



Ecole Nationale des Sciences de l'Informatique

Module : SGBD

Dr. Nizar Sghaier
nizar.sghaier@ensi-uma.tn

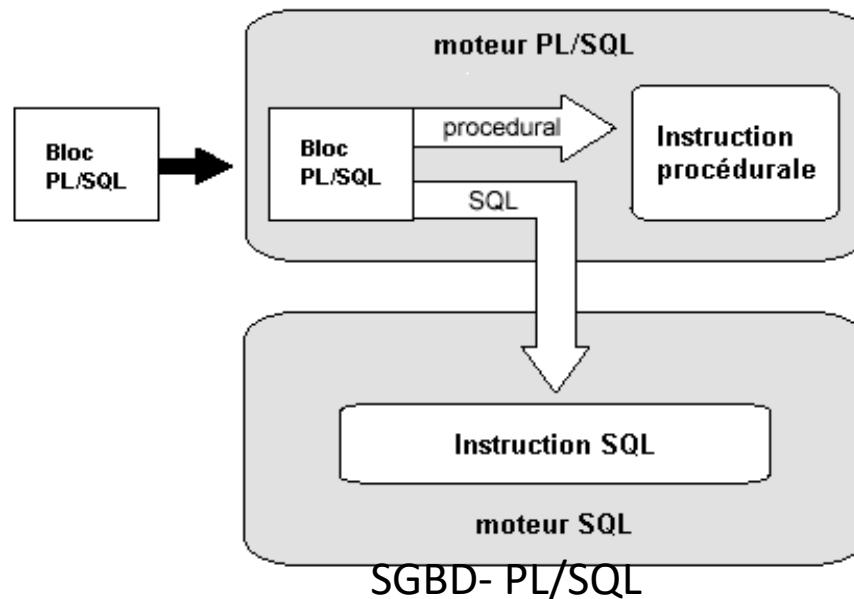
Partie II: Le Langage PL/SQL

Introduction au langage PL/SQL

- ❖ PL/SQL (Procedural Language / Structured Query Language) est une extension procédurale à SQL : PL/SQL intègre parfaitement le langage SQL en lui apportant une dimension procédurale.
- ❖ En effet, le langage SQL est un langage déclaratif non procédural permettant d'exprimer des requêtes dans un langage relativement simple. En contrepartie il n'intègre aucune structure de contrôle permettant par exemple d'exécuter une boucle itérative.
- ❖ Le langage PL/SQL permet de définir un ensemble de commandes contenues dans ce que l'on appelle un "bloc" PL/SQL. Un bloc PL/SQL peut lui-même contenir des sous-blocs.
- ❖ Le bloc PL/SQL est analysé et exécuté par le moteur PL/SQL.

Introduction au langage PL/SQL

- Lorsque le moteur PL/SQL reçoit un bloc pour exécution, effectue les opérations suivantes :
 - Séparation des ordres SQL et PL/SQL
 - Passage des commandes SQL au processeur SQL (SQL Statement Executor)
 - Passage des instructions procédurales au processeur d'instructions procédurales (Procedural Statement Executor)

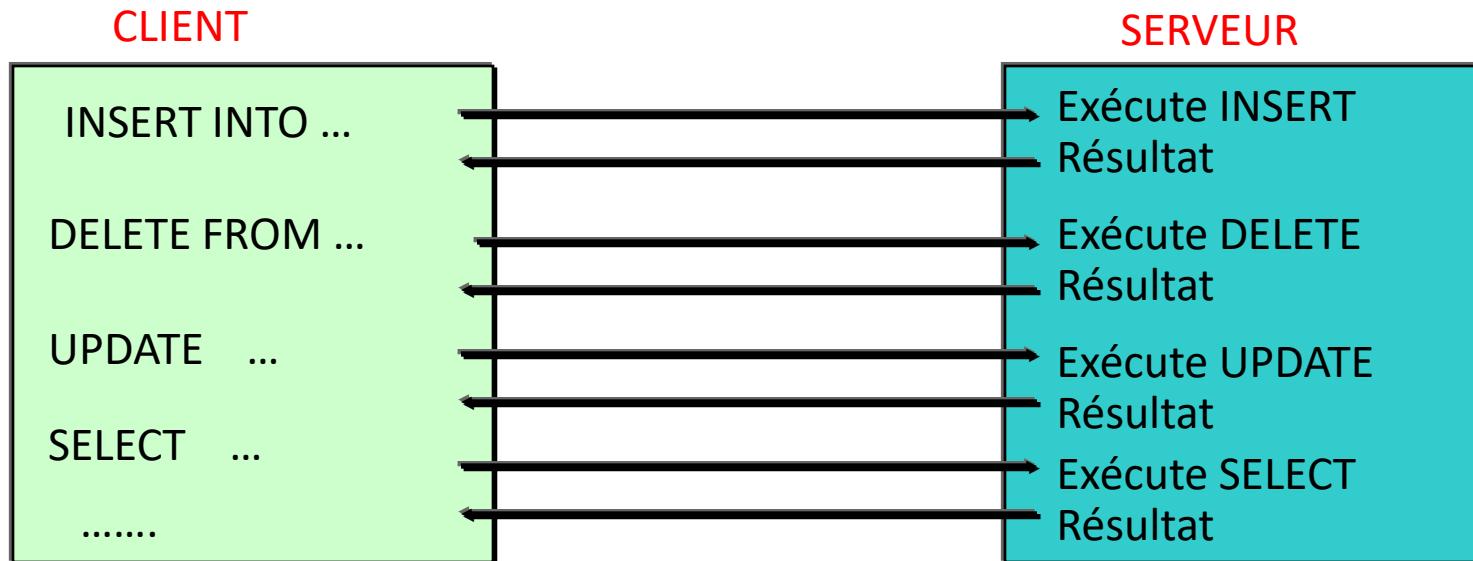


Avantages de PL/SQL

- PL/SQL joue un rôle central entre le serveur Oracle (à travers les procédures, les fonctions, les packages, les déclencheurs de la base de données, etc.) et les différents outils de développement d'Oracle.
- PL/SQL peut regrouper un ensemble de requêtes SQL en un seul bloc et les envoie au serveur en une seule fois, ce qui réduit le trafic au niveau du serveur et augmente la performance.
- PL/SQL peut être utilisé par les outils d'Oracle (Forms, Reports, Designer2000, etc.)

Requêtes SQL

- Chaque requête ‘client’ est transmise au serveur de données pour être exécutée avec retour de résultats



Bloc PL/SQL

- Le bloc de requêtes est envoyé sur le serveur. Celui-ci exécute le bloc et renvoie 1 résultat final.



PL/SQL Vs SQL

PL/SQL = SQL +

- Variables et constantes
- Traitements conditionnels
- Traitements itératifs
- Curseurs en PL/SQL
- Les enregistrements et les tables PL/SQL
- Procédures
- Fonctions
- Les packages en Oracle
- Le traitement des exceptions (erreurs)

Format d'un bloc PL/SQL

Section DECLARE :

- Variables;
- Constantes;
- Curseurs;
- Tables PL/SQL;
- Enregistrements PL/SQL;
- Exceptions;

Section BEGIN

- Section des ordres exécutables
- Ordres SQL
- Ordres PL

Section EXCEPTION

- Réception en cas d'erreur
- Exceptions SQL ou utilisateur

```
DECLARE  
    --déclarations  
BEGIN  
    --exécutions  
EXCEPTION  
    --erreurs  
END;  
/
```

Commentaires

- Chaque instruction se termine par ‘;’
- Les parties ‘DECLARE’ et ‘EXCEPTION’ sont facultatives
- On peut inclure des commentaires dans un bloc PL/SQL
 - commentaires sur une seule ligne
 - /* */** commentaires sur plusieurs lignes

Les variables

- **Elles servent à :**
 - stocker temporairement des données
 - manipuler des valeurs stockées
 - elles sont réutilisables
 - elles sont faciles à maintenir
- **Types de variables :**
 - a. variables de type Oracle
 - b. variables booléennes
 - c. variables faisant référence au dictionnaire de données

Les variables

- Le nom des variables ne doit pas être le même que celui d'une colonne ou d'une table de la base de données.
- Pour affecter une valeur à une variable, on utilise :

:=

Default

Not null

Exemples : v_sal := 2500;

v_Mgr number(4) Default 7839;

v_loc varchar(13) not null := 'MONTRÉAL';

- Il est préférable de précéder le nom des variables par 'v_'.
Ex. : v_sal, v_nom, etc.

a. Variables de type Oracle :

Char, Varchar2, Number, Date, Long, Long Row

b. Variables booléennes :

Exemple : v_valide BOOLEAN := true;

c. Variables faisant référence au dictionnaire de données.

Les variables: exemple 1

- Pour la table Dept, créer des variables qui auront le même type et la même dimension que les colonnes de Dept. Leur affecter respectivement les valeurs : 60, ‘RHU’, ‘MONTRÉAL’
- NB. : On utilise sous SQL*Plus la commande : **SET SERVEROUTPUT ON** pour pouvoir afficher les résultats de la commande Oracle (Package) : **DBMS_OUTPUT.PUT_LINE(chaîne)**

Réponse

```
DECLARE
    v_deptno number(2) := 60;
    v_dname varchar2(14) := 'RHU';
    v_loc varchar2(13) := 'MONTRÉAL';
BEGIN
    DBMS_OUTPUT.PUT_LINE('No dept : '||v_deptno);
    DBMS_OUTPUT.PUT_LINE('Nom dept : '||v_dname);
    DBMS_OUTPUT.PUT_LINE('Loc : '||v_loc);
END;
```

N.B. Pour affecter les colonnes d'une ligne d'une table :

SELECT colonne INTO v_col

FROM nom de table

WHERE condition(s);

Les variables: exemple 2

- Déclarer les variables v_deptno, v_dname, v_loc correspondantes aux colonnes de Dept.
- Affecter à ces variables le no, le nom et la localisation du département no 20.
- Afficher les valeurs de ces variables.

Réponse

```
DECLARE
    v_deptno number(2);
    v_dname varchar2(14);
    v_loc varchar2(13);
BEGIN
    SELECT deptno, dname, loc INTO v_deptno, v_dname, v_loc
    FROM DEPT
    WHERE deptno=20;
    DBMS_OUTPUT.PUT_LINE('No dept : '||v_deptno);
    DBMS_OUTPUT.PUT_LINE('Nom dept : '||v_dname);
    DBMS_OUTPUT.PUT_LINE('Loc : '||v_loc);
END;
```

Les variables: exemple 3

- Insérer les valeurs 60, ‘RHU’, ‘MONTRÉAL’ dans la table DEPT.

Réponse

```
DECLARE
    v_deptno number(2) := 60;
    v_dname varchar2(14) := 'RHU';
    v_loc varchar2(13) := 'MONTRÉAL';
BEGIN
    INSERT INTO DEPT('deptno, dname, loc)
    VALUES (v_deptno, v_dname, v_loc);
END;
```

Les variables: exemple 4

-
- Utilisation de variables SQL*Plus.
 - Créer des variables sous SQL*Plus dans lequel on affecte les valeurs 70, ‘Finance’, ‘Québec’
 - Affecter ces variables à d’autres variables qu’on va créer dans un bloc PL/SQL.
 - Afficher ces variables.

Réponse

a. Sous SQL*Plus, on crée les variables suivantes :

```
ACCEPT p_dept 70
ACCEPT p_dname Finance
ACCEPT p_loc Québec
```

b. Bloc PL/SQL

```
DECLARE
```

```
    v_deptno number(2) := &p_deptno;
    v_dname varchar2(14) :=&p_dname;
    v_loc varchar2(13) := &p_loc;
```

```
BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE('No dept : '|| v_deptno);
    DBMS_OUTPUT.PUT_LINE('Nom dept : '|| v_dname);
    DBMS_OUTPUT.PUT_LINE('Loc dept : '|| v_loc);
```

END;

Variables de même type qu'une colonne d'une table de BD

Syntaxe :

Nom_var Table.colonne%type;

Exemple : v_dname DEPT.dname%type;

Exemple:

- Affecter les valeurs des colonnes de la table DEPT dont le no Dept=30 aux variables v_deptno, v_dname, v_loc.
- Afficher les valeurs de ces variables .

Réponse :

DECLARE

```
v_deptno Dept.deptno%type := 60;  
v_dname Dept.dname%type := 'RHU';  
v_loc Dept.loc%type := 'MONTRÉAL';
```

BEGIN

```
DBMS_OUTPUT.PUT_LINE('No dept : ' || v_deptno);  
DBMS_OUTPUT.PUT_LINE('Nom dept : ' || v_dname);  
DBMS_OUTPUT.PUT_LINE('Loc : ' || v_loc);
```

END;

Variables de même type qu'une Ligne d'une table de BD

Syntaxe :

- v_ligne Table%ROWTYPE
- exemple : v_employe EMP%ROWTYPE

Exemple:

- Affecter la ligne de la table DEPT dont deptno=30 à la variable v_dept.
- Afficher le contenu de la variable.

Réponse :

```
DECLARE
    v_dept Dept%rowtype;
BEGIN
    SELECT * INTO v_dept
    FROM DEPT
    WHERE deptno=30;
    DBMS_OUTPUT.PUT_LINE('No dept : '||v_dept.deptno);
    DBMS_OUTPUT.PUT_LINE('Nom dept : '||v_dept.dname);
    DBMS_OUTPUT.PUT_LINE('Loc : '||v_dept.loc);
END;
```

Affectations

Il existe plusieurs possibilités pour affecter une valeur à une variable :

- l'affectation comme on la connaît dans les langages de programmation (*variable, expression*)
 - par la directive DEFAULT ;
 - par la directive INTO d'une requête (SELECT ... **INTO variable FROM ...**).

Le tableau suivant décrit quelques exemples :

Code PL/SQL	Commentaires
DECLARE v_brevet VARCHAR2(6); v_brevet2 VARCHAR2(6); v_prime NUMBER(5,2); v_naissance DATE; v_trouvé BOOLEAN NOT NULL DEFAULT FALSE;	Déclarations de variables.
BEGIN v_brevet := 'PL-1';	Affectation d'une chaîne de caractères.
v_brevet2 := v_brevet;	Affectation d'une variable.
v_prime := 500.50;	Affectation d'un nombre.
v_naissance := '04-07-2003';	Affectation de dates.
v_naissance := TO_DATE('04-07-2003 17:30', 'DD/MM/YYYY HH24:MI');	
v_trouvé := TRUE;	Affectation d'un booléen.
SELECT brevet INTO v_brevet FROM Pilote WHERE nom = 'Gratien Viel';	Affectation d'une chaîne de caractères par une requête.
...	

Les traitements inconditionnels

```
IF condition1 THEN  
    traitement1;  
ELSIF condition2 THEN  
    traitement2;  
[ELSE  
    traitement3;]  
ENDIF;
```

- Les opérateurs utilisés dans les conditions sont les mêmes que dans SQL (=, <, >, <=, >=, IS NULL, LIKE, ...)
- Dès que l'une des conditions est vraie, le traitement qui suit le THEN est exécuté.
- Si aucune condition n'est vraie, c'est le traitement qui suit le ELSE qui est exécuté.

Les traitements inconditionnels

Exemple: Affecter le nom et le salaire de l'employé 'SMITH' aux variables n_nom, v_sal.

Afficher ces variables, pour v_sal si :

- v_sal < 1000 alors afficher aussi 'Bas salaire'
- v_sal >= 1000 alors afficher aussi 'Haut salaire'

Réponse :

```
DECLARE
    v_nom EMP.ename%type;
    v_sal EMP.sal%type;
BEGIN
    SELECT ename, sal INTO v_nom, v_sal
    FROM EMP
    WHERE ename = 'SMITH';
    DBMS_OUTPUT.PUT_LINE('Nom : ' || v_nom);
    IF v_sal < 1000 THEN
        DBMS_OUTPUT.PUT_LINE('Salaire : ' || v_sal || '' || 'Bas salaire');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Salaire : ' || v_sal || '' || 'Haut salaire');
    ENDIF;
END;
```

Les traitements itératifs: LOOP

1. la boucle LOOP

Syntaxe :

LOOP

 Expression1;

 Expression2;

.

 EXIT [WHEN condition];

END LOOP;

Exemple: Initialiser la variable v_nb à 10.

Afficher la valeur de cette variable, puis incrémenter sa valeur de 1 jusqu'à la valeur 15.

Réponse:

DECLARE

 v_nb NUMBER :=10;

BEGIN

 LOOP

 DBMS_OUTPUT.PUT_LINE(v_nb);

 v_nb := v_nb + 1;

 EXIT WHEN v_nb > 15;

 END LOOP;

END;

Les traitements itératifs: LOOP

Exemple2:

Créer une table de multiplication Mult(Op char(5), Res char(3)).

Écrire un programme qui fait le calcul et l'affichage de la table Mult.

On demandera à l'utilisateur quelle table de multiplication il souhaite.

Réponse:

```
CREATE TABLE Mult(Op char(5), res char(3));
PROMPT Quelle table voulez-vous?
ACCEPT TABLE
DECLARE
    Compteur NUMBER := 0;
BEGIN
    LOOP
        Compteur := compteur + 1;
        INSERT INTO Mult VALUES(
            TO_CHAR(compteur) || '*' || TO_CHAR(&TABLE),
            TO_CHAR(compteur*&TABLE));
        EXIT WHEN compteur = 10;
    END LOOP;
END;
SELECT * FROM Mult;
ROLLBACK;
```

Les traitements itératifs: FOR

Exécution d'un traitement un certain nombre de fois, le nombre étant connu.

Syntaxe :

BEGIN

...

FOR indice IN [REVERSE] exp1..exp2

LOOP

Instructions;

END LOOP;

END;

N.B. :

- Inutile de déclarer la variable 'indice'
- Indice varie de exp1 à exp2 avec 1 pas de 1
- Si 'REVERSE' est précisé alors 'indice' varie de exp1 à exp2 avec un pas de -1

Les traitements itératifs: FOR

Exemple: Faites le calcul de factoriel de 5. Afficher le résultat. $5!=5*4*3*2*1$

Réponse :

```
DECLARE
v_fact NUMBER := 1;
BEGIN
    FOR I IN 1..5
    LOOP
        v_fact = v_fact * i;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Factoriel de 5 : '||v_fact);
END;
```

Les traitements itératifs: WHILE

Exemple:

Créer une table de multiplication Mult(op char(5), Res char(3)).

Écrire un programme qui fait le calcul et l'affichage de la table Mult.

On demandera à l'utilisateur quelle table de multiplication il souhaite.

Réponse :

```
DECLARE
    Compteur NUMBER :=0;
BEGIN
    WHILE compteur < 10
    LOOP
        Compteur := compteur + 1;
        INSERT INTO MULT VALUES (
        TO_CHAR(compteur)||'*'||TO_CHAR(&TABLE),
        TO_CHAR(compteur*&TABLE));
    END LOOP;
END;
SELECT * FROM MULT;
```

Structure alternative : CASE (1)

Cette instruction permet de mettre en place des structures de test conditionnel de type IF .. ELSE .. END IF,
2 syntaxes sont possibles :

[<<étiquette>>]

CASE var

WHEN *contenu-var* THEN *instructions1*;

WHEN *contenu-var* THEN *instructions2*;

...

WHEN *contenu-var* THEN *instructionsN*;

[ELSE *instructionsN+1*;]

END CASE [*étiquette*];

[<<étiquette>>]

CASE

WHEN *condition1* THEN *instructions1*;

WHEN *condition2* THEN *instructions2*;

...

WHEN *conditionN* THEN *instructionsN*;

[ELSE *instructionsN+1*;]

END CASE [*étiquette*];

Structure alternative : CASE (2)

```
Declare
    V_Num integer := 0 ;
Begin
    Loop
        V_Num := V_Num + 1 ;
        CASE V_Num
            WHEN 1 Then dbms_output.put_line( '1' ) ;
            WHEN 2 Then dbms_output.put_line( '2' ) ;
            WHEN 3 Then dbms_output.put_line( '3' ) ;
            ELSE
                EXIT ;
            END CASE ;
        End loop ;
    End ;
```

Les enregistrements et les tables PL/SQL

- De même que les variables, les variables composées ont aussi des types de données.
- Les types de données composées (collection) sont soit des enregistrements (Record), soit des tables (Matrice).
- Un enregistrement est un groupe d'items stockés dans des champs chacun d'eux (item) a un nom et un type de données.
- Une table contient une colonne et une clé primaire qui permet l'accès aux lignes.
- Une fois définis, les enregistrements et les tables PL/SQL peuvent être réutilisés.
- Chaque enregistrement peut avoir autant de champs (colonnes) que nécessaire.
- On peut associer à un enregistrement des valeurs initiales et peuvent être Not Null.
- Les champs sans valeurs initiales sont initialisées à Null
- La valeur ‘Default’ peut être utilisée lors de la définition d'un enregistrement.
- On peut définir un enregistrement dans la partie déclaration d'un bloc, d'une procédure, d'une fonction ou d'un package.
- Un enregistrement peut être composé d'un autre enregistrement.

Création d'un enregistrement

Syntaxe :

```
TYPE nom_type IS RECORD  
    Décl_champ1, décl_champ2, ...);  
    Identificateur nom_type;
```

Où

nom_type : nom de l'enregistrement

déclar_champ : nom_champ {type_champ | variable%type |
 table.col%type |
 table%rowtype}[[not null] { := | Default} expr]

N.B. : Les champs déclarés Not Null doivent être initialisés.

Création d'un enregistrement

Exemple 1 :

Créer un enregistrement qui contient 3 champs : le nom, le job et le salaire.
Lui affecter le nom, le job et le salaire de l'employé 'SMITH' de la EMP.

Réponse :

```
DECLARE
    TYPE type_enregist IS RECORD
        (nom EMP.ename%type, job EMP.job%type, sal EMP.sal%type);
    enregist type_enregist;
BEGIN
    SELECT ename, job, sal INTO enregist
    FROM EMP
    WHERE ename='SMITH';
    DBMS_OUTPUT.PUT_LINE(enregist.nom||enregist.job||enregist.sal);
END;
```

Création d'un enregistrement

Exemple 2:

Écrire un programme pour afficher les informations de la table DEPT en utilisant les enregistrements, et SQL*Plus pour faire entrer le no dept.

Réponse :

SQL*Plus	{	Set server output on Set verify off Accept p_deptno	(Paramètres SQL*Plus) affiche la valeur de p_deptno avant et après exécution
----------	---	---	---

```
DECLARE
    v_dept DEPT%ROWTYPE;
BEGIN
    SELECT * INTO v_dept
    FROM DEPT
    WHERE deptno = &p_deptno
    DBMS_OUTPUT.PUT_LINE(v_dept.deptno||v_dept.dname||v_dept.loc);
END;
```

Les tables

Une table PL/SQL est similaire à un tableau (Matrice).

Une table PL/SQL doit contenir deux (02) composants.

C'est une matrice dynamique (dimension variable)

1. Une clé primaire de type “BINARY-INTEGER” (entier binaire) qui indexe la table PL/SQL
2. Une colonne de type scalaire (char, date, number,...) ou de type enregistrement pour les données dans la table PL/SQL

Syntaxe :

```
TYPE nom_type IS TABLE OF  
    {type_col | variable%type | table.col%type} [Not Null]  
    index by binary_integer;
```

ident nom_type;

où :

nom_type : c'est la déclaration du nom du type de la table.

Type_col : char, number, varchar2, date,...

Ident : c'est le nom de l'identificateur qui représente la tablePL/SQL.

Les tables

Exemple1: Créer la table PL/SQL de type date dans laquelle on affecte le champ hiredate de la table EMP pour les employés ‘SMITH’ et ‘SCOTT’.

Réponse:

```
DECLARE
    v_ind number := 1;
    TYPE d_table_type IS TABLE OF date
        Index by binary-integer;
    d_table d_table_type;
BEGIN
    SELECT hiredate INTO d_table(v_ind)
    FROM EMP
    WHERE ename = 'SMITH';
    DBMS_OUTPUT.PUT_LINE(d_table(v_ind));
    v_ind := v_ind + 1;
    SELECT hiredate INTO d_table(v_ind)
    FROM EMP
    WHERE ename = 'SCOTT';
    DBMS_OUTPUT.PUT_LINE(d_table(v_ind));
END;
```

Les tables

Exemple2: Créer une table PL/SQL qui contient toute la ligne de DEPT.

Réponse:

```
DECLARE
    v_nb number(2);
    TYPE dept_table_type IS TABLE OF DEPT%type
    Index by binary-integer;
    dept_table dept_table_type;
BEGIN
    SELECT COUNT (*) INTO v_nb
    FROM DEPT;
    FOR i IN 1..v_nb LOOP
        SELECT * INTO dept_table(i)
        FROM DEPT
        WHERE deptno = i*10;
    END LOOP;
    FOR i IN 1..v_nb LOOP
        DBMS_OUTPUT.PUT_LINE(dept_table(i).deptno ||"||"
        dept_table(i).dname || "||"
        dept_table(i).loc);
    END LOOP;
END;
```

Les tables

Utilisation des méthodes pour les tables PL/SQL

- **Exists(n)** : retourne vrai si nième élément de la table PL/SQL existe.
- **Count** : retourne le nombre d'éléments que la table contient.
- **First** : retourne le plus petit indice de la table PL.
- **Last** : retourne le grand indice de la table PL. Retourne NULL si la table est vide.
- **Prior(n)** : retourne l'indice qui précède l'indice n.
- **Next(n)** : retourne l'indice qui succède l'indice n.
- **Extend(n,i)** : augmente la taille d'une table PL.
 - Extend : ajoute un élément null à la table
 - Extend(n) : ajoute n éléments null à la table.
 - Extend(n,i) : ajoute n copies du i^{ème} élément à la table.
- **Trim :**
 - Trim : supprime un élément à partir de la fin de la table.
 - Trim(n) : supprime n éléments à partir de la fin de la table.
- **Delete :**
 - Delete : supprime tous les éléments de la table PL.
 - Delete(n) : supprime n éléments de la table PL.
 - Delete(m,n) : supprime tous les éléments dans l'intervalle m..n de la table PL.

Syntaxe :

nom_table.nom_méthode[(paramètres)]

Ex :

Ajouter à l'exercice précédent ce qui suit :

```
DBMS_OUTPUT.PUT_LINE('Le nombre d\'éléments de la table est : '||d_table.count);
DBMS_OUTPUT.PUT_LINE('Le plus petit indice est : '||d_table.first);
DBMS_OUTPUT.PUT_LINE('Le plus grand indice est : '||d_table.last);
```

Les Curseurs

Principe des curseurs

Il existe deux types de curseurs :

- **Curseurs implicite** : c'est un curseur SQL généré et géré par le noyau d'Oracle pour chaque ordre SQL d'un bloc.
- **Curseur explicite** : c'est un curseur SQL généré et géré par l'utilisateur pour traiter un ordre SELECT qui ramène plus d'une ligne.

Curseur explicite : 4 étapes

1. Déclaration du curseur
2. Ouverture du curseur
3. Traitement des lignes
4. Fermeture du curseur

Démarche générale des curseurs

- Déclaration du curseur : DECLARE
 - Ordre SQL sans exécution
- Ouverture du curseur : OPEN
 - SQL ‘monte’ les lignes sélectionnées en SGA
 - Verrouillage préventif possible (voir + loin)
- Sélection d’une ligne : FETCH
 - Chaque FETCH ramène une ligne dans le programme client
 - Tant que ligne en SGA : FETCH
- Fermeture du curseur : CLOSE
 - Récupération de l’espace mémoire en SGA

Démarche générale des curseurs

1. **Déclaration du curseur :** se fait dans la partie DECLARE.

Syntaxe :

```
DECLARE
    CURSOR cur_emp IS
        SELECT empno, ename, sal
        FROM EMP
        WHERE deptno = 10;
        ...
BEGIN
    ...
END;
```

2. **Ouverture du curseur :** l'ouverture du curseur lance l'exécution de l'ordre SELECT associé au curseur. L'ouverture se fait dans la section BEGIN du bloc.

Syntaxe :

```
OPEN nom curseur
```

3. **Traitement de lignes :** Après l'exécution du SELECT, les lignes ramenées sont traitées une par une, la valeur de chaque colonne du SELECT doit être stockée dans une variable réceptrice.

Syntaxe :

```
FETCH cur_emp INTO liste_variables
```

4. **Fermeture du curseur :** Il est important de fermer le curseur une fois qu'on en a plus besoin, cela permet de libérer la mémoire occupée par ce dernier.

Syntaxe :

```
CLOSE nom curseur; /* voir l'exemple précédent */
```

Démarche générale des curseurs

Exemple1:

Afficher le No d'employé, le nom, hiredate de l'employé ‘SCOTT’. Utiliser un curseur.

Réponse:

```
DECLARE
    v_empno EMP.empno%type;
    v_nom EMP.ename%type;
    v_date EMP.hiredate%type;
    CURSOR cur_scott IS
        SELECT empno, ename, hiredate
        FROM EMP
        WHERE ename = 'SCOTT';
BEGIN
    OPEN cur_scott;
    FETCH cur_scott INTO v_empno, v_nom, v_date;
    CLOSE cur_scott;
    DBMS_OUTPUT.PUT_LINE(v_empno, v_nom, v_date);
END;
```

Démarche générale des curseurs

Exemple2:

Afficher le nom, le salaire des cinq premiers employés de la table EMP.

Réponse:

```
DECLARE
    v_nom EMP.ename%type;
    v_sal EMP.sal %type;
    CURSOR cur_sal IS
        SELECT e.ename, e.sal
        FROM EMP
BEGIN
    OPEN cur_sal;
    FOR i IN 1..5
    LOOP
        FETCH cur_sal INTO v_nom, v_sal;
        DBMS_OUTPUT.PUT_LINE('Nom : '||v_nom);
        DBMS_OUTPUT.PUT_LINE('Salaire : '||v_sal);
    END LOOP;
    CLOSE cur_sal;
END;
```

Les attributs d'un curseurs

%FOUND } Dernière ligne traitée
%NOTFOUND

%ISOPEN : ouverture d'un curseur

%ROWCOUNT : nombre de lignes déjà traitées

nom curseur%FOUND = true : si le dernier FETCH a ramené une ligne.

nom curseur%NOTFOUND = true : si le dernier FETCH n'a pas ramené de ligne

nom curseur%ISOPEN = true : si le curseur est ouvert

ex. : IF NOT (nom curseur%ISOPEN) THEN
 OPEN nom curseur;
 ENDIF;

nom curseur%ROWCOUNT : traduit la nième ligne ramenée par le FETCH.

Les attributs d'un curseurs

Exemple: Afficher le nom et le salaire des 6 premiers employés de EMP. Utiliser %ROWCOUNT, %NOTFOUND et la boucle LOOP.

Réponse :

```
DECLARE
    v_nom EMP.ename%TYPE;
    v_sal EMP.sal%TYPE;
    CURSOR cur_sal IS
        SELECT ename, sal FROM EMP;
BEGIN
    OPEN cur_sal;
    LOOP
        FETCH cur_sal INTO v_nom, v_sal;
        DBMS_OUTPUT.PUT_LINE('Nom : ' || v_nom || ' ' || 'Sal : ' || v_sal);
        EXIT WHEN cur_sal%ROWCOUNT > 6 OR cur_sal%NOTFOUND;
        /* affichage de la dernière ligne*/
        DBMS_OUTPUT.PUT_LINE('Nom : ' || v_nom || ' ' || 'Sal : ' || v_sal);
    END LOOP;
    CLOSE cur_sal;
END;
```

Les attributs d'un curseurs

Curseur FOR LOOP :

Syntaxe :

```
FOR nom_enregist IN nom curseur LOOP  
    Instruction1;  
    Instruction2;  
    ...  
END LOOP;
```

N.B. :

- l'ouverture du curseur, son FETCH et sa fermeture sont implicite
- l'enregistrement (nom_enregist) est déclaré implicitement.

Exo #1:

Afficher le nom et le salaire de tous les employés de EMP.

Réponse :

```
DECLARE  
    CURSOR cur_emp IS SELECT ename, sal, FROM EMP;  
BEGIN  
    FOR v_emp IN cur_emp LOOP  
        DBMS_OUTPUT.PUT_LINE('Nom : '|| v_emp.ename || ' '|| 'Sal : '||v_emp.sal);  
    END LOOP;  
END;
```

Les attributs d'un curseurs

Curseurs avec paramètres

Syntaxe :

```
CURSOR nom_curseur [(param1 type1, param2 type2, ...)] IS ordre select;  
...  
OPEN nom_curseur(contenu de param1, contenu de param2, ...);
```

N.B. :

type = CHAR, VARCHAR2, NUMBER, %TYPE,...

Exo #1 :

Créer un curseur avec paramètre p_deptno qui ramène le No, le nom et la localisation du département no 10

Réponse :

```
DECLARE  
    v_dept DEPT%ROWTYPE;  
    CURSOR cur_dept(p_num DEPT.deptno%type) IS  
        SELECT * FROM DEPT WHERE deptno = p_num;  
BEGIN  
    OPEN cur_dept(10); /* département 10 */  
    FETCH cur_dept INTO v_dept;  
    DBMS_OUTPUT.PUT_LINE(v_dept.deptnoll' || v_dept.dnamell' || v_dept.loc);  
    CLOSE cur_dept;  
END;
```

Les attributs d'un curseurs

Curseurs avec paramètres

Syntaxe :

```
CURSOR nom_curseur [(param1 type1, param2 type2, ...)] IS ordre select;
```

```
...
```

```
OPEN nom_curseur(contenu de param1, contenu de param2, ...);
```

N.B. :

type = CHAR, VARCHAR2, NUMBER, %TYPE,...

Exo #1 :

Créer un curseur avec paramètre p_deptno qui ramène le No, le nom et la localisation du département no 10

Réponse :

```
DECLARE
    v_dept DEPT%ROWTYPE;
    CURSOR cur_dept(p_num DEPT.deptno%type) IS
        SELECT * FROM DEPT WHERE deptno = p_num;
BEGIN
    OPEN cur_dept(10); /* département 10 */
    FETCH cur_dept INTO v_dept;
    DBMS_OUTPUT.PUT_LINE(v_dept.deptnoll' || v_dept.dnamell' || v_dept.loc);
    CLOSE cur_dept;
END;
```

Les Exceptions

Les exceptions (la gestion des erreurs)

La section EXCEPTION permet de gérer les erreurs survenues lors de l'exécution d'un bloc PL/SQL.

Il existe deux (2) types d'erreurs :

1. Erreurs utilisateur
2. Erreurs Oracle

Erreurs utilisateur

Syntaxe :

```
DECLARE
    nom_erreur EXCEPTION;
    ...
BEGIN
    ...
    IF condition THEN
        RAISE nom_erreur; /* On déclenche l'erreur */
    ...
EXCEPTION
    WHEN nom_erreur THEN traitement de l'erreur;
    [WHEN OTHERS THEN instr1;
        ....]
END;
```

N.B. :

Quand une erreur se déclenche, on peut identifier le code et le message de l'erreur, en utilisant les deux fonctions SQLCODE et SQLERRM.

SQLCODE : retourne le no de l'erreur.

SQLERRM : retourne le message associé à l'erreur.

Erreurs utilisateur

Exemple: Écrire un programme qui affiche le no, le nom et la localisation d'un département donné (le no dept servira d'entrée avec ACCEPT de SQL*PLUS. Si le no dept n'existe pas dans la table DEPT alors erreur.

Réponse :

ACCEPT p_dept

PL/SQL :

DECLARE

```
v_deptno DEPT.deptno%type;
v_dname DEPT.dname%type;
v_loc DEPT.loc%type;
CURSOR cur_dept IS
select deptno, dname, loc from DEPT where deptno = &p_dept;
v_except EXCEPTION;
```

BEGIN

OPEN cur_dept;

FETCH cur_dept INTO v_deptno, v_dname, v_loc;

IF cur_dept%NOTFOUND THEN

RAISE v_except;

ENDIF;

DBMS_OUTPUT.PUT_LINE(v_deptno||'||v_dname||'||v_loc);

CLOSE cur_dept;

EXCEPTION

WHEN v_except THEN

```
DBMS_OUTPUT.PUT_LINE('Le no dept n''existe pas ...');
```

```
DBMS_OUTPUT.PUT_LINE('Le no de l''erreur : '||SQLCODE);
```

```
DBMS_OUTPUT.PUT_LINE('Le message de l''erreur : '||SQLERRM);
```

END;

Erreurs oracle

Les plus courantes sont :

1. **NO_DATA_FOUND** : lorsqu'un SELECT ne ramène pas d'information.

Exp :

```
DECLARE
```

```
    v_ename EMP.ename%type;
```

```
BEGIN
```

```
    SELECT ename INTO v_ename FROM EMP WHERE empno = &v_empno;
```

```
    DBMS_OUTPUT.PUT_LINE('L"employé est : '||v_ename);
```

```
EXCEPTION
```

```
    WHEN NO_DATA_FOUND THEN
```

```
        DBMS_OUTPUT.PUT_LINE('Le no d"employé n'existe pas');
```

```
        DBMS_OUTPUT.PUT_LINE('Le no de l"erreur : '||SQLCODE); ➔100
```

```
        DBMS_OUTPUT.PUT_LINE('Le nom de l"erreur: '||SQLERRM);
```

```
END;
```



ORA-01403 : No data found

Erreurs oracle

2. **TOO_MANY_ROWS** : lorsqu'un SELECT ramène plus d'une ligne.

Exp :

DECLARE

v_ename EMP.ename%type;

BEGIN

SELECT ename INTO v_ename FROM EMP WHERE deptno = &v_deptno;

EXCEPTION

WHEN NO_DATA_FOUND THEN

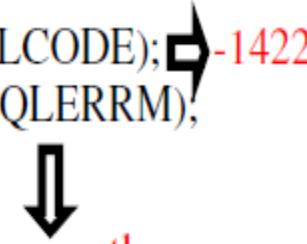
DBMS_OUTPUT.PUT_LINE('Le no de dept n''existe pas');

WHEN TOO_MANY_ROWS THEN

DBMS_OUTPUT.PUT_LINE('Le no de l''erreur: '||SQLCODE);

DBMS_OUTPUT.PUT_LINE('Le nom de l''erreur: '||SQLERRM),

END;



ORA-1422 : exact fetch returns more than ...

Erreurs oracle

3. **ZERO_DEVIDE** : division par zéro.

Exp :

```
DECLARE
```

```
    v_val number := 20;
```

```
BEGIN
```

```
    v_val := v_val / &v_val1;
```

```
    DBMS_OUTPUT.PUT_LINE('v_val = '||v_val);
```

```
EXCEPTION
```

```
    WHEN ZERO_DIVIDE THEN
```

```
        DBMS_OUTPUT.PUT_LINE('Division par zéro ...');
```

```
        DBMS_OUTPUT.PUT_LINE('Le no de l"erreur: '||SQLCODE);
```

```
        DBMS_OUTPUT.PUT_LINE('Le nom de l"erreur: '||SQLERRM);
```

```
END;
```



ORA-01476 : Division is equal to zero

Erreurs oracle

DUP_VAL_ON_INDEX : violation des contraintes de la clé primaire ou unique.
Insertion de valeurs dupliquées.

VALUE_ERROR : erreur arithmétique.

INVALID_NUMBER : problème dans la conversion de caractères en nombres.

CURSOR_ALREADY_OPEN : le curseur est déjà ouvert.

INVALID_CURSOR : l'erreur surgit quand on essaie de fermer un curseur qui est déjà fermé.

ROWTYPE_MISMATCH : les types de données de l'enregistrement auxquels des données du curseur sont assignées sont incompatibles.

Les procédures et les fonctions

Introduction

- Bien que PL/SQL permet l'imbrication de blocs, cette pratique n'est pas recommandée pour deux raisons majeures :
 - Réduit la lisibilité du code et rend plus difficile sa maintenance
 - Les blocs emboîtés ne peuvent pas être appelés
- Les sous programmes
 - PL/SQL offre la possibilité de travailler avec des sous programmes
 - Un sous programme PL/SQL est un bloc nommé défini dans la section DECLARE et pouvant posséder des paramètres
 - Il est réutilisable et facilite la maintenance des applications

Procédures

- Syntaxe

```
PROCEDURE nom_procédure [(paramètre [, paramètre...])]  
{IS | AS}  
/*déclarations si nécessaires*/  
BEGIN  
--instructions exécutables  
[EXCEPTION  
--Section optionnelle de  
--Gestion des exceptions ]  
END [nom_procédure];
```

- Exemple 1 (procédure sans paramètres)

Enregistrer l'utilisateur courant et la date système dans la table Tab_Trace (User_Uid, User_Name, User_Trace_Date)

~~DECLARE~~

....

~~PROCEDURE~~ RECORD_Trace

As

BEGIN

INSERT INTO Tab_Trace(User_Uid, User_Name, User_Trace_Date)

VALUES (UID, USER, SYSDATE);

EXCEPTION

WHEN OTHERS THEN

DBMS_OUTPUT.PUT_LINE ('erreur déclenchée');

END RECORD_Trace;

-- autres déclarations

BEGIN --début bloc externe

RECORD_Trace;--appel procédure

END;--fin bloc externe

- Exemple 2 (procédure avec paramètres)

Augmenter le salaire d'un employé identifié par EMPNO d'un montant donné

DECLARE

....

~~PROCEDURE~~ AUGMENTE_SAL (P_EMPNO EMP.EMPNO%TYPE, P_Montant NUMBER) IS

BEGIN

UPDATE EMP

SET SAL = SAL + P_Montant

WHERE EMPNO = P_EMPNO

AND P_EMPNO NOT NULL;

IF SQL%NOTFOUND THEN

DBMS_OUTPUT.PUT_LINE ('MAJ Non Effectuée');

ELSE

DBMS_OUTPUT.PUT_LINE ('MAJ Réussie');

END IF

END AUGMENTE_SAL;

-- autres déclarations

BEGIN --début bloc externe

AUGMENTE_SAL (1234,1000);

END;--fin bloc externe

Les paramètres

- Types :
 - Formel
 - D'appel
- Correspondance
 - Par position
 - Fléchée
 - Mixte
- Mode de passage :
 - IN (read-only)
 - Out (write-only)
 - IN/OUT (read-write)

Modes de passage des paramètres

- Nom_param [**IN|OUT|IN OUT**] Type [{:=|DEFAULT} expr]

Paramètre formel



Paramètre d'appel

IN (read-only)

(1) Le sous PG reçoit une copie

OUT (write-only)

(2) Le sous PG fournit un résultat

IN OUT (read-write)

(1)&(2)

IN : Passage par valeur

IN OUT : Passage par @ échange dans les deux sens

OUT : Passage par @ pour retourner un résultat

- Exemple

```
PROCEDURE ModeParam (P_X IN NUMBER, P_Y OUT NUMBER, P_Z  
NUMBER ) IS
```

```
V_Local NUMBER;
```

```
BEGIN
```

```
P_X :=20; -- Affectation illégale
```

```
P_Y est null ici
```

```
P_Y:=10; --légale
```

```
V_local :=P_Y; --illégale (Write only)
```

```
P_Z reçoit la valeur du param d'appel
```

```
P_Z :=P_Z*3;
```

```
END ModeParam;
```

- Teste de la procédure ModeParam

DECLARE

V_X NUMBER := 1;

V_Y NUMBER :=2;

V_Z NUMBER :=3;

BEGIN

ModeParam (V_X,V_Y,V_Z);

DBMS_OUTPUT.PUT_LINE ('V_X : ' || V_X);

DBMS_OUTPUT.PUT_LINE ('V_Y : ' || V_Y);

DBMS_OUTPUT.PUT_LINE ('V_Z : ' || V_Z);

END;

- Test du bloc

V_X : 1

V_Y : 10

V_Z: 9

Procédure PL/SQL terminée avec succès

Fonctions

Une fonction est un sous programme qui retourne une valeur explicitement par la clause RETURN

- SYNTAXE

```
FUNCTION nom_fonction [(paramètre [, paramètre...])]
```

```
RETURN type IS
```

```
    /*déclarations si nécessaires*/
```

```
BEGIN
```

```
    --instructions exécutables
```

```
    --RETURN expression;
```

```
[EXCEPTION
```

```
--Gestionnaires d'exceptions]
```

```
END [nom_fonction] ;
```

- Le type de la fonction peut utiliser %TYPE
- Mêmes modes de passage de paramètres que les procédures

- Exemple

La fonction Avg_Sal retourne le salaire moyen des chercheurs d'un laboratoire

DECLARE

--déclarations si nécessaires

V_Sal_Moy NUMBER (10,3);

FUNCTION Avg_Sal (P_Labno IN Chercheur.Labno%TYPE)

RETURN NUMBER IS

V_Avg_Sal Chercheur.Sal%TYPE;

BEGIN

SELECT AVG(Sal) INTO V_Avg_Sal FROM Chercheur

WHERE Labno=P_Labno;

RETURN V_Avg_Sal;

EXCEPTION

WHEN OTHERS THEN

RAISE_APPLICATION_ERROR (-20110, SUBSTR(SQLERRM, 1, 80));

END AVG_SAL;

--autres déclarations

BEGIN

V_Sal_Moy := Avg_Sal (10);-- Appel fonction

DBMS_OUTPUT.PUT_LINE ('Salaire Moyen : ' || V_Sal_Moy);

SGBD- PL/SQL

END;--fin bloc externe

Les sous- programmes stockés et les packages

Introduction

- La différence entre une procédure stockée et une procédure déclarée et utilisée dans un bloc anonyme est importante. Cette dernière est temporaire. C'est-à-dire qu'elle n'existe plus lorsque l'exécution du bloc anonyme est terminée
- Fonction ou Procédure stockée : sous programme appelable par un utilisateur,

PROCEDURES Stockées

- Syntaxe

CREATE [OR REPLACE] Procedure

Où

Procedure : syntaxe habituelle de déclaration d'une procédure

OR REPLACE : Remplace une procédure existante (suppression + re-création)

SQL> **EXECUTE** raise_salary(7788, 1000);

- Exemple
-

```
CREATE OR REPLACE PROCEDURE raise_salary (p_empno
emp.empno%TYPE, p_montant Number) IS
V_sal_actuel emp.sal%type;
E_sal_NULL EXCEPTION;
BEGIN
SELECT sal INTO v_sal_actuel FROM emp
WHERE empno = p_empno and p_empno is not null;
IF v_sal_actuel IS NULL THEN
RAISE E_sal_NULL;
ELSE
UPDATE emp SET sal = sal + p_montant
WHERE empno = p_empno;
END IF;
EXCEPTION
WHEN NO_DATA_FOUND THEN
.....
WHEN E_sal_NULL THEN
-- instructions exécutables
END raise_salary;
```

- Autres Exemples (1)

```
CREATE OR REPLACE PROCEDURE Affiche_Sal_Moy
```

```
IS
```

```
V_avg_sal emp.sal%type ;
```

```
BEGIN
```

```
SELECT AVG(SAL) INTO v_avg_sal FROM EMP ;
```

```
DBMS_OUTPUT.PUT_LINE (concat ('salaire Moyen :  
' ,v_avg_sal));
```

```
END Affiche_Sal_Moy;
```

```
SQL> EXECUTE affiche_sal_moy;
```

```
Salaire Moyen : 2073,21
```

- Autres Exemples (2)

```
CREATE OR REPLACE PROCEDURE Affiche_Sal_Moy_deptno
(p_deptno dept.deptno%type) IS
V_avg_sal emp.sal%type;
BEGIN
SELECT AVG(SAL) INTO v_avg_sal
FROM EMP
WHERE deptno = p_deptno AND p_deptno is not null;
DBMS_OUTPUT.PUT_LINE('salaire Moyen :'||v_avg_sal);
END Affiche_Sal_Moy_deptno;
```

SQL> EXECUTE Affiche_Sal_Moy_deptno(10);

salaire Moyen :2916,67

- **CREATE PROCEDURE/FUNCTION**
Compile et Stocke dans le Dictionnaire de Oracle
-

- **La vue USER_ERRORS**

Affiche les erreurs de compilation

SQL>SHOW ERRORS (commade SQL*Plus)

- **La vue USER_OBJECTS**

Présente les objets-Oracle de chaque utilisateur

Name	Null?	Type
OBJECT_NAME		VARCHAR2(128)
OBJECT_ID		NUMBER
OBJECT_TYPE		VARCHAR2(13)
CREATED		DATE
STATUS		VARCHAR2(7)

- La vue **USER_SOURCE**

Présente le code source des sous programmes de chaque utilisateur

Name	Null?	Type
NAME	NOT NULL	VARCHAR2(30)
TYPE		VARCHAR2(12)
LINE	NOT NULL	NUMBER
TEXT		VARCHAR2(2000)

Possibilité de récupérer le texte source d'un sous programme stockée :

```
SELECT      line, Text
FROM  USER_SOURCE
WHERE      NAME = 'nom_sous_prog'
AND        TYPE = 'FUNCTION'
ORDER BY    line ;
```

LINE TEXT

```
1 procedure get_emp_ename (
2                               p_empno IN emp.empno%type,
3                               p_ename OUT emp.ename%type)
4 IS
5 BEGIN
6     SELECT ename
7     INTO   p_ename
8     FROM   emp
9     WHERE  empno = p_empno;
10 EXCEPTION
11     WHEN NO_DATA_FOUND THEN
12         DBMS_OUTPUT.PUT_LINE ('employé non trouvé');
13 END get_emp_name;
```

FONCTIONS Stockées

Doit retourner une valeur. **CREATE [OR REPLACE]**
fonction

Où fonction : syntaxe habituelle de déclaration d'une
fonction

- Exemple 1

```
CREATE OR REPLACE FUNCTION LIB_JOUR (p_date
DATE) RETURN VARCHAR2 IS
BEGIN
    RETURN (TO_CHAR(p_date, 'Day'));
EXCEPTION
    When Others Then RETURN (NULL);
END LIB_JOUR;
```

- ① **V_Lib_Jour := Lib_Jour(sysdate) ; -- appel de la fonction**

- Exemple

```
CREATE OR REPLACE FUNCTION SECOND_SAL_DEPTNO (P_DEPTNO
    DEPT.DEPTNO%TYPE)
```

```
RETURN NUMBER IS
```

```
/* DECLARATIONS */
```

```
CURSOR C_EMP (P_DEPTNO DEPT.DEPTNO%TYPE) IS
```

```
SELECT SAL FROM EMP
```

```
WHERE DEPTNO = P_DEPTNO and P_DEPTNO IS NOT NULL
```

```
ORDER BY SAL;
```

```
V_SAL EMPSAL%TYPE DEFAULT NULL;
```

```
BEGIN
```

```
OPEN C_EMP(P_DEPTNO);
```

```
LOOP
```

```
FETCH C_EMP INTO V_SAL;
```

```
EXITWHEN C_EMP%NOTFOUND OR C_EMP%ROWCOUNT=2;
```

```
END LOOP ;
```

```
CLOSE C_EMP;
```

```
RETURN(V_SAL);
```

```
END SECOND_SAL_DEPTNO;
```

FONCTIONS Stockées

- Exemple **Find_Grade** qui retourne le grade de salaire d'un empno s'il existe,-1 , sinon.

SQL> CREATE OR REPLACE function Find_Grade (p_empno in emp.empno%type)

```
2 RETURN number
3 is
4 v_grade salgrade.grade%type;
5 begin
6 select grade
7 into v_grade
8 from emp, salgrade
9 where empno = p_empno and p_empno is not null
10 and sal between losal and hisal;
11 return (v_grade);
12 exception
13 when no_data_found then
14 return(-1);
15 end Find_grade;
```

FONCTIONS Stockées

- Exemple **Find_Grade** qui retourne le grade de salaire d'un empno s'il existe, -1 , sinon.

```
SQL> var H_grade number;
SQL> execute :H_grade :=
  find_grade(7788);--appel de la fonction
SQL> print H_grade
CHEF_Dept
-----
-----
```

Suppression de sous-programmes stockés

- Suppression d'une

FONCTION/PROCEDURE

DROP {PROCEDURE | FUNCTION}
nompf ;

- Remarques

1. CREATE et DROP : Commandes du LDD. !!!
2. Modes de passage de paramètres identiques aux sous-programmes non stockés.

Sous-programmes et priviléges

- **Le privilège EXECUTE**

Le propriétaire d'un s/pg stocké ou d'un package peut autoriser d'autres utilisateurs à s'en servir en leur accordant le privilège EXECUTE :

- **GRANT EXECUTE ON {fonct | proc | pkg} TO username ;**
- **SQL> GRANT EXECUTE ON Lib_Jour TO B;**

Structure d'un package

réunion des fonctions et des procédures stockées

Composé de 2 parties :

- **Package Specification** : déclarative variables globales, Exceptions, curseurs, types prototypes de fonctions et des procédures
 - **Package Body** : Description complète des fonctions/procédures globales
§ BEGIN optionnelle
- Déclaré en 2 phases :

Développement d'un package

```
CREATE OR REPLACE PACKAGE pkg_name
AS
    -- Déclaration variables Globales, Exceptions, Curseurs
    ...
    -- Prototypes des Fonctions / Procédures Globales
    prototype1;
    prototype2;
    ...
    prototype n;
END [pkg_name];
```

Développement d'un package

CREATE OR REPLACE PACKAGE BODY *pkg_name*
AS

-- Corps des Fonctions / Procédures Globales

BEGIN -- optionnelle

Commandes à exécuter

(initialisation de variables par des valeurs calculées)

END [*pkg_name*];

Développement d'un package

- Exemple

```
CREATE OR REPLACE PACKAGE pkg_EMP AS
```

```
-- Déclaration variables Globales
```

```
    G_nb_emp          number(3);
```

```
    G_nb_dept         number(2);
```

```
    G_nb_MANAGERS    number(2);
```

```
    G_avg_sal        number;
```

```
-- Prototypes des Fonctions / Procédures Globales
```

```
    function get_sal(p_empno in emp.empno%type) return emp.sal%type;
```

```
    function get_grade(p_empno in emp.empno%type) return  
        salgrade.grade%type;
```

```
    function get_nb_emp_exceeds_avg_sal return number;
```

```
END pkg_EMP;
```

Développement d'un package

CREATE OR REPLACE PACKAGE BODY Pkg_EMP IS

```
Function get_sal (p_empno in emp.empno%type)
return emp.sal%type    IS
v_sal number ;
BEGIN
IF p_empno is not null THEN
    RAISE NO_DATA_FOUND;          -- ou une exception utilisateu
END IF;
select sal      into v_sal
    from emp
where empno =p_empno;
    return v_sal;
EXCEPTION
when no_data_found then
    raise_application_error (-20000, 'N° Employé ' ||
p_empno || ' Non Trouvé ou non renseigné');
    return null;
END get_sal;
```

Développement d'un package

```
Function get_grade(p_emno in emp.empno%type)
    return salgrade.grade%type           IS
        v_grade salgrade.grade%type;
BEGIN
    select grade               into v_grade
    from   emp , salgrade
    where  empno = p_emno and p_emno is not null
          and  sal between losal and hisal;
return v_grade;
EXCEPTION
when no_data_found then
raise_application_error (-20001,
    'Grade de Salaire non trouvé pour Employé n°'||p_emno
return null;
END get_grade;
```

Développement d'un package

```
Function get_nb_emp_exceeds_avg_sal Return number IS
    v_nb integer;
BEGIN
    select count(*) into v_nb
        from emp      where sal > G_avg_sal;
return v_nb;                                -- utilisation var Globale
EXCEPTION
    when no_data_found then
        raise_application_error (-20001, 'Erreur ');
    return null;
END get_nb_emp_exceeds_avg_sal ;
```

Développement d'un package

```
BEGIN
    -- Initialiser la variable G_nb_emp
    select count(*)      into G_nb_emp
        from emp;
    -- Initialiser la variable G_nb_dept
    select count(*)  into G_nb_dept
        from dept;
    -- Initialiser la variable G_nb_MANAGERS
    select count(*)      into G_nb_MANAGERS
        from emp
        where upper(job) = 'MANAGER';
    -- Initialiser la variable G_avg_sal
    select avg(sal)      into G_avg_sal
        from emp;
END pkg_EMP;
```

APPEL des s/pg d'un package

Préfixer par le nom du package.

Exemple

```
BEGIN  
DBMS_OUTPUT.PUT_LINE ('Salaire : ' ||PKg_emp.get_sal(7788) );  
END;  
ou  
Execute :h_sal := PKg_emp.get_sal(7788);
```

Les variables globales du package peuvent être référencées :

```
BEGIN  
DBMS_OUTPUT.PUT_LINE ('G nb dept : ' ||PKg_emp.g_nb_dept );  
END;
```

Exercice recap1

Soit la table **mois** (**num_mois**, **lib_mois**)

- Ecrire une procédure permettant de remplir la table **mois** avec tous les mois de l'année

Correction Exercice recap1

Create or replace Procedure P_Mois

IS

v_date DATE :='01/01/18';

BEGIN

FOR i IN 1..12 LOOP

INSERT INTO mois (Num_mois, Lib_mois) VALUES (i,
TO_CHAR(v_date, 'MONTH'));

v_Date := ADD_MONTHS(v_date,1);

END LOOP;

END P_Mois;

Exercice recap2

Soit la table **mois (num_mois, lib_mois)**

- Avec sql ajouter une colonne nb_jours
- Ecrire une procédure permettant mettre à jour la colonne nb_jours de cette table avec le nombre de jours de chaque mois.

Correction Exercice recap2

Create or replace Procedure P_nbjours

IS

v_date DATE :='01/01/18';

A NUMBER;

BEGIN

FOR i IN 1..12 LOOP

If (i=1 or i=3 or i=5 or i=7 or i=8 or i=10 or i=12) then

UPDATE mois SET nb-jours=31 where Num_mois=i;

Else If (i=4 or i=6 or i=9 or i=11) then

UPDATE mois SET nb-jours=30 where Num_mois=i;

Else

A:=EXTRACT(YEAR FROM v_date);

If (A MOD 4)=0)Then

UPDATE mois SET nb-jours=29 where Num_mois=2;

Else

UPDATE mois SET nb-jours=28 where Num_mois=2;

End if; End if; END LOOP; END P_Mois;

Les déclencheurs

Introduction

- On appelle déclencheur (trigger) un traitement qui se déclenche suite à un événement.
- Dans une programmation traditionnelle se faisait par un appel de sous programme.
- Il existe deux types de déclencheurs :
 - Des déclencheurs applicatifs créés et manipulés au niveau d'une application
 - Des déclencheurs de bases de données stockés dans le dictionnaire de données du SGBD et associés à des événements sur des tables

Utilité des déclencheurs

- Les déclencheurs sont utiles pour plusieurs fins :
 - Implémenter certaines règles métier du SI de l'organisation (par exemple, MAJ du stock suite à une entrée, déclenchement d'une demande de réapprovisionnement à la suite d'une insuffisance de stock,...)
 - Mettre en œuvre une politique de sécurité complexe (par exemple, interdire des modifications des données à certains utilisateurs, garder trace mises à jour douteuses,...)

Caractéristiques des déclencheurs

- Les Triggers ressemblent aux sous programmes stockés sur les points suivants :
 - Sont des blocs PL/SQL nommés
 - Objets stockés dans Oracle sous forme de code source et p-code (exécutable)
- Mais ils en diffèrent sur les points suivants :
 - Liés à un événement : exécution implicite
 - Ne sont pas paramétrables
 - Peuvent être désactivés, etc ...

Description d'un déclencheur

- Eléments de description

Un trigger est défini à travers les éléments suivants :

Elément	Valeurs possibles	Explication
Nom_Trigger	Choisie par le développeur	Identifiant du déclencheur
Nom_Table	Nom d'une table	La table sur laquelle le trigger est défini. Si l'événement est exécuté sur cette table, le trigger sera déclenché
Evénement	INSERT UPDATE, DELETE	Evénement déclenchant l'exécution du trigger : commande LMD ou une combinaison utilisant OR
Séquencement	BEFORE ou AFTER	Caractérise le <u>séquencement</u> du déclencheur (timing) par rapport à son événement. Avec BEFORE (AFTER), le déclencheur est lancé avant (après) que l'événement déclenchant ne soit exécuté.

Niveaux de déclenchement

Il existe deux niveaux de déclenchement :

1. Niveau instruction (statement level trigger)
2. Niveau ligne (row level trigger)

(1) : un déclencheur de niveau 1 s'exécute une seule fois

Création : la commande de création d'un déclencheur permet sa compilation et son stockage dans la métabase. Après une compilation correcte, le déclencheur est créé à l'état actif (ENABLE)

Création d'un TRIGGER

```
CREATE [ OR REPLACE ] TRIGGER nom_Trigger
{ BEFORE | AFTER } événement
ON nom_Table
DECLARE
-- déclarations variables, curseurs, records, ...
BEGIN
-- traitement
EXCEPTION
-- Gestionnaires d'exceptions
END [nom_Trigger];
```

Déclencheur de Niveau ligne (row level trigger)

Un déclencheur de ligne s'exécute une fois pour chaque n-uplet affecté par l'événement déclenchant.

- Création d'un TRIGGER

CREATE [OR REPLACE] TRIGGER *nom_Trigger*

{ BEFORE | AFTER } *événement*

ON *nom_Table*

FOR EACH ROW [WHEN *condition*]

[REFERENCING {[old [AS] *nom_old*] | New [AS] *nom_new*}]]

DECLARE

-- *déclarations variables, curseurs, records, ...*

BEGIN

- - *traitement*

EXCEPTION

- - *Gestionnaires d'exceptions*

END [*nom_Trigger*];

For each row	Clause optionnelle, si utilisée le déclencheur spécifié sera de niveau ligne. Autrement il est de niveau instruction
Condition	Option de la clause For each Row (valable uniquement pour les triggers de niveau ligne), introduit une condition selon laquelle le corps du déclencheur peut être exécuté ou simplement abandonné
Referencing	Permet de changer les noms standards des pseudo records OLD et NEW par de nouveaux noms nom_old et nom_new

- Exemples
1. Ecrire un trigger de BD qui affiche un message à la suite de la suppression d'un dept

Choix des composants du trigger				
Nom	Evénement	Séquencement	Niveau	Table
TRG_DEL_DEPT	DELETE	BEFORE (ou AFTER)	Ligne	DEPT

```
CREATE OR REPLACE TRIGGER TRG_DEL_DEPT  
AFTER DELETE          -- ou bien BEFORE  
ON DEPT  
FOR EACH ROW           -- de niveau n-uplet  
BEGIN  
DBMS_OUTPUT.PUT_LINE('TRG0 : Dépt supprimé');  
END TRG_DEL_DEPT;
```

Test :

```
SQL> delete from dept  
      where deptno NOT IN (Select deptno from emp);  
TRG0 : Dépt supprimé  
1 ligne(s) supprimée(s).
```

Déclencheur de Niveau Instruction

- Remarques
 - Comme il s'agit d'un 'Row level Trigger' le corps du déclencheur sera exécuté avec chaque n-uplet affecté par l'événement déclenchant (DELETE). Si aucune ligne n'est affectée alors aucun message affiché
 - Utiliser un trigger de niveau instruction pour exécuter le corps du déclencheur une seule fois quelque soit le nombre de lignes affectées (même 0). Si aucune ligne n'est affectée alors le message sera affiché.

- Exemple de Trigger de niveau instruction

```
CREATE OR REPLACE TRIGGER TRG_DEL_DEPT_Niv_Ins
AFTER DELETE      -- ou bien BEFORE
ON DEPT
BEGIN
DBMS_OUTPUT.Put_Line('TRG_DEL_DEPT_Niv_Ins Déclenché');
END TRG_DEL_DEPT_Niv_Ins;
```

Test du trigger :

```
SQL> delete from dept where
deptno is null;
```

TRG_DEL_DEPT_Niv_Ins Déclenché

Les Pseudo-records : old et : new

- Un trigger de niveau ligne se déclenche une fois avec chaque n-uplet traité par l'événement du trigger
- A l'intérieur d'un Trigger de niveau ligne, on a souvent besoin d'accéder aux données du n-uplet en cours de manipulation. PL/SQL le permet par le biais des 2 pseudo-records **:old** et **:new** de même structure que la table sur laquelle le trigger est défini.

Evénement	OLD	NEW
INSERT	Tous les champs sont nulls	Valeurs à insérer en provenance de la commande INSERT qui a déclenché le trigger
UPDATE	Valeurs originales du n-uplet avant UPDATE	Nouvelles valeurs après UPDATE
DELETE	Valeurs Originales du n-uplet avant DELETE	

- Non valables pour les triggers de niveau instruction.
- Utilisables dans les curseurs implicites et explicites.

- Exemple
1. Ecrire un trigger qui enregistre dans la table ARCHIVE_DEPT chaque dept. Supprimé.

Nom	Événement	Séquencement	Niveau	Table
TRG_Archive_Dept	DELETE	BEFORE	Ligne	DEPT

```

CREATE OR REPLACE TRIGGER TRG_Archive_Dept
BEFORE DELETE
ON DEPT
FOR EACH ROW
BEGIN
INSERT INTO ARCHIVE_DEPT (DEPTNO, Dname, Loc)
VALUES (:old.deptno, :old.dname, :old.loc);
EXCEPTION
    When OTHERS THEN
        RAISE_APPLICATION_ERROR(-20001,'Erreur
dans Trigger Before Delete On DEPT') ;
END TRG_Archive_Dept;

```

- **Exemple**

Ecrire un trigger de BD qui à la suite d'une modification de salaire enregistre dans la table AUDIT_SAL (Utilisateur, date_maj, Empno, anc_sal, nouv_sal)

```
CREATE OR REPLACE TRIGGER TRG_Audit_Sal
AFTER UPDATE OF SAL
ON EMP
FOR EACH ROW
BEGIN
INSERT INTO AUDIT_SAL VALUES
(user,sysdate, :old.empno,          :old.sal, :new.sal);
EXCEPTION
    When others then null;
END ;
```

```
SQL> update emp  
      set sal= sal + 1000  
     where deptno= 10;
```

3 ligne(s) mise(s) à jour.

```
SQL> select * from audit_sal;
```

UTILISATE	DATEMAJ	EMPNO	ANC_SAL	NOUV_SAL
-----	-----	-----	-----	-----
SCOTT	18/04/02	7782	2450	3450
SCOTT	18/04/02	7839	5000	6000
SCOTT	18/04/02	7934	1300	2300

La clause WHEN «condition»

Cette clause est valide uniquement pour les row-level triggers. Si utilisée, la condition sera évaluée pour chaque ligne affectée par l'événement du trigger, et le corps du trigger n'est exécuté pour cette ligne que si l'expression logique est vraie.

- Les pseudo-record **old** et **new** sont autorisés dans la clause WHEN mais sans le préfixe (:)

- **Exemple 1**

```
CREATE OR REPLACE TRIGGER TRG_B_IU_EMP
BEFORE INSERT OR UPDATE of deptno
ON EMP
FOR EACH ROW
WHEN (new.deptno = 40)
BEGIN
    Update      dept
    Set         nb_emp = nvl(Nb_emp,0) + 1
    Where       deptno = :new.deptno;
ENDTRG_B_U_EMP;
```

Test du trigger :

```
SQL>      update      emp  
          set      deptno      = 40  
          where      deptno      = 10;
```

3 ligne(s) mise(s) à jour.

```
SQL> select * from dept where deptno = 40;
```

DEPTNO	DNAME	LOC	NB_EMP
40	OPERATIONS	BOSTON	3

- **Exemple 2**

Modification systématique de la commission pour la fonction SALESMAN : Augmenter leur commission en fonction de leur augmentation du salaire.

```
CREATE OR REPLACE TRIGGER TRG_B_Usal
BEFORE UPDATE OF SAL
ON EMP
FOR EACH ROW
WHEN (upper(new.job) = 'SALESMAN')
BEGIN
IF nvl(:old.sal,0)>0 THEN
:new.comm := :old.comm *(:new.sal/:old.sal);
END IF;
ENDTRG_B_Usal;
```

Test du trigger :

EMPNO	SAL	COMM	JOB
-------	-----	------	-----

7499	1600	300	SALESMAN
------	------	-----	----------

UPDATE EMP

SET SAL = SAL+320 -- 320 = 20% de sal

WHERE EMPNO = 7499;

EMPNO SAL COMM

7499	1760	360
------	------	-----

- **Remarques :**

- La clause WHEN condition équivaut un test portant sur tout le corps du trigger
- Ainsi, TRG_B_UI_EMP peut être réécrit comme suit :

```
CREATE OR REPLACE TRIGGER TRG_B_IU_EMP
BEFORE INSERT OR UPDATE of deptno
ON EMP
FOR EACH ROW
BEGIN
IF (:new.deptno = 40) THEN
/* corps du trigger */
END IF;
ENDTRG_B_U_EMP;
```

- **Exemple**

Interdire les modifications sur la table EMP en dehors des horaires de travail supposés entre 8h à 17h.

```
CREATE OR REPLACE TRIGGER TRG_Grant_Emp
BEFORE UPDATE
ON EMP
DECLARE
  E_ACCES_DENIED EXCEPTION ;
BEGIN
  If (to_number (to_char(sysdate,'HH24')) ) NOT between 8 and 17)
  THEN
    RAISE E_ACCES_DENIED ;
  End IF ;
EXCEPTION
  WHEN E_ACCES_DENIED THEN
    RAISE_APPLICATION_ERROR (-20001, 'Accès interdit en dehors des
horaires de travail') ;
END;
```

Gestion des déclencheurs

- **DROP TRIGGER triggerName ;**
- **Activation / Désactivation d'un trigger**

Parfois il est préférable de désactiver un trigger que de le supprimer en vue de le réactiver ultérieurement.

- **ALTER TRIGGER triggerName { ENABLE |DISABLE } ;**

A la création un trigger est activé

Un trigger désactivé (DISABLE) continue à exister dans le DD.

- **Affichage des erreurs de compilation**

SQL>SHOW ERRORS

- **Activation/Désactivation de Tous les Triggers d'une table**

ALTER TABLE TableName { ENABLE | DISABLE } ALL TRIGGERS ;
SGBD- PL/SQL