. Jeżeli procedura wyzwalana zwraca `NULL`, operacja `UPDATE` nie jest wykonywana, jeżeli zwracany jest wiersz danych, jest on wykorzystywany jako źródło aktualizacji, dając okazję procedurze wyzwalanej do zmiany danych przed zatwierdzeniem ich w bazie danych. W przypadku wyzwalaczy typu `STATEMENT` funkcja powinna zwracać `NULL`.

Zmienne specjalne w wyzwalaczach

Funkcja napisana w PL/pgSQL, która ma być wołana przez wyzwalacz nie może zawierać żadnych argumentów i musi zwracać typ `trigger`. Wewnątrz funkcji wyzwalacza dostępne są specjalne zmienne:

- OLD – Reprezentuje wiersz przed modyfikacją, lub jest NULL-em jeżeli użyto znacznik FOR EACH STATEMENT; występuje dla UPDATE/DELETE;
- NEW – Reprezentuje wiersz po modyfikacji, lub jest NULL-em jeżeli użyto znacznik FOR EACH STATEMENT; wystepuje dla INSERT/UPDATE;
- TG_NAME – Nazwa wykonywanego wyzwalacza.
- TG_WHEN – BEFORE lub AFTER
- TG_LEVEL – ROW lub STATEMENT
- TG_OP – INSERT, UPDATE, lub DELETE
- TG_RELID, TG_RELNAME – OID i nazwa tabeli, dla której uruchomiono wyzwalacz
  TGTG _NARGS, TG_ARGV[] – Liczba argumentów i ich kolekcja, przekazanych do funkcji wyzwalacza.

Przykład – podstawy

```
create function fn_zapotrzebowanie() returns trigger as
$$
declare
  s integer;

begin
  update kompozycje set stan = stan - 1
  where idkompozycji = new.idkompozycji;

  select into s stan from kompozycje
  where idkompozycji = new.idkompozycji;

  if s = 1 then
  insert into zapotrzebowanie
    values(new.idkompozycji, current_date, null);
  end if;

  return null;
end;
$$
language plpgsql;

create trigger  tr_zapotrzebowanie after insert on Zamowienia
for each row execute procedure fn_zapotrzebowanie();
```

Przykład

Dzięki wyzwalaczom można uniknąć występowania błędów, przy próbie wykonania operacji wbrew ograniczeniom:

```
CREATE FUNCTION dzial_tr_after()
  RETURNS "trigger" AS
$BODY$
BEGIN
   IF (TG_LEVEL = 'ROW' AND
      (TG_OP = 'UPDATE' OR TG_OP = 'INSERT')) THEN
      IF (NEW.kierownik IS NOT NULL) THEN
         IF (SELECT COUNT(*) FROM prac
           WHERE id_prac = NEW.kierownik) <> 1 THEN
    --nie ma rekordu w prac. Dodajemy odpowiedni
            INSERT INTO prac VALUES (NEW.kierownik, '',
                '', '1999-12-15', NEW.id_dzial,
                'kierownik', 1500);
      END IF;
      END IF;
      RETURN NEW; --Akceputacja modyfikacji
   END IF;
 RETURN NULL;  --Anulujemy
END;
$BODY$
  LANGUAGE 'plpgsql';
CREATE TRIGGER dzial_trigger_after
  AFTER INSERT OR UPDATE
  ON dzial
  FOR EACH ROW
  EXECUTE PROCEDURE dzial_tr_after();
```

Przykład

```
CREATE FUNCTION dzial_tr_befor()
  RETURNS "trigger" AS
$BODY$
BEGIN
   IF (TG_LEVEL = 'ROW' AND
     TG_OP = 'INSERT') THEN
      IF (SELECT COUNT(*) FROM dzial
        WHERE id_dzial = NEW.id_dzial) = 1 THEN
         UPDATE dzial SET nazwa = NEW.nazwa,
    --modyfikujemy rekord dzial
          lokalizacja = NEW.lokalizacja,
          kierownik = NEW.kierownik
          WHERE id_dzial = NEW.id_dzial;
      --tam gdzie id_dzial = NEW.id_dzial
        RETURN NULL;
      END IF;
   RETURN NEW;
   END IF;
   RETURN NULL;
 END;
$BODY$
  LANGUAGE 'plpgsql';
CREATE TRIGGER dzial_trigger_befor
  BEFORE INSERT
  ON dzial
  FOR EACH ROW
  EXECUTE PROCEDURE dzial_tr_befor();
```

Przy próbie zmiany wartości kolumny *kierownik* w tabeli *dzial* na taką, która nie istnieje w tabeli *prac*, automatycznie zostaje dodany wiersz do tabeli *prac*:

```
UPDATE dzial SET kierownik = '9999' WHERE id_dzial='PD303';
SELECT * FROM prac;
```

id_prac	nazwisko	imie	data_ur	dzial	stanowisko	pobory
101	Kowal	Adam	1989-12-15	PD303	robotnik	1500.00
11	Kowalik	Artur	1998-12-13	PR202	majster	1500.00
1011	Kowalewski	Adam	1989-11-15	PR202	kierownik	3500.00
110	Kowalczyk	Amadeusz	1998-12-17	PD303	robotnik	1000.00
1001	Kowalski	Antoni	1999-12-15	PD303	kierownik	4500.00
1101	Kowalowski	Alojzy	1998-11-15	PK101	kierownik	2500.00
111	Kowalczuk	Adam	1998-11-12	PR202	majster	2500.00
9999			1999-12-15	PD303	kierownik	1500.00

Porównanie wyzwalaczy i reguł

Reguły i wyzwalacze rozróżnia przede wszystkim język w którym są pisane, skutkiem czego ich zastosowania mogą się różnić. Różnice te obrazuje tablica nr. 3.

Tablica 1: Wyzwalacze vs. Reguły

	Reguły	Wyzwalacze
Język	SQL	Dowolny, zainstalowany na serwerze
Wydajność	Duża wydajność dzięki optymalizowaniu zapytań, związanych z regułami.	Mniejsza, zwłaszcza dla wyzwalaczy uruchamianych dla każdego modyfikowanego wiersza.
Obsługa widoków	Wszystkie zdarzenia.	Tylko INSERT. Wyzwalacze na UPDATE i DELETE nie wykonają się, gdyż widok to pusta tabela.
Funkcjonalność	Ograniczona. Brak możliwości modyfikowania wprowadzanych danych.	Pełna.
Zmaterializowane widoki	Wykonują się przed wykonaniem modyfikacji, dlatego nie nadają się do odświeżania zmaterializowanych widoków	Wyzwalacze AFTER mogą odświeżać dane w zmaterializowanych widokach po wykonanych modyfikacjach.

Funkcje agregujące

Mając daną zdefiniowaną funkcję możemy zdefiniować funkcję agregującą[2], skutkiem czego będzie można ją wykorzystać w zapytania SQL w celu dokonania operacji na grupie rekordów:

```
CREATE TYPE complex AS ( x real,
   y real);
CREATE TABLE complex_tab (
  val complex
);
INSERT INTO complex_tab (val) VALUES((1,2.3));
INSERT INTO complex_tab (val) VALUES((1,2));
CREATE FUNCTION complex_add(c1 complex, c2 complex)
  RETURNS complex AS $BODY$
DECLARE
    ret complex;
BEGIN
    ret.x := c1.x + c2.x;
    ret.y := c1.y + c2.y;
    RETURN ret;
END; $BODY$
  LANGUAGE 'plpgsql';
CREATE AGGREGATE complex_sum(
  BASETYPE=complex, SFUNC=complex_add,
  STYPE=complex, INITCOND='(0,0)'
);
SELECT complex_sum(val) AS sum FROM complex_tab;
```

sum
(2,4.3)

[2]http://www.postgresql.org/docs/8.3/interactive/xaggr.html

Operatory

PostgreSQL pozwala również definiować operatory[3]. Pozwala to na wygodne korzystanie z funkcji i przysłanianie standardowych działań operatorów. Definicja operatora:

```
CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
SELECT (val + val) AS m FROM complex_tab;
```

m
(2,4.6)
(4,4.6)

[3]http://www.postgresql.org/docs/8.3/interactive/xoper.html

Interfejs indeksu

Po zdefiniowaniu nowego typu możemy go wykorzystywać w tabelach. Nie jest jednak możliwe stworzenie indeksu na kolumnie o nowo zdefiniowanym typie, dopóki nie zostanie zdefiniowana *klasa operatorów* wykorzystywanych przez indeks. Taka klasa może równie dobrze być wykorzystana do przysłaniania standardowego działania indeksów.
W PostgreSQL są dostępne cztery typy indeksów[4], różniących się, algorytmem budowania drzewa rekordów i możliwymi strategiami przeszukiwania:

- B-Tree — B-drzewo, najczęściej wykorzystywane (automatycznie dodawany do kolumn UNIQUE i PRIMARY KEY) wspiera 5 strategi wyszukiwania: *„mniejszy”*, *„mniejszy, lub równy”*, *„równy”*, *„większy, lub równy”*, oraz *„większy”*.
- Hash - bardzo szybki indeks, często wykorzystywany w mechanizmach wewnętrznych PostgreSQL. Wspiera tylko jedną strategie: *„równy”*.
- GiST - uniwersalny interfejs indeksów, który pozwala na zdefiniowanie dowolnej ilości strategi przeszukiwania. W PostgreSQL jest zaimplementowana klasa operatorów korzystające z tego typu indeksu, do obsługi geometrycznych typów dwuwymiarowych.
- GIN - funkcjonalnie zbliżony do GiST typ indeksów, służący do indeksowania list odwrotnych i tablic. Powstał głównie na potrzeby *tsearch*.

[4] http://www.postgresql.org/docs/8.3/interactive/xindex.html

Przykład

Przykład definicji klasy operatorów dla *complex*, korzystający z interfejsu B-Tree:

```
CREATE FUNCTION complex_cmp(c1 complex, c2 complex)
  RETURNS int4 AS $BODY$
DECLARE
    ret1 float;
    ret2 float;
BEGIN
    ret1 := sqrt(c1.x * c1.x + c1.y * c1.y);
    ret2 := sqrt(c2.x * c2.x + c2.y * c2.y);
    IF (ret1 > ret2) THEN
       RETURN 1;
    ELSIF (ret1 < ret2) THEN
       RETURN -1;
    ELSE
       RETURN 0;
    END IF;
END;
$BODY$
  LANGUAGE 'plpgsql' IMMUTABLE STRICT;

CREATE FUNCTION complex_lt(c1 complex,c2 complex)
 RETURNS bool AS $BODY$
DECLARE
    ret1 float;
    ret2 float;
BEGIN
    ret1 := sqrt(c1.x * c1.x + c1.y * c1.y);
    ret2 := sqrt(c2.x * c2.x + c2.y * c2.y);
    RETURN (ret1 < ret2);
```

```
END;
$BODY$
  LANGUAGE 'plpgsql' IMMUTABLE STRICT;
```

C.d.

```
CREATE OPERATOR < (
   leftarg = complex,
   rightarg = complex,
   procedure = complex_lt,
   commutator = > , negator = >= ,
   restrict = scalarltsel, join = scalarltjoinsel
);

CREATE OPERATOR CLASS complex_ops
--klasa operatorow dla typu complex
 DEFAULT FOR TYPE complex USING btree AS
  OPERATOR    1  <,
  FUNCTION    1  complex_cmp(complex, complex);
```

Przykład użycia:

```
CREATE INDEX complex_idx
         ON complex_tab USING btree(val)

SELECT * FROM complex_tab WHERE val < (2,2);
```

val
(1,2.3)

Materializacja widoków

Widoki standardowo są „materializowane” (aktualizowane) za każdym razem, kiedy występuje odwołanie do nich, tzn. zapytanie budujące widok jest wołane i zwracany jest jego wynik przy każdej próbie pobrania rekordów. Jeżeli to zapytanie jest związane z czasochłonnymi obliczeniami, warto jest taki widok „zmaterializować na stałe”, to znaczy wiersze widoku zapisać w jakiejś tabeli i na żądanie odczytu widoku zwracać wiersze z tej tabeli.

Taki widok zmaterializowany wymaga odświeżania danych według jednej z następujących strategii:

- Na żądanie użytkownika,
- Co pewien interwał czasowy,
- Po każdej modyfikacji tabel, których kolumny budują widok.

PostgreSQL, w przeciwieństwie do Oracle, nie ma wbudowanych mechanizmów do obsługi widoków zmaterializowanych. Można jednak zaimplementować taką obsługę widoków. Przykład takiej implementacji zaprezentowano poniżej[5]:

```
CREATE TABLE matviews
(
  mv_name name NOT NULL,
  v_name name NOT NULL,
  last_refresh timestamptz,
  CONSTRAINT matviews_pkey PRIMARY KEY (mv_name)
);
```


[5] http://www.varlena.com/GeneralBits/Tidbits/matviews.html

C.d.

```
CREATE FUNCTION create_matview(name, name)
  RETURNS void AS
$BODY$
DECLARE
    matview ALIAS FOR $1;
    view_name ALIAS FOR $2;
    entry matviews%ROWTYPE;
BEGIN
 SELECT * INTO entry FROM matviews WHERE mv_name = matview;
 IF FOUND THEN
  RAISE EXCEPTION 'Materialized view ''%''already exists.',
   matview;
 END IF;
 EXECUTE 'REVOKE ALL ON ' || view_name || ' FROM PUBLIC';
 EXECUTE 'GRANT SELECT ON ' || view_name || ' TO PUBLIC';
 EXECUTE 'CREATE TABLE ' || matview ||
  ' AS SELECT * FROM ' || view_name;
 EXECUTE 'REVOKE ALL ON ' || matview || ' FROM PUBLIC';
 EXECUTE 'GRANT SELECT ON ' || matview || ' TO PUBLIC';
 INSERT INTO matviews (mv_name, v_name, last_refresh)
  VALUES (matview, view_name, CURRENT_TIMESTAMP);
 RETURN;
END; $BODY$
  LANGUAGE 'plpgsql';
```

Funkcja *create_matview* przyjmuje dwa argumenty: pierwszy to nazwa zmaterializowanego widoku, który ma być utworzony, a drugi to nazwa widoku-wzorca.

C.d.

```
CREATE FUNCTION drop_matview(name)
  RETURNS void AS
$BODY$
DECLARE
    matview ALIAS FOR $1;
    entry matviews%ROWTYPE;
BEGIN
 SELECT * INTO entry FROM matviews WHERE mv_name = matview;
 IF NOT FOUND THEN
  RAISE EXCEPTION 'Materialized view % does not exist.',
  matview;
 END IF;
 EXECUTE 'DROP TABLE ' || matview;
 DELETE FROM matviews WHERE mv_name=matview;
 RETURN;
END; $BODY$ LANGUAGE 'plpgsql';
```

Funkcja *drop_matview* usuwa podany zmaterializowany widok.

C.d.

```
CREATE FUNCTION refresh_matview(name)
  RETURNS void AS
$BODY$
DECLARE matview ALIAS FOR $1;
        entry matviews%ROWTYPE;
BEGIN
 SELECT * INTO entry FROM matviews WHERE mv_name = matview;
 IF NOT FOUND THEN
  RAISE EXCEPTION 'Materialized view % does not exist.',
  matview;
 END IF;
 EXECUTE 'DELETE FROM ' || matview;
 EXECUTE 'INSERT INTO ' || matview
  || ' SELECT * FROM ' || entry.v_name;
 UPDATE matviews
  SET last_refresh=CURRENT_TIMESTAMP
   WHERE mv_name=matview;
 RETURN;
END; $BODY$
 LANGUAGE 'plpgsql';
```

Funkcja *refresh_matview* oblicza na nowo i zapisuje dane w zmaterializowanym widoku.

```
SELECT create_matview('mat_pobory_dzial', 'pobory_dzial');
```

Tak utworzony widok można odświeżyć wykonaniem instrukcji:

```
SELECT refresh_matview('mat_pobory_dzial');
```

Aby odświeżać automatycznie ten widok należy stworzyć wyzwalacz dla tabeli *pobory* ze znacznikiem AFTER, który będzie wykonywał tą instrukcję.