PL/SQL : Procedural Language for SQL

Oracle : PL/SQL, PosgreSQL : PL/pgSQL, SQL Server : Transact-SQL, DB2 : SQLPL
PL/* : Langages syntaxiquement proche de Pascal et Ada

Structure de PL/pgsql

PL/pgSQL est structuré en blocs :

```
[ <<label>>]
[ DECLARE
   déclarations ]
BEGIN
   instructions
END [label];
```

Chaque déclaration et chaque instruction se terminent par un point-virgule `;`.
Les commentaires sont initiés par `--` ou délimités par `/*` et `*/`.

Déclarations

Syntaxe : **`nom [CONSTANT] type [NOT NULL][ {DEFAULT | := } expression];`**

- Exemples :

```
quantite integer DEFAULT 42;
url varchar := 'http://www.univ-lehavre.fr';
age_capitaine CONSTANT integer := 42;
uneLigne nomTable%ROWTYPE;
uneAutreLigne RECORD;
unChamp nomTable.nomColonne%TYPE;
```

Tous les types SQL sont utilisables.
Les records sont des structures composites comme en Pascal similaires au `struct` du C.

Manipulation de tableaux

PostgreSQL accepte les tableaux comme type de données (y compris dans les tables)

- **Déclaration :**

```
nomVar1 typeSimple[]
nomVar2 typeSimple[][]
nomVar3 typeSimple ARRAY[longueur]
```

- **Expression littérale :** `'{val1, val2, ...}'`
- **Accesseurs :** `nomTableau[indice]`

Types composites

Un type composite correspond à la structure d'un enregistrement associant le nom des champs à leur type.
Lors de la création d'une nouvelle table un type composite correspondant est systématiquement associé

- **Déclaration :** (similaire à `CREATE TABLE`)

```
CREATE TYPE personne AS (
prenom VARCHAR[30],
nom VARCHAR[30]);
```

- **Usage dans une table :**

```
CREATE TABLE auteur (
  auteur_id SERIAL,
  auteur_nom_prenom personne);

INSERT INTO auteur VALUES (DEFAULT, ROW('Victor','Hugo');
```

- **Accesseurs :** `nomVariableComposite.nomChamp`

Fonctions et paramètres

```
CREATE [OR REPLACE] FUNCTION nomFonc([[nom] type]*) RETURNS (type|void) AS $$
...
$$ LANGUAGE plpgsql;
```

$$ est le délimiteur de début et fin de définition de fonction.
Chaque paramètre anonyme peut être identifié par $n où n est le numéro d'ordre du paramêtre.
Avant la version 8.0 il fallait explicitement déclarer l'alias : `nom ALIAS FOR $n;`

- Exemple :

```
CREATE FUNCTION prixTTC(prixHT real, real) RETURNS real AS $$
DECLARE
   taux ALIAS FOR $2;
BEGIN
   RETURN prixHT * (1+taux);
END;
$$ LANGUAGE plpgsql;
```

Instructions simples

Ne rien faire : **`NULL;`**
Assignation : **`var := val;`**
Concaténer deux chaînes de caractères : **`'SELECT *' || ' FROM table';`**
Récupération du résultat d'une requête ne retournant qu'une seule ligne :
`SELECT select_expr INTO varCible FROM ...;`
où varCible peut être un enregistrement (RECORD), une variable ligne ou une

liste de variables séparées par des virgules. La variable spéciale **FOUND** (booléenne) est associé au résultat de la requête.
Exécution d'une fonction : **SELECT fct(arg$_1$, ...);**

Exécution d'une expression ou d'une requête sans résultat :
PERFORM query;
Exécution dynamique de commandes :
EXECUTE commande [INTO cible];
où commande est une expression interprétable.
Récupérer l'état du résultat :

> **GET DIAGNOSTICS variable = item [, ...] avec item ∈ {ROW_COUNT, RESULT_OID}**

Structures de contrôle simples

RETURN expression;
RETURN NEXT expression; : accumule un résultat local dans une mémoire tampon destinée au résultat complet
RETURN QUERY requête; : même principe pour le résultat d'une requête
IF expression THEN ... END IF;
IF expression THEN ... ELSE ... END IF;
IF expression THEN ... ELSEIF ... END IF;
IF expression THEN ... ELSIF ... END IF;

```
CREATE FUNCTION prixTTC(prixHT real, taux real) RETURNS real AS $$
BEGIN
   IF taux < 0 THEN
     RAISE EXCEPTION 'taux négatif %', taux;
   ELSIF taux <= 1 THEN
     RETURN prixHT * (1+taux);
   ELSE
     RETURN prixHT * (taux);
   END IF;
END;
$$ LANGUAGE plpgsql;
```

CASE [...] WHEN ... THEN ... [WHEN ... THEN ...] ELSE ... END CASE;

Boucles

LOOP ... END LOOP;
EXIT [WHEN expression];
CONTINUE [WHEN expression];

- Exemple :

```
DECLARE
  n integer := 0;
BEGIN
  LOOP
    n := n+1;
    CONTINUE WHEN (n<100);
    EXIT WHEN (n=100);
  END LOOP;
END;
```

Boucles

WHILE

```
WHILE n<100 LOOP
  n := n+1;
END LOOP;
```

FOR var in [REVERSE] debut .. fin [BY increment] LOOP

```
FOR n in 1 .. 100 LOOP

END LOOP;
```

Gestion des erreurs

- Syntaxe :

```
BEGIN
   instructions
EXCEPTION
   WHEN condition [OR condition]* THEN
      instructions
   [ WHEN condition [OR condition]* THEN
      instructions ]*
END;
```

Permet d'éviter les interruptions d'exécution.
Coût d'exécution élevé !
Quelques codes d'erreurs : DIVISION_BY_ZERO, PRIVILEGE_NOT_GRANTED, INTEGRITY_CONSTRAINT_VIOLATION, ...

Messages et erreurs

- L'instruction **RAISE** permet l'envoie de différents niveaux de message au client ou au système.
- Syntaxe : **RAISE niveau 'format' [, expr [, ...]];**
- niveau ∈ {DEBUG, LOG, INFO, NOTICE, WARNING, EXCEPTION}
- format est une chaîne où % est remplacé par l'expression correspondante.
- Exemple

```
RAISE NOTICE 'taux négatif %', taux;
```

- La génération d'une exception interrompt la transaction en cours.

Les Curseurs

Structure du langage qui permet de s'affranchir des ensembles pour manipuler les résultats de requêtes ligne par ligne.
Fonctionne comme une tête de lecture sur les résultats de requêtes.

Déclaration :

```
DECLARE
   curseur1 refcursor;
curseur2 CURSOR FOR SELECT ...;
curseur3 CURSOR(key integer) IS SELECT ... WHERE id=key;
```

Les curseurs associés à une requête sont qualifiés de *curseur lié (bound_cursor)*

Ouvrir un curseur non lié : **`OPEN unbound_cursor FOR query;`**
Ouvrir un curseur lié : **`OPEN bound_cursor [(arg)];`**
Manipuler une ligne de résultat : **`FETCH cursor INTO cible;`**. Avance le curseur sur la ligne suivante et indique via `FOUND` si elle existe.
Fermer un curseur et libérer les ressources associées : **`CLOSE cursor;`**

```
CREATE OR REPLACE FUNCTION fct() RETURNS void AS $$
DECLARE
    rec RECORD;
    curs refcursor;
BEGIN
    EXECUTE 'CREATE TABLE temp (idliv integer, titre varchar(50))';
    OPEN curs FOR SELECT liv_num, liv_titre FROM livre;
    LOOP
        FETCH curs INTO rec;
        EXIT WHEN NOT FOUND;
        INSERT INTO temp VALUES(rec.liv_num,rec.liv_titre);
    END LOOP;
    CLOSE curs;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE function fct() RETURNS void AS $$
DECLARE
    rec temp%ROWTYPE;
BEGIN
    execute 'DELETE FROM TEMP';
    FOR rec IN SELECT liv_num, liv_titre from livre LOOP
       INSERT INTO temp VALUES(rec.liv_num,rec.liv_titre);
    END LOOP;
```

```
END;
$$ LANGUAGE plpgsql;
```

Fonctions tables (en SQL)

Il est possible de définir des fonctions dont le résultat correspond à un ensemble de lignes provenant d'une table ou d'une vue.
Le type de retour de ces fonctions est `SETOF nomTable`.

- **Exemple :**

```
create FUNCTION selectTitres(varchar) RETURNS SETOF livre AS $$
    SELECT * FROM livre WHERE liv_titre like $1;
$$ LANGUAGE SQL;

SELECT * FROM selectTitres('Les %') AS t1;
```

Elles s'utilisent ensuite comme des tables ou des vues.
Il est également possible de retourner une table créée dynamiquement avec le type de retour `TABLE(`*`cols`*`)`

- **Exemple :**

```
CREATE OR REPLACE FUNCTION selectTitres(debut int, fin int) RETURNS
TABLE(numero int, titre VARCHAR) AS $$
    SELECT liv_num, liv_titre FROM livre WHERE liv_num BETWEEN $1 AND $2;
$$ LANGUAGE SQL;

SELECT * FROM selectTitres(1,5) AS t1;
```

Fonctions tables (en PL/PGSQL)

Le type de retour de ces fonctions est aussi **`SETOF nomTable`**.
Il faut utiliser un curseur pour parcourir toutes les lignes d'un résultat de **`SELECT`**.
Chaque ligne est renvoyée par l'instruction **`RETURN NEXT`**
Une instruction **`RETURN`** clos la fonction.

```
create FUNCTION selectTitres(varchar) RETURNS SETOF livre AS $$
DECLARE
  ligne livre%ROWTYPE;
BEGIN
    FOR ligne IN SELECT * FROM livre WHERE liv_titre like $1 LOOP
        RETURN NEXT ligne;
    END LOOP;
    RETURN;
END;
$$ LANGUAGE plpgsql;

SELECT * FROM selectTitres('Les %') AS t1;
```

Procédures déclenchées (Triggers)

Un trigger est une fonction qui doit être éxécutée lorsque certaines opérations sont réalisées.
Ces déclenchements peuvent avoir lieu avant ou après un **INSERT**, un **UPDATE** ou un **DELETE**.
Les opération automatiques peuvent être réalisées une fois par instruction ou une fois par ligne concernée.
Ce sont des fonctions particulières sans paramètre et de type `trigger`
Lors de l'exécution d'un trigger, des variables spéciales sont créées automatiquement : **NEW**, **OLD**, ...

- La mise place une procédure déclenchée se fait en deux temps :
    1. définition d'une fonction réalisant les opérations réalisées automatiquement lors de l'observation des événements déclencheurs ;
    2. définition du déclencheur lui-même avec **CREATE TRIGGER ...**

Paramètres automatiques d'un trigger

- **NEW** : record correspondant à une ligne en insertion ou modification.
- **OLD** : record correspondant à une ligne en destruction ou modification.
- **TG_NAME** : nom du trigger en cours d'exécution.
- **TG_WHEN** : **BEFORE** ou **AFTER**
- **TG_LEVEL** : **ROW** ou **STATEMENT**
- **TG_OP** : **INSERT**, **UPDATE** ou **DELETE**
- **TG_RELID** : identifiant de l'objet ayant déclenché le trigger.
- **TG_TABLE_NAME** : nom de la table ayant provoqué le trigger.
- **TG_SCHEMA_NAME** : nom du schéma de la table ayant provoqué le trigger.
- **TG_NARGS** : nombre d'arguments
- **TG_ARGV[]** : tableau d'arguments

Création et modification des triggers

- Il faut au préalable avoir défini une fonction de type trigger.
- **Syntaxe :**

```
CREATE TRIGGER nomTrigger
   { BEFORE | AFTER }                        -- moment du déclenchement
   { event [ OR ... ] }                      -- événements concernés
   ON nomTable                               -- table concernée
   [ FOR [ EACH ] { ROW | STATEMENT } ]      -- modalité d'exécution des
opérations automatiques
   EXECUTE PROCEDURE nomFonction(arguments) -- appel de la fonction gérant
les opérations automatiques
```

- **Modification :** `ALTER TRIGGER name ON table RENAME TO newname`
- **Suppression :** `DROP TRIGGER name ON table [ CASCADE | RESTRICT ]`
- Ces commandes requièrent le privilège TRIGGER.