Université Paris XII, IUT de Fontainebleau
Département d'informatique

Introduction à Oracle PL/SQL - Guide

PL/SQL (pour PROCEDURAL LANGUAGE/SQL) est un langage procédural d'Oracle corporation étendant SQL. Il permet de combiner les avantages d'un langage de programmation classique avec les possibilités de manipulation de données offertes par SQL.

Ces notes se donnent pour but de présenter succintement PL/SQL. Il ne s'agit en aucun cas d'un guide complet du langage.

1 Structure d'un programme PL/SQL

La structure de base d'un programme PL/SQL est celle de bloc (possiblement imbriqué). Il a généralement la forme suivante:

```
DECLARE

/* section de declaration */

BEGIN

/* corps du bloc de programme
   Il s'agit de la seule zone
   dont la presence est obligatoire */

EXCEPTION

/* gestion des exceptions */

END;
```

Le corps du programme (entre le BEGIN et le END) contient des instructions PL/SQL (assignements, boucles, appel de procédure) ainsi que des instructions SQL. Il s'agit de la seule partie qui soit obligatoire. Les deux autres zones, dites zone de déclaration et zone de gestion des exceptions sont facultatives. Les seuls ordres SQL que l'on peut trouver dans un bloc PL/SQL sont: `SELECT, INSERT, UPDATE, DELETE`. Les autres types d'instructions (par exemple `CREATE, DROP, ALTER`) ne

peuvent se trouver qu'à l'extérieur d'un tel bloc. Chaque instruction se termine par un ";".

Le PL/SQL ne se soucie pas de la casse (majuscule vs. minuscule). On peut inclure des commentaires par `---` (en début de chaque ligne commentée) ou par `/* .... */` (pour délimiter des blocs de commentaires).

2 Variables et types

Un nom de variable en PL/SQL comporte au plus 30 caractères. Toutes les variables ont un type On trouve trois sortes de types de variables en PL/SQL. A savoir :

- un des types utilisés en SQL pour les colonnes de tables.
- un type particulier au PL/SQL.
- un type faisant référence à celui d'une (suite de) colonne(s) d'une table.

Les types autorisés dans PL/SQL sont nombreux. On retiendra principalement:

- Pour les types numériques : `REAL`, `INTEGER`, `NUMBER` (précision de 38 chiffres par défaut), `NUMBER(x)` (nombres avec `x` chiffres de précision), `NUMBER(x,y)` (nombres avec `x` chiffres de précision dont `y` après la virgule).

- Pour les types alphanumériques : `CHAR(x)` (chaine de caracère de longueur fixe `x`), `VARCHAR(x)` (chaine de caracère de longueur variable jusqu'à `x`), `VARCHAR2` (idem que précédent excepté que ce type supporte de plus longues chaines et que l'on est pas obligé de spécifier sa longueur maximale). PL/SQL permet aussi de manipuler des dates (type `DATE`) sous différents formats.

Une déclaration pourrait donc contenir les informations suivantes :

```
DECLARE
   n NUMBER;
   mot VARCHAR(20)
   mot2 VARCHAR2
```

Une autre spécificité du PL/SQL est qu'il permet d'assigner comme type à une variable celui d'un champs d'une table (par l'opérateur `%TYPE`) ou d'une ligne entière (opérateur `%ROWTYPE`). Dans la déclaration suivante :

```
DECLARE
   nom  emp.name%TYPE;
   employe emp%ROWTYPE;
```

la variable `nom` est définie comme étant du type de la colonne `name` de la table `emp` (qui doit exister au préalable). De même, `employe` est un vecteur du type d'une ligne de la table `emp`. A supposer que cette dernière ait trois champs `numero, name, age` de type respectifs `NUMBER, VARCHAR, INTEGER`, la variable `employe` disposera de trois composantes : `employe.numero`, `employe.name`, `employe.age`, de même types que ceux de la table.

Un premier petit progamme (noter au passage l'instruction d'affectation "`a:=a+b`") :

```
DECLARE
   a  NUMBER;
   b NUMBER;
BEGIN
   a:=a+b;
END;
```

Un deuxième petit programme (incluant une requête SQL) :

```
DECLARE
   a  NUMBER;
BEGIN
   SELECT numero INTO a
   FROM emp
   WHERE noemp='21';
END;
```

Ce dernier exemple donne comme valeur à la variable `a` le résultat de la requête (qui doit être du même type). Il est impératif que la requête ne renvoie qu'un et un seul résultat (c'est à dire qu'il n'y ait qu'un seul employé numéro 21). Autrement, une erreur se produit.

3 Opérateurs

PL/SQL supporte les opérateurs suivants :

- Arithmétique : `+, - *, /, **` (exponentielle)
- Concaténation : `||`
- Parenthèses (contrôle des priorités entre opérations): `()`

- Affectation: `:=`
- Comparaison : `=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN`
- Logique : `AND, OR, NOT`

4 Structures de contrôle

Comme n'importe quel langage procédural, PL/SQL possêde un certain nombre de structures de contrôles évoluées comme les branchements conditionnels et les boucles.

4.1 Les branchements conditionels

Syntaxe :

```
IF <condition> THEN
     commandes;
[ ELSEIF <condition> THEN
     commandes; ]
[ ELSE
     commandes; ]
END IF;
```

Un petit exemple :

```
IF nomEmploye='TOTO' THEN
     salaire:=salaire*2;
ELSEIF salaire>10000 THEN
     salaire:=salaire/2;
ELSE
     salaire:=salaire*3;
END IF;
```

4.2 boucles

PL/SQL admet trois sortes de boucles. La première est une boucle potentiellement infinie :

```
LOOP
   commandes;
END LOOP;
```

Au moins une des instructions du corps de la boucle doit être une instruction de sortie :

```
EXIT WHEN <condition>;
```

Dès que la condition devient vraie (si elle le devient...), on sort de la boucle.

Le deuxième type de boucle permet de répéter un nombre défini de fois un même traitement :

```
FOR <compteur> IN [REVERSE] <limite_inf> .. <limite_sup>
   commandes;
END LOOP;
```

Enfin, le troisième type de boucle permet la sortie selon une condition prédéfinie.

```
WHILE <condition> LOOP
   commandes;
END LOOP;
```

Toutes ces structures de contrôles sont évidemment imbriquables les unes dans les autres. Voici un même exemple traité de trois manières différentes suivant le type de boucle choisi.

```
DECLARE
   x NUMBER(3):=1;
BEGIN
  LOOP
    INSERT INTO employe(noemp, nomemp, job, nodept)
           VALUES (x, 'TOTO', 'PROGRAMMEUR', 1);
    x:=x+1;
    EXIT WHEN x>=100
  END LOOP;
END;
```

Deuxième exemple :

```
DECLARE
   x NUMBER(3);
BEGIN
  FOR x IN 1..100
    INSERT INTO employe(noemp, nomemp, job, nodept)
           VALUES (x, 'TOTO', 'PROGRAMMEUR', 1);
  END LOOP;
END;
```

Troisième exemple :

```
DECLARE
   x NUMBER(3):=1;
BEGIN
  WHILE x<=100 LOOP
    INSERT INTO employe(noemp, nomemp, job, nodept)
           VALUES (x, 'TOTO', 'PROGRAMMEUR', 1);
    x:=x+1;
  END LOOP;
END;
```

5 Curseurs

Jusqu'à présent, nous avons vu comment récupérer le résultat de requêtes SQL dans des variables PL/SQL lorsque ce résultat ne comporte au plus qu'une seule ligne. Bien evidemment, cette situation est loin de représenter le cas général : les requêtes renvoient très souvent un nombre important et non prévisible de lignes. Dans la plupart des langages procéduraux au dessus de SQL, on introduit une notion de "curseur" pour récupérer (et exploiter) les résultats de requêtes.

Un *curseur* est une variable dynamique qui prend pour valeur le résultat d'une requête. La méthode pour utiliser un curseur est invariable. En premier lieu, celui-ci doit être défini (dans la zone de declaration). Exemple :

```
CURSOR empCur IS SELECT * FROM emp;
```

Le curseur de nom `empCur` est chargé dans cet exemple de récupérer le résultat de la requête qui suit. Il peut alors être ouvert lorsqu'on souhaite l'utiliser (dans le corps d'un bloc) :

```
OPEN empCur;
```

Pour récupérer les tuples successifs de la requête, on utilise l'instruction :

```
FETCH empCur INTO employeVar;
```

La première fois, c'est le premier tuple du résultat de la requête qui est affecté comme valeur à la variable `employeVar` (qui devra être définie de façon adéquate). À chaque execution de l'instruction, un nouveau tuple résultat est chargé dans `employeVar`. Enfin, lorsque le traitement sur le résultat de la requête est terminé, on ferme le curseur.

```
CLOSE empCur;
```

Le petit exemple suivant sélectionne l'ensemble des employés dont le salaire ne dépasse pas 6000 francs et les augmente de 500 francs.

```
DECLARE
  CURSOR SalCur IS
       SELECT * FROM EMP WHERE SAL<6000.00;
  employe   EMP%ROWTYPE;
BEGIN
  OPEN SalCur;
  LOOP
        FETCH SalCur INTO employe;
        UPDATE EMP
        SET SAL=6500.00  WHERE EMPNO=employe.empno;
        EXIT WHEN SalCur%NOTFOUND;
  END LOOP;
END;
```

Lorsqu'on souhaite parcourir un curseur dans une boucle pour effectuer un traitement, on peut simplifier l'utilisation de ceux-ci. Le programme suivant, parfaitement légal, est équivalent au précédent.

```
DECLARE
  CURSOR SalCur IS
         SELECT * FROM EMP WHERE SAL<6000.00;
  employe   EMP%ROWTYPE;
BEGIN
  FOR employe IN SalCur
  LOOP
        UPDATE EMP
        SET SAL=6500.00  WHERE EMPNO=employe.empno;
  END LOOP;
END;
```

Dans cette boucle, l'ouverture, la fermeture et l'incrémentation du curseur `SalCur` sont implicites et n'ont pas besoin d'être spécifiées.

Un certain nombre d'informations sur l'état d'un curseur sont exploitables à l'aide d'attributs prédéfinis : l'attribut `%FOUND` renvoie vrai si le dernier `FETCH` a bien ramené un tuple. L'attribut `%NOTFOUND`, dual du précédent, permet de décider si on est arrivé en fin de curseur. `%ROWCOUNT` renvoie le nombre de lignes ramenés du curseur au moment de l'interrogation. Enfin `%ISOPEN` permet de déterminer si le curseur est ouvert.

6 La gestion des exceptions

PL/SQL permet de définir dans une zone particulière (de gestion d'exception), l'attitude que le programme doit avoir lorsque certaines erreurs définies ou prédéfinies se produisent.

Un certain nombre d'exceptions sont prédéfinies sous Oracle. Citons, pour les plus fréquentes : `NO_DATA_FOUND` (devient vrai dès qu'une requête renvoie un résultat vide), `TOO_MANY_ROWS` (requête renvoie plus de lignes qu'escompté), `CURSOR_ALREADY_OPEN` (curseur déja ouvert), `INVALID_CURSOR` (curseur invalide)...

L'utilisateur peut définir ses propres exceptions. Dans ce cas, il doit définir celles ci dans la zone de déclaration. Exemple :

```
excpt1 EXCEPTION
```

Puis, cette exception est levée quelque part dans le programme (après un test non concluant, par exemple), par l'instruction :

```
RAISE excpt1
```

Enfin, dans la zone d'exception un traitement est affecté à chaque exception possible (définie ou prédéfnie) :

```
EXCEPTION
   WHEN <exception1> [OR <exception2> OR ...] THEN <instructions>
   WHEN <exception3> [OR <exception2> OR ...] THEN <instructions>
   WHEN OTHERS THEN <instructions>
END;
```

Evidemment, un seul traitement d'exception peut se produire avant la sortie du bloc.

7 Procédures et fonctions

Il est possible de créer des procédures et des fonctions comme dans n'importe quel langage de programmation classique. La syntaxe de création d'une procédure est la suivante :

```
CREATE OR REPLACE PROCEDURE <nom>[ (liste de parametres) ]
AS
    <zone de declaration de varaibles>
BEGIN

    <corps de la procedure>
```

```
EXCEPTION
    <traitement des exceptions>
END;
```

A noter que toute fonction ou procdure crée devient un objet part entière de la base (comme une table ou une vue, par exemple). Elle est souvent appelée "procédure stockée". Elle est donc, entre autres, sensible à la notion de droit : son créateur peut dcider par exemple d'en permettre l'utilisation à d'autres utilisateurs. Elle est aussi appelable depuis n'importe quel bloc PL/SQL [1]. Il y a trois façon de passer les paramêtres dans une procédure : IN (lecture seule), OUT (écriture seule), INOUT (lecture et écriture). Le mode IN est réservé au paramêtres qui ne doivent pas être modifiés par la procédure. Le mode OUT, pour les paramêtres transmis en résultat, le mode INOUT pour les variables dont la valeur peut être modifiée en sortie et consultée par la procédure (penser aux différents passages de paramêtres dans les langages de programmation classiques...). Regardons l'exemple suivant :

```
CREATE TABLE T (num1 INTEGER, num2 INTEGER)

CREATE OR REPLACE PROCEDURE essai(IN x NUMBER, OUT y NUMBER, INOUT z NUMBER)
AS
BEGIN
    y:=x*z;
    z:=z*z;
    INSERT INTO T VALUES(x,z);
END;
```

Cette fonction est appelée dans le bloc suivant :

```
DECLARE
    a NUMBER;
    b NUMBER;
BEGIN
    a:=5;
    b:=10;
    essai(2,a,b);
    a:=a*b;
```

Après l'appel, le couple (a, b) vaut $(20, 100)$ et c'est aussi le tuple qui est inséré dans T. Puis, a prend la valeur $20 * 100 = 2000$.

[1] sur le même principe, on peut créer des bibliothèques de fonctions et de procédures appelées "packages"

Lors de l'appel de la procédure, les arguments correspondants aux paramêtres passés en mode OUT ou INOUT ne peuvent être des constantes (puisqu'ils doivent être modifiables).

Cette distinction des paramêtres en fonction de l'usage qui va en être fait dans la procédure est très pratique et facilite grandement le débuggage.

On peut aussi créer des fonctions. Le principe est le même, l'en-tète devient :

```
CREATE OR REPLACE FUNCTION <nom>[(parametres)] RETURN <type du resultat>
```

Une instruction `RETURN <expression>` devra se trouver dans le corps pour spécifier quel résultat est renvoyé.

Les procédures et fonctions PL/SQL supportent assez bien la surcharge (i.e. coexistence de procédures de même nom ayant des listes de paramêtres différentes). C'est le système qui, au moment de l'appel, inférera, en fonction du nombre d'arguments et de leur types, quelle est la bonne procédure à appeler.

On peut détruire une procédure ou une fonction par [2] :

```
DROP PROCEDURE <nom de procedure>
```

8 Les déclencheurs ("triggers")

Les déclencheurs ou "triggers" sont des séquences d'actions définis par le programmeur qui se déclenchent, non pas sur un appel, mais directement quand un événement particulier (spécifié lors de la définition du trigger) sur une ou plusieures tables se produit.

Un trigger sera un objet stocké (comme une table ou une procédure) La syntaxe est la suivante:

```
CREATE [OR REPLACE] TRIGGER <nom>
{BEFORE|AFTER} {INSERT|DELETE|UPDATE} ON <nom de table>
[FOR EACH ROW [WHEN (<condition>)]]
<corps du trigger>
```

Un trigger se déclenche avant ou après (`BEFORE|AFTER`) une insertion, destruction ou mise à jour (`INSERT|DELETE|UPDATE`) sur une table (à noter que l'on peut exprimer des conditions plus complexes avec le `OR`: `INSERT OR DELETE` ...).

L'option `FOR EACH ROW [WHEN (<condition>)]]` fait s'executer le trigger à chaque modification d'une ligne de la table spécifiée (on dit que le trigger est de "niveau ligne"). En l'absence de cette option, le trigger est executé une seule fois ("niveau table").

Soit l'exemple suivant :

[2] les procédures et fonctions sont des objets stockés. On peut dont les détruire par des instructions similaires aux instructions de destructions de tables...

```
CREATE TABLE T1 (num1 INTEGER, num2 INTEGER);
CREATE TABLE T2 (num3 INTEGER, num4 INTEGER);

CREATE TRIGGER inverse
AFTER INSERT ON T1
FOR EACH ROW WHEN (NEW.num1 <=3)
BEGIN
     INSERT INTO T2 VALUES(:NEW.num2, :NEW.num1);
END inverse;
```

Ce trigger va, en cas d'insertion d'un tuple dans $T1$ dont la première coordonnée est inférieure à 3, insérer le tuple inverse dans $T2$. Les prefixes `NEW` et `OLD` (en cas de `UPDATE` ou de `DELETE`) vont permettre de faire référence aux valeurs des colonnes après et avant les modifications dans la table. Ils sont utilisés sous la forme `NEW.num1` dans la condition du trigger et sous la forme `:NEW.num1` dans le corps.

Un trigger peut être activé ou désactivé par les instructions :

```
ALTER TRIGGER <nom> {ENABLE|DISABLE};
```

et détruit par :

```
DROP TRIGGER <nom>.
```

9 Quelques remarques

9.1 Affichage

PL/SQL n'est pas un langage avec des fonctionalités d'entrées sorties évoluées (ce n'est pas son but!). Toutefois, on peut imprimer des messages et des valeurs de variables de plusieurs manières différentes. Le plus pratique est de faire appel à un package prédéfini : DBMS_output.

En premier lieu, taper au niveau de la ligne de commande :

```
SET SERVEROUTPUT ON
```

Pour afficher, on utilise alors la commande suivante :

```
DBMS_OUTPUT.PUT_LINE('Au revoir' || nom || ' a bientot');
```

Si on suppose que `nom` est une variable dont la valeur est "toto", l'instruction affichera : `Au revoir toto a bientot`.

9.2 Debugger

Pour chaque objet crée (procédure, fonction, trigger...), en cas d'erreur de compilation, on peut voir ces erreurs en tapant (si "essai" est un nom de procédure) :

```
SHOW ERRORS PROCEDURE essai;
```