Bases de données avancées
PL/pgSQL

Wiesław Zielonka

17 décembre 2011

Table des matières

Résumé

Ces notes ne sont pas corrigées, mais peut-être vous les trouverez quand même utiles pour préparer l'examen ou projet. Ne pas distribuer.

Les avantages de l'utilisation PL/pgSQL : vitesse - code compilé une fois réside sur le serveur. Structure de base - bloc de code :

```
<label declaration>
DECLARE
   statements
BEGIN
   statements
END
```

1 Déclaration de et initialisation de variables

```
name [ CONSTANT ] type [ NOT NULL ] [ {DEFAULT | := } value]
```

- `CONSTANT` variable non modifiable (read only),
- `NOT NULL` par défaut les variables initialisées avec la valeur NULL,

```
----------------- add.sql ------------------------------------------
CREATE FUNCTION add(integer, integer) RETURNS INTEGER AS $$
DECLARE
  q INTEGER := 0;
BEGIN
  q := $1+$2;
  RETURN q;
END
$$ LANGUAGE plpgsql;
--------------------------------------------------------------------
```

2 Noms de variables. Les types %TYPE et %ROWTYPE

Pour plus de souplesse on peut déclarer de variables du même type que le type d'un attribut ou un type d'une table :

`nom_table.nom_colonne%TYPE`

`nom_de_table%ROWTYPE`

On rappelle que `||` est l'opérateur de concaténation en SQL.

La construction

`expression::type`

permet de faire un « cast » et changer le type d'une expression.

```
----------------------- concat.sql ------------------------------------------------
CREATE OR REPLACE FUNCTION concat(ingredients)
   RETURNS TEXT AS $$
   DECLARE
     a ingredients.name%TYPE;
     b ingredients.unit%TYPE;
     c  ALIAS FOR $1;   -- creation d'un nom alias pour le parametre $1
     x text;
   BEGIN
     a := c.name;
     b := c.unit;
     x := (a ||' ')::CHAR(30) ;
     x := x||b;
```

```
     RETURN x ;
   END
$$ LANGUAGE 'plpgsql';
-- Maintenant o peut faire
-- SELECT concat(ingredients) FROM ingredients;
-- ou
-- SELECT concat(ingredients.*) FROM ingredients;
-- Par contre SELECT concat(ingredients);  ne marche pas
--------------------------------------------------------------------------
CREATE OR REPLACE FUNCTION concat(a ingredients.name%TYPE, b ingredients.unit%TYPE)
                  RETURNS TEXT AS $$
   DECLARE
     x text;
   BEGIN
     x := a::CHAR(30);
     x := x||b;
     RETURN x ;
   END
$$ LANGUAGE 'plpgsql';

--select concat(name,units) from ingredients;
--------------------------------------------------------------------------

---------------------------- changer_prix --------------------------------
CREATE OR REPLACE FUNCTION changer_prix(f items.price%TYPE, combien REAL )
          RETURNS items.price%TYPE AS '
   DECLARE
      toto items%ROWTYPE;
      prix items.price%TYPE DEFAULT 0;
   BEGIN
      prix  := f * combien;
      RETURN prix;
   END
   '
   LANGUAGE 'plpgsql';
--------------------------------------------------------------------------

select changer_prix(price,1.7),price
FROM   items
WHERE  price IS NOT NULL;
```

3 Noms de paramètres

Les paramètres d'une fonction sont nommés `$1, $2,...` etc. On peut donner les noms avec `ALIAS` et les renommer avec `RENAME`.

```
nom ALIAS FOR old_name
RENAME old_name TO new_name;
```

Utile si les variable NEW et OLD dans les triggers. A noter que c'est une déclaration alors à mettre dans DECLARE.

```
---------------------- addnumbers -------------------------------------
CREATE OR REPLACE FUNCTION addnumbers(INTEGER, INTEGER) RETURNS INTEGER AS '
     DECLARE
          Number_1 ALIAS FOR $1;
          Number_2 ALIAS FOR $2;
          RENAME $1 TO Orig_Number;
     BEGIN
          RETURN Orig_Number + Number_2;
     END;
     ' LANGUAGE 'plpgsql';
     ------------------------------------------------------------------------------
```

Les paramètres peuvent être utilisés uniquement pour passer des données, jamais pour passer les noms de tables.

Par exemple

`INSERT INTO mytable VALUES ($1);`

est correct mais

`INSERT INTO $1 VALUES (42);`

ne l'est pas.

4 IF

Les instructions `IF`

```
IF condition THEN
   instructions
ELSIF condition THEN
   instructions
ELSIF condition THEN
   instructions
ELSE
   instructions
END IF;
```

On peut faire de fonctions récursives. Noter aussi l'utilisation d'un bloc interne avec son propre `DECLARE ... BEGIN...END`.

```
------------------------- factorial.sql -----------------------------------
CREATE OR REPLACE FUNCTION my_factorial(value INTEGER) RETURNS INTEGER AS $$
   DECLARE
     arg INTEGER;
   BEGIN
     arg := value;
     IF arg IS NULL OR arg < 0 THEN
          RAISE NOTICE 'Invalid Number';  -- pour faire un affichage sur la console
          RETURN NULL;
     ELSE
         IF arg = 1 THEN
           RETURN 1;
         ELSE
           DECLARE
             next_value INTEGER;
           BEGIN
             next_value := my_factorial(arg - 1) * arg;
             RETURN next_value;
           END;
        END IF;
     END IF;
   END;
 $$ LANGUAGE 'plpgsql';
-------------------------------------------------------------------------------
```

5 Plusieurs fonctions avec le même nom

Comme Java PL/pgSQL supporte plusieurs fonctions avec le même nom si les signatures différentes. Pour supprimer une fonction définie par utilisateur on utilise :
`DROP FUNCTION nom-fonction(parametres);` L'exemple suivant définit trois fonctions qui portent le même nom.

```
------------------------- compute_date.sql ----------------------------
CREATE OR REPLACE FUNCTION compute_due_date(DATE) RETURNS DATE AS $$
   DECLARE           ------ declaration de variables
     due_date        DATE;
     rental_period   INTERVAL := '7 days';
```

```
   BEGIN
     due_date := $1 + rental_period;
     RETURN due_date;
   END;
 $$ LANGUAGE 'plpgsql';
-------------------------------------------------------------------------
CREATE OR REPLACE FUNCTION compute_due_date(DATE, INTERVAL) RETURNS DATE AS $$
   BEGIN
-- c'est un commentaire dans une fonction
     RETURN( $1 + $2 );
   END;
 $$ LANGUAGE 'plpgsql';
-- SELECT compute_due_date(current_date,INTERVAL '1 MONTH');
-------------------------------------------------------------------------
CREATE OR REPLACE FUNCTION compute_due_date(dans INTERVAL) RETURNS DATE AS $$
  BEGIN
    RETURN current_date + dans;
  END
$$ LANGUAGE 'plpgsql';
-- select compute_due_date(interval '3 months');
-------------------------------------------------------------------------
```

6 Boucles LOOP

Boucle infinie

```
LOOP
   insctructions
END LOOP;
```

On termine cette boucle par `RETURN` ou `EXIT`. La forme générale de `EXIT`

```
EXIT [ etiquette ] [ WHEN expression-boolenne ] ;
```

`EXIT` déplace le contrôle après la boucle. On peut étiqueter les boucles par `<<etiquette>>` ce qui permet de terminer les boucles imbriquées par `EXIT <<etiquette>>;` Donc EXIT fait la même chose que `break` en C.

```
LOOP
     x:=x+1;
     IF x>10 THEN
          EXIT;
     END IF;
END LOOP;

LOOP
     x:=x+1;
     EXIT WHEN x>10;
END LOOP;
```

7 Boucles WHILE

```
WHILE condition LOOP
   instructions
END LOOP;
```

8 Boucles FOR

```
FOR I IN  1..10 LOOP
      I Prendra les valeurs 1,2,...,10
END LOOP;

FOR I IN REVERSE 10..9 LOOP
      I Prendra les valeurs 10,9,8,7,...,1
END LOOP;

FOR I IN REVERSE 10..1 BY 2 LOOP
    I prendra les valeurs 10,8,6,4,2
END LOOP;
```

Noter l'ordre de bornes inférieure et supérieure dans `REVERSE`, il est l'inverse à celui de PL/SQL d'ORACLE.

`CONTINUE` à l'intérieur de la boucle aura le même effet qu'en langage C, on retourne au début de la boucle. La forme générale :

```
CONTINUE [ etiquette ] [ WHEN expression-booleenne ];
```

```
---------------------------------- new_factorial.sql -------------------------
CREATE OR REPLACE FUNCTION new_factorial(i INTEGER) RETURNS INTEGER AS $$
   DECLARE
     k INTEGER := 1;
   BEGIN
     -- IF
     IF i <= 0 THEN
       RETURN 0;
     END IF;
     -- Boucle for indexee par variable INTEGER.
     -- La variable qui indexe la boucle (L) est de type INTEGER.
     -- Inutile de la declarer (meme il ne faut pas la declarer).
     -- S'il y a deja une variable L declaree dans le bloc externe
     -- cette anciene variable sera invisible a l'interieur de la boucle.
     FOR L IN 1..i LOOP
       k := k*l;
     END LOOP;

     RETURN k;
   END
$$ LANGUAGE 'plpgsql';
--------------------------------------------------------------------
```

9 SELECT ... INTO et la variable FOUND

```
SELECT liste-select INTO destination FROM ...;
```

La requête ne doit pas retourner plus qu'une ligne. Avec `FOUND` on pourra vérifier si la requête a effectivement retournée une ligne ou non.

```
--------------- into.sql -------------------------------------
CREATE OR REPLACE FUNCTION intot(ingredients.ingredientid%TYPE)
       RETURNS ingredients.unit%TYPE AS $$
   DECLARE
     a ingredients%ROWTYPE; --Declaration d'une variable du type:
                            --ligne de la table ingredients
   BEGIN
     -- Maintenant on peut faire SELECT dans a:
     SELECT * INTO a
     FROM ingredients
```

```
     WHERE $1 = ingredients.ingredientid ;

     IF FOUND THEN --FOUND est TRUE si le select
                   --a trouve des lignes
        RETURN a.unit;
     ELSE
        RETURN 'unknown' ;
     END IF;
   END;
 $$ LANGUAGE 'plpgsql';
-----------------------------------------------------------------
```

Destination peut-être une liste de variables :

```
---------------------- intob.sql --------------------------------
CREATE OR REPLACE FUNCTION intot(ingredients.ingredientid%TYPE) RETURNS TEXT AS $$
   DECLARE
     a ingredients.unit%TYPE;
     b ingredients.name%TYPE;
   BEGIN

     SELECT ingredients.unit, ingredients.name
     INTO   a,b
     FROM ingredients
     WHERE $1 = ingredients.ingredientid ;

     IF FOUND THEN
        RETURN a||' '||b ;
     ELSE
        RETURN 'unknown' ;
     END IF;
   END;
 $$ LANGUAGE 'plpgsql';

-- select intot('CHESE');
```

10 Boucle FOR sur les résultats de SELECT.

Une boucle for pour le parcours des résultats d'une requête `SELECT`. La variable `iterator` doit être soit de type `RECORD` soit `%ROWTYPE`

```
[<<label>>]
  FOR iterator IN
       requete-SELECT
  LOOP
       instructions
  END LOOP;
```

```
------------------------------ forin.sql ----------------------------------------
CREATE OR REPLACE FUNCTION forin(a ingredients.unit%TYPE) RETURNS DECIMAL(5,3) AS $$
   DECLARE
     toto RECORD;  -- noter une variable de type RECORD
     prix ingredients.unitprice%TYPE;
     s REAL DEFAULT 0;  -- initialisation avec DEFAULT
     i INTEGER := 0 ;   -- initialisation avec :=
   BEGIN

     FOR toto IN
       SELECT name, unitprice FROM ingredients WHERE unit = a
     LOOP
       s := s + toto.unitprice;
       i := i + 1 ;
```

```
     END LOOP;
     IF i = 0 THEN
        RETURN 0;
     END IF;
     RETURN (s/i) :: DECIMAL(5,3) ; --noter un cast

   END;
 $$ LANGUAGE 'plpgsql';

-- SELECT forin('slice');
-- retourne le prix moyen par unite de 'slice' (tranche)
```

A la place d'une requête `SELECT` on peut mettre `INSERT, UPDATE, DELETE` avec une clause `RETURNING`.

11 Paramètres OUT

Certains paramètres peuvent être déclarés `OUT`, c'est-à-dire comme de paramètres de sortie. La dernière valeur affectée sur un paramètres `OUT` sert de la valeur de sortie.

```
-----------------------------------sum_n_product-----------------------
CREATE OR REPLACE  FUNCTION
sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
-------------------------------------------------------------------------
```

La fonction ci-dessus sera appelée comme : `sum_n_roduct(5,7)` Si on utilise `RETURN` dans cette fonction elle devra retourner `RECORD`.

12 Fonctions avec les types polymorphes

Les paramètres d'une fonction peuvent être déclarés en utilisant les types polymorphes : `anyelement, anyarray, anynonarray, anyenum`. Nous n'utiliserons que `anyelement`. Dans la fonction

```
---------------------------- add_three_values--------------------------
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
-------------------------------------------------------------------------
```

`anyelement` désigne le type quelconque (à condition que l'addition soit supportée par ce type) mais tous les trois arguments et la valeur retournée doivent être du même type (c'est-à-dire quelconque mais le même).

Une fonction dont le type de retour est polymorphe possède une variable (le paramètre) `$0`, initialisée NULL, qui permet de stocker la valeur à retourner. Sinon `$0` est utilisé comme tous les autres `$i`.

13 Fonctions avec des types composés

```
------------------------------ create emp ----------------------------------------
CREATE TABLE emp (
    name        text,
    salary      numeric,
    age         integer,
    cubicle     point
);
----------------------------------------------------------------------------------

--------------------------------------- double_salary ------------------------------------------
CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;
------------------------------------------------------------------------------------------------
```

Dans la fonction ci-dessus le paramètre `emp` de la fonction signifie que la fonction prend en paramètre une ligne (un record) avec les même types qu'une ligne de la table `emp`. La fonction est ensuite utilisée comme

```
SELECT name, double_salary(emp.*) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

Si une fonction retourne un type composé alors pour accéder aux champs on utilise une de deux syntaxes :

```
SELECT (new_emp()).name;
SELECT name(new_emp());
```

La notation fonctionnelle peut être aussi utilisée avec les noms d'attributs d'une table : `toto.prix` c'est la même chose que `prix(toto)` pour une table `toto`.

Si une fonction retourne un type composé elle peut être utilisée partout où on utilise de tables, par exemple dans `FROM` (ceci est vrai aussi pour les valeurs scalaires mais dans ce cas peu utile).

14 Les fonction retournant des tables

14.1 RETURNS SETOF

Une telle fonction déclarée avec
`RETURNS SETOF sometype`
Elle peut-être utilisée dans FROM à la place d'une table.

```
---------------------------------- commander ---------------------------------------
-- le parametre ingredients ci-dessous indique juste que $1 devait être du même type
-- que la table ingredients
CREATE OR REPLACE FUNCTION commander() RETURNS SETOF ingredients AS $$
BEGIN
 -- le select suivant peut retourner plusieurs lignes.
 -- Sans SETOF le select retourne une seule ligne.
  RETURN QUERY SELECT *  FROM ingredients WHERE inventory = 0 OR INVENTORY IS NULL;
END ;
$$ LANGUAGE plpgsql;
------------------------------------------------------------------------------------

---------------------------- modifier_prix-----------------------------------------------------
DROP FUNCTION modifier_prix(NUMERIC(10,2)) ;
CREATE OR REPLACE FUNCTION modifier_prix(NUMERIC(10,2), OUT non items.name%TYPE, OUT prix items.price%TYPE)
    RETURNS SETOF record  AS $$
    BEGIN
    RETURN QUERY SELECT name, price * $1 FROM items WHERE price IS NOT NULL;
     END
$$ LANGUAGE plpgsql;
-----------------------------------------------------------------------------------------------
```

14.2 RETURNS TABLE

Cette version est conforme avec le standard SQL (mais implémentée seulement à partir de PostgreSQL 8.4). Pas de paramètres `IN OUT` explicites dans ce cas, tous les paramètres `OUT` sont dans `TABLE(param)`.

```
-------------------------- modifier_prix-------------------------------
DROP FUNCTION modifier_prix(NUMERIC(10,2)) ;

CREATE OR REPLACE FUNCTION modifier_prix(NUMERIC(10,2))
    RETURNS TABLE(nom items.name%TYPE, prix items.price%TYPE)
    AS $$
    BEGIN
    RETURN QUERY SELECT name, price * $1 FROM items ;
    END ;
$$ LANGUAGE plpgsql;
-----------------------------------------------------------------------------
```

Noter `RETURN QUERY` dans la fonction pour retourner une table.

RETURN QUERY ne termine pas la fonction mais ajoute des lignes dans la table que est construite. RETURN sans paramètre termine la fonction et retourne la table construite de cette façon. Notez aussi `RETURN NEXT` qui ajoute juste une ligne dans la table construite par la fonction.

15 Exécuter le code SQL à l'intérieur d'une fonction

Vous pouvez mettre le code SQL directement dans les fonctions PL/pgSQL : `CREATE TABLE`, `INSERT`, `UPDATE`, `DELETE`, etc. sauf `SELECT` et sauf l'appel à une autre fonction si le résultat de SELECT ou de l'appel n'est pas récupéré.

Si on veut exécuter `SELECT` sans récupération de résultat on fera

```
PERFORM requete-select ;
```

comme dans

```
PERFORM * FROM ingredients WHERE unitprice < 5;
```

Noter que `PERFORM` remplace le mot `SELECT`.

`PERFORM` peut être utilisé pour évaluer une expression SQL sans récupération de résultat, par exemple

```
PERFORM fonction(parametres);
```

Après `PERFORM` on pourra tester la variable `FOUND` pour voir si la requête a produit un résultat ou si elle a retourné NULL. Si on utilise PERFORM c'est pour les effets de bord de la requête et non pas pour la valeur retournée (la valeur on l'ignore sciemment).

```
------------------------------ ajouter_meal()-------------------------------------------------
CREATE OR REPLACE FUNCTION ajouter_meal( meals.mealid%TYPE, meals.name%TYPE)
       RETURNS INTEGER AS $$
       DECLARE
         id ALIAS FOR $1;
         nom ALIAS FOR $2;
         a INTEGER;
       BEGIN
       -- si a place de l'instruction suivante on it
       -- SELECT * FROM meals WHERE mealid=id;
       -- cela compile OK mais probl 'extion.
         a := add(5,6);
         PERFORM add(7,8);
         PERFORM  * FROM meals WHERE mealid=id;
```

```
        IF  FOUND THEN
           RETURN 0;
        END IF;
        INSERT INTO meals VALUES(id,nom);
        RETURN 1;
      END;
      $$ LANGUAGE 'plpgsql';
-------------------------------------------------------------------------------------------------
```

16 EXECUTE pour exécuter les requêtes dynamiques.

```
EXECUTE string;
```

permet d'exécuter une requête contenue dans une chaîne de caractères. Cela permet de faire de requêtes dynamiques qui ne sont pas precompilées.

```
------------------- exect.sql --------------------------------
CREATE OR REPLACE FUNCTION exect(commande VARCHAR) RETURNS INTERVAL AS $$
   DECLARE
     beg_time TIMESTAMP;
     end_time TIMESTAMP;
   BEGIN
     beg_time := current_timestamp;
     EXECUTE commande;
     end_time := current_timestamp;
     RETURN end_time - beg_time;
   END
 $$ LANGUAGE 'plpgsql';

--SELECT exect('SELECT * FROM ingredients');
-----------------------------------------------------------------------

---------------------------------- modif_prix ----------------------------------------
CREATE OR REPLACE FUNCTION modif_prix(VARCHAR(30), DEC(5,2), DEC(6,2))
          RETURNS SETOF ingredients AS $$
DECLARE
  qui ALIAS FOR $1;
  plus ALIAS FOR $2;
  pcent ALIAS FOR $3;
  val ingredients.unitprice%TYPE;
  tal ingredients.unitprice%TYPE;
  r RECORD;
  u RECORD;
BEGIN
  FOR r IN
     EXECUTE 'SELECT * FROM ingredients WHERE '|| qui
  LOOP
     val := r.unitprice * ( 100 + pcent )/100 ;
     tal := r.unitprice + plus;
     IF val > tal THEN
        u := ROW( r.ingredientid, r.name, r.unit, val, r.foodgroup, r.inventory, r.vendorid )::ingredients;
        RETURN NEXT u;
     ELSE
        u := ROW( r.ingredientid, r.name, r.unit, tal, r.foodgroup, r.inventory, r.vendorid )::ingredients;
        RETURN NEXT u;
     END IF;
   END LOOP;
   RETURN;
 END; $$
 LANGUAGE plpgsql;
 ------------------------------------------------------------------------------------------------
```

```
---------------------- test_modif_prix----------------
SELECT I.name, I.unitprice, AUG.unitprice
FROM ingredients I JOIN
     modif_prix('foodgroup=''Milk''',
                 0.05, 5) AS AUG USING(ingredientid)  ;
-----------------------------------------------------
```

Forme générale de `EXECUTE` :

```
EXECUTE command-string [ INTO [ STRICT ] target ] [ USING expression, ...] ] ;
```

`command-string` de type `text`. `target` est une variable de type `RECORD`, ou une variable de type `ROWTYPE` ou une liste de variables simples. Le résultat de la requête sera stocké dans ces variables. Si la requête retourne plusieurs lignes alors c'est la première ligne qui sera stockée dans le(s) variable(s) `INTO` . Avec `STRICT` il aura une erreur si la requête retourne plusieurs lignes. `command-string` peur avoir des paramètres, `$1, $2, ...` qui seront substitués par les valeurs calculées dans `USING`. Cela ne concerne que les paramètres, `$1,...` ne concerne pas les noms de tables qui doivent être injectés comme les chaînes de caractères.

On peut utiliser les fonction `quote_lireal` `quote_ident` `quote_nullable`.

`quote_indent(text)` retourne une chaîne de caractères avec des ' pour l'utilisation comme identificateur.*

`quote_literal(text)` pour que ' approprié comme string-literal. Elle retourne NULL si l'argument NULL. Si on veut NULL pour l'argument NULL il faut utiliser `quote_nullable(text)`.

17 Requête dynamiques dans une boucle FOR.

```
[<<label>>]
  FOR iterator IN
    EXECUTE string-requete [ USING expression,... ]
  LOOP
    instructions
  END LOOP;
```

`string-requete` s'évalue comme une chaîne de caractères contenant une requête `SELECT`

```
---------------------------- somme2.sql ---------------------------------------
CREATE OR REPLACE FUNCTION somme2(req VARCHAR) RETURNS REAL AS $$
   DECLARE
     toto REAL := 0;  -- noter une variable de type RECORD
     s REAL DEFAULT 0;  -- initialisation par DEFAULT
   BEGIN

     FOR toto IN
       EXECUTE req
     LOOP
       s := s + toto;
     END LOOP;

     RETURN s;

   END;
 $$ LANGUAGE 'plpgsql';

--SELECT somme2('SELECT unitprice FROM ingredients');
-------------------------------------------------------------------------------

---------------------------- somme_square.sql ---------------------------------------
CREATE OR REPLACE FUNCTION somme_square(tables text, col text) RETURNS REAL AS $$
   DECLARE
     toto REAL := 0;  -- noter une variable de type RECORD
     s REAL DEFAULT 0;  -- initialisation par DEFAULT
```

```
   BEGIN

     FOR toto IN
       EXECUTE 'SELECT ' || quote_ident(col) || ' FROM '|| quote_ident(tables)
     LOOP
       s := s + toto*toto;
     END LOOP;

     RETURN sqrt(s);

   END;
 $$ LANGUAGE 'plpgsql';

select somme_square('ingredients','unitprice');
```

18 Utilisation de curseur.

Pour travailler avec le curseur : `DECLARE, OPEN, FETCH, CLOSE`

18.1 Curseur liés à une requête

Déclaration dans la partie `DECLARE` d'un bloc :

```
nom_cursor CURSOR FOR requete_select ;
nom_cursor [ [NO] SCROLL] CURSOR(parametres_formelles) FOR requete_select ;
```

déclarent des curseur liés. `SCROLL` indique qu'on pourra faire un parcours en arrière sur le résultat. Paramètres formelles est une liste de couples `nom type_de_donnee`.

Exemples :

```
DECLARE
    curs1 refcursor;  --curseur non lie
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) IS SELECT * FROM tenk1 WHERE unique1 = key;
```

Toutes les trois variables sont de type `refcursor` mais les deux dernières sont liées à des requêtes. De plus `curs3` possède un paramètre `key` de type `INTEGER`.

18.2 FETCH

Pour récupérer les valeurs faire dans une boucle :

```
FETCH [ direction {FROM|IN} ] nom_cursor INTO destinations(s) ;
```

où `destination(s)` une variable `RECORD` ou une variable `%ROWTYPE` ou une liste de variables. Ici un exemple qu'on aurait pu faire avec `FOR`. Noter `EXIT WHEN NOT FOUND;` qui permet de tester si le parcours est fini. Si rien à lire `destination` prendra la valeur `NULL`.

`direction`, une de directions parmi : `NEXT, PRIOR, FIRST`, `LAST, ABSOLUTE count`, `RELATIVE count`, `FORWARD,` or `BACKWARD`;

`FETCH` sans direction est équivalent à `FETCH NEXT`

18.3 MOVE

```
MOVE [ direction { FROM | IN } ] cursor;
```

Fait la même chose que `FETCH` mais ne retourne rien.

18.4 UDATE/DELETE

```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
```

les opérations sont effectuées sur la ligne pointée par le curseur.

18.5 OPEN/CLOSE

Pour ouvrir un curseur lié avec les paramètres :

```
OPEN nom_cursor(parametres_actuels);
```

Pour fermer un curseur non utilisé :

```
CLOSE cursor;
```

```
------------------------------------ curs.sql ------------------------------------
CREATE OR REPLACE FUNCTION
       curs(ingredients.unitprice%TYPE, ingredients.unitprice%TYPE)
       RETURNS REAL AS $$
   DECLARE
     -- declaration d'un cursor
     toto CURSOR FOR
          SELECT * FROM ingredients WHERE unitprice BETWEEN $1 AND $2;
     resultat RECORD;
     s REAL DEFAULT 0;  -- initialisation par DEFAULT

   BEGIN

     OPEN toto; -- ouvrir le cursor
     LOOP
       FETCH toto INTO resultat; -- fetch une ligne dans un record
       EXIT WHEN NOT FOUND;        -- Plus rien a lire alors quitter la boucle
       s := s + resultat.unitprice * resultat.unitprice ;
     END LOOP;

     CLOSE toto; -- fermer le cursor
     RETURN s;

   END;
 $$ LANGUAGE 'plpgsql';

--SELECT curs(0.01,0.03);
--------------------------------------------------------------------------------
```

18.6 Afficher des messages

Pour afficher des messages on utilise :

```
RAISE NOTICE 'message' [,variable ..];
```

Le message peut contenir les % dont les occurrences sont substituées par les variables de la liste.

```
------------------------------ cursp.sql -----------------------------------------
CREATE OR REPLACE FUNCTION cursp()
       RETURNS VOID AS $$
   DECLARE
```

```
    -- declaration d'un cursor simple
    vendor CURSOR FOR
         SELECT * FROM vendors;

    -- declaration d'un cursor parametre
    ingr CURSOR(ID ingredients.vendorid%TYPE) FOR
         SELECT * FROM ingredients WHERE ID = ingredients.vendorid;

    resultat RECORD;
    val ingredients%ROWTYPE;

    I INTEGER;
  BEGIN

    OPEN vendor; -- ouvrir le cursor
    LOOP
      FETCH vendor INTO resultat; -- fetch une ligne de vendors dans un record
      EXIT WHEN NOT FOUND;        -- Plus rien a lire alors quitter la boucle

      I := 0;
      OPEN ingr(resultat.vendorid) ;

      LOOP   --boucle interne
         FETCH ingr INTO val; --fetch une ligne
         EXIT WHEN NOT FOUND;
         I := I+1;
      END LOOP;

      CLOSE ingr; --fermer cursor

      IF I > 2 THEN
         RAISE NOTICE 'bon vendor % ',  resultat.repfname;
      ELSE
         RAISE NOTICE 'mauvais vendor %', resultat.repfname;
      END IF;

    END LOOP; --fin boucle externe

    CLOSE vendor; -- fermer le cursor
    RETURN ;

  END;
 $$ LANGUAGE 'plpgsql';

---------------  utilisation:
--SELECT cursp();
-------------------------------------------------------------------------------
```

18.7 Retourner le curseur

Si on retourne le cours ce curseur peut être parcouru par l'appelant. Il faut d'abord associer un nom avec le curseur avant OPEN. Pour cela on fait une affectation d'une chaîne de caractères sur la variable curseur. Cette chaîne de caractère c'est le nom de curseur. (Mais ce nom peut être généré automatiquement, donc il n'est pas nécessaire d'associer un nom au curseur.)

Exemple suivant montre comment l'appelant passe le nom de curseur vers la fonction, c'est le plus simple puisque autrement l'appelant devrait trouver le nom de curseur autrement.

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
```

```
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

18.8 Boucler sur le résultats d'un curseur

```
[ <<label>> ]
FOR recordvar IN bound_cursorvar [ ( argument_values ) ] LOOP
    statements
END LOOP [ label ];
```

Cela concerne uniquement le curseurs liés, le curseur **ne peut pas** être ouvert à ce moment. La variable `recordvar` est automatiquement déclarée comme `record` et existe seulement dans la boucle.

18.9 Curseur non liés

On peut déclarer un curseur qui ne sera pas associé à une requête. Le curseur de ce type est déclaré comme une variable de type `refcursor` :

```
mon_cursor refcursor;
```

Si nous avons un curseur non lié alors au moment de faire `OPEN` on spécifie la requête `SELECT` :

```
OPEN mon_cursor_non_lie [[NO] SCROLL] FOR requête;
```

pour une requête fixe ou

```
OPEN mon_cursor [[ NO ] SCROLL ] FOR EXECUTE string-requête;
```

pour une requête dynamique.

Ensuite nous pouvons utiliser le curseur pour parcourir les résultats de la requête.

L'exemple précédent réécrit pour utiliser les curseurs non liés :

```
------------------------------ curspd.sql -------------------------------------------
CREATE OR REPLACE FUNCTION curspd()
       RETURNS VOID AS $$
   DECLARE
     -- declaration des cursor  non lies
     vendor refcursor;
     ingr refcursor;

     resultat RECORD;
     val ingredients%ROWTYPE;
     I INTEGER;
   BEGIN

     OPEN vendor FOR SELECT * FROM vendors; -- ouvrir le cursor
     LOOP
       FETCH vendor INTO resultat; -- fetch une ligne de vendors dans un record
       EXIT WHEN NOT FOUND;        -- Plus rien a lire alors quitter la boucle
```

```
       I := 0;
       OPEN ingr FOR
            SELECT * FROM ingredients
            WHERE resultat.vendorid = ingredients.vendorid;

       LOOP   --boucle interne
          FETCH ingr INTO val; --fetch une ligne
          EXIT WHEN NOT FOUND;
          I := I+1;
       END LOOP;

       CLOSE ingr; --fermer cursor

       IF I > 2 THEN
          RAISE NOTICE 'bon vendor % ',  resultat.repfname;
       ELSE
          RAISE NOTICE 'mauvais vendor %', resultat.repfname;
       END IF;

     END LOOP; --fin boucle externe

     CLOSE vendor; -- fermer le cursor
     RETURN ;

   END;
 $$ LANGUAGE 'plpgsql';

---------------  utilisation:
--SELECT curspd();
-------------------------------------------------------------------------------
```

19 Exceptions

Si il y une exception pendant l'exécution d'une fonction alors PLpgSQL fait automatiquement `ROLLBACK` jusqu'au le début du bloc `BEGIN`.