

Bases de données avancées



LANGUAGE PL/SQL

Niveau:

Licence SIR S5

Bases de données avancées

2

INTRODUCTION

Généralités



- PL/SQL: Procedural Language extensions to SQL.
- Objectif : bonne intégration du langage avec SQL.
- PL/SQL n'existe pas comme un langage autonome ; il est utilisé à l'intérieur d'autres produits Oracle.
- Permet des interactions avec une base SQL.
- Langage de programmation qui inclut des ordres SQL

Pourquoi PL/SQL

4

- SQL est un langage non procédural.
- Les traitements complexes sont parfois difficiles à écrire si on ne peut utiliser des variables et les structures de programmation comme les boucles et les alternatives.
- On ressent vite le besoin d'un langage procédural pour lier plusieurs requêtes SQL avec des variables et dans les structures de programmation habituelles.

Pourquoi PL/SQL (2)

5

- Les contraintes prédéfinies ne sont pas toujours suffisantes.

Exemple : tout nouveau prix pour un CD doit avoir une date de début supérieure à celle des autres prix pour ce CD.

- L'insertion, la suppression ou la mise à jour de certaines données peut nécessiter des calculs sur la base.
- Utilisation de fonction propres à l'application dans des requêtes.

Caractéristiques du PL/SQL

6

- Extension de SQL : des requêtes SQL cohabitent avec les structures de contrôle habituelles de la programmation structurée (blocs, alternatives, boucles).
- La syntaxe ressemble au langage Ada et proche de Pascal.
- Un programme est constitué de procédures et de fonctions.
- Des variables permettent l'échange d'information entre les requêtes SQL et le reste du programme.

Utilisation de PL/SQL



- PL/SQL peut être utilisé pour l'écriture des procédures stockées et des triggers (Oracle accepte aussi le langage Java).
- Il convient aussi pour écrire des fonctions utilisateurs qui peuvent être utilisées dans les requêtes SQL (en plus des fonctions prédéfinies).
- Il est aussi utilisé dans des outils Oracle, *Forms et Report en particulier*.

PL/SQL

8

- PL/SQL est un langage structuré en blocs, constitués d'un ensemble d'instructions. C'est un langage de programmation à la fois puissant, simple et moderne.

Comparaison avec SQL

9

- **SQL:**
 - Langage assertionnel et non procédural.
- **Pl/Sql:**
 - Langage procédural qui intègre des ordres sql
 - ✦ Select, insert, update, delete
 - Langage à part entière comprenant:
 - ✦ Définition de variables, constantes, expressions, affectations.
 - ✦ Traitement conditionnels, répétitifs.
 - ✦ Traitement de curseurs.
 - ✦ Traitement des erreurs et d'exceptions.
 - ✦ Etc...

Avantages du PL/SQL

10

PL/SQL offre de nombreux avantages:

- Intégration parfaite du SQL
- Support de la programmation orientée objet
- Très bonne performance
- Portabilité
- Facilité de programmation

Objectifs du chapitre

11

A la fin de ce chapitre, vous saurez:

- Identifier les différentes parties d'un bloc PL/SQL
- Spécifier des variables PL/SQL
- Déclarer des variables intégrées PL/SQL
- Déclarer des variables typées dynamiquement
- Exécuter un bloc PL/SQL
 - ✦ Avec ou sans accès à la base

Bases de données avancées

12

STRUCTURE D'UN PROGRAMME PL/SQL

Blocs

13

- Un programme est structuré en blocs d'instructions de 3 types :
 - procédures anonymes
 - procédures nommées
 - fonctions nommées
- Un bloc peut contenir d'autres blocs

Structure des blocs

14

- Une section facultative de déclaration et initialisation de types, variables et constantes
- Une section obligatoire contenant les instructions d'exécution
- Une section facultative de gestion des erreurs

Structure des blocs (2)

15

DECLARE

-- définitions de variables

BEGIN

-- Les instructions à exécuter

EXCEPTION

-- La récupération des erreurs

END;

Structure des blocs (3)

16

Les blocs PL/SQL peuvent être imbriqués:

```
DECLARE
...
BEGIN
  DECLARE
    ....
    BEGIN
      ....
      BEGIN
        .....
        END ;
      .....
    END ;
  .....
END ;
```


Structure des blocs (3)

17

- Les blocs comme les instructions se terminent par un ‘;’.
- Seuls ‘Begin’ et ‘End’ sont obligatoire.

Bases de données avancées

18

LES VARIABLES

Les variables

19

- Identificateurs Oracle :
 - 30 caractères au plus
 - commence par une lettre
 - peut contenir lettres, chiffres, __, \$ et #
- Pas sensible à la casse.
- Portée habituelle des langages à blocs.
- Doivent être déclarées avant d'être utilisées.

Les variables (2)

20

- Déclaration :

Nom_variable type_variable;

- Initialisation:

Nom_variable := valeur;

- Déclaration et initialisation :

Nom_variable type_variable := valeur;

Les variables

21

- **var [CONSTANT] datatype [NOT NULL] [:= | DEFAULT expr];**
- Adopter des conventions pour nommer des objets.
- Initialiser les constantes et les variables déclarées **NOT NULL**.
- Initialiser les identifiants en utilisant l'opérateur d'affectation (**:=**) ou le mot réservé **DEFAULT**.
- Déclarer au plus un identifiant par ligne.
- Le type peut être primitif ou objet.

Commentaires

22

- **-- Pour une fin de ligne**
- **/* Pour plusieurs lignes */**

Les types de variables

23

- **VARCHAR2**

- Longueur maximale : 32767 octets
- Syntaxe:

Nom_variable VARCHAR2(30);

Exemple: name VARCHAR2 (30) ;

 name VARCHAR2 (30) := 'toto' ;

- **NUMBER**

Nom_variable NUMBER(long,dec);

avec Long : longueur maximale

 Dec : longueur de la partie décimale

Exemple: num_tel number (10) ;

 toto number (5,2) =142.12 ;

Les types de variables (2)

24

- **DATE**

Nom_variable DATE;

- Par défaut DD-MON-YY (18-DEC-02)
- Fonction TO_DATE

Exemple :

```
start_date := to_date('29-SEP-2003','DD-MON-YYYY');  
start_date := to_date('29-SEP-2003:13:01','DD-MON-  
YYYY:HH24:MI');
```

- **BOOLEAN**

- TRUE, FALSE ou NULL

Les types de variables (3)

25

- identificateur [CONSTANT] type [:= valeur];
- Exemples :
 - **age integer;**
 - **nom varchar(30);**
 - **dateNaissance date;**
 - **ok boolean := true;**
- Déclarations multiples **interdites** :
 - **i, j integer;**

Exemples

26

```
c          CHAR( 1 );
name       VARCHAR2(10) := 'Scott';
cpt        BINARY_INTEGER := 0;
total      NUMBER( 9, 2 ) := 0;
order      DATE := SYSDATE + 7;
Ship       DATE;
pi         CONSTANT NUMBER ( 3, 2 ) := 3.14;
done       BOOLEAN NOT NULL := TRUE;
ID         NUMBER(3) NOT NULL := 201;
PRODUIT    NUMBER(4) := 2*100;
V_date     DATE := TO_DATE('17-OCT-01', 'DD-MON-YY');
V1         NUMBER := 10;
V2         NUMBER := 20;
V3         BOOLEAN := (v1>v2);
Ok         BOOLEAN := (z IS NOT NULL);
```

Quelques conventions en PL/SQL

27

- Deux variables peuvent partager le même nom si elles sont dans des portées distinctes.
- Les noms des variables doivent être différents des noms des colonnes des tables utilisées dans un bloc:
 - `v_empno` (variable)
 - `g_deptno` (globale)
 - `c_emp` (CURSOR)
- L'identifiant est limité à 30 caractères, le premier caractère devant être une lettre.

Utilisation des variables

28

- On utilise des variables pour :
 - Le stockage temporaire de données
 - La manipulation de valeur stockées
 - La possibilité de les réutiliser
 - Simplifier la maintenance du code
 - Variable typée dynamiquement au moyen d'attributs spéciaux
 - ✦ **%ROWTYPE** ou **%TYPE**
- Les variables Abstract Data Type et les collections seront abordées ultérieurement
 - Oracle9i est également un SGBD orienté objet

Bloc PL/SQL Anonyme accédant à la base

29

```
SQL> SET SERVEROUTPUT ON
```

```
SQL> DECLARE
```

```
    v_name VARCHAR2(10);
```

Affectation

```
BEGIN
```

```
    SELECT ename INTO v_name
```

```
    FROM emp; -- WHERE empno=7839;
```

```
    DBMS_OUTPUT.PUT_LINE(v_name);
```

```
EXCEPTION
```

```
    WHEN OTHERS
```

```
        THEN NULL;
```

```
END;
```

```
/
```

```
PL/SQL Procedure successfully completed.
```

Cas à plusieurs variables hôtes

30

```
SET SERVEROUTPUT ON
```

```
SQL> DECLARE
```

```
    v_ename      VARCHAR2 (12) ;
```

```
    v_sal        NUMBER (7,2) ;
```

```
BEGIN
```

```
    SELECT      ename, sal INTO
```

```
               v_ename, v_sal
```

```
    FROM emp WHERE ROWNUM = 1 ;
```

```
    DBMS_OUTPUT.PUT_LINE (v_ename) ;
```

```
    DBMS_OUTPUT.PUT_LINE (v_sal) ;
```

```
EXCEPTION
```

```
    WHEN OTHERS
```

```
    THEN NULL;
```

```
END ;
```

```
/
```

Optionnel à
ce stade

Bases de données avancées

Variables typées dynamiquement

Les types de variables (4)

32

- Types composites adaptés à la récupération des colonnes et lignes des tables SQL :
%TYPE, %ROWTYPE.

Typage dynamique %TYPE

33

- Déclarer une variable à partir :
 - D'une autre variable déjà déclarée
 - D'une définition d'un attribut de la base de données
- Préfixer %TYPE avec :
 - La table et la colonne de la base de données
 - Le nom de la variable déclarée précédemment
- PL/SQL évalue le type de donnée et la taille de la variable.
- Inspiré du langage ADA

Déclaration %TYPE

34

- On peut déclarer qu'une variable est du même type qu'une colonne d'une table ou d'une vue (ou qu'une autre variable) :

nom emp.name.%TYPE;

L'Attribut %TYPE - Exemple

35

```
DECLARE
```

```
    ename            scott.emp.ename%TYPE;  
    job              emp.job%TYPE;  
    balance          NUMBER( 7, 2 );  
    mini_balance     balance%TYPE := 10;  
    rec              emp%ROWTYPE
```

- Le type de données de la colonne peut être inconnu.
- Le type de données de la colonne peut changer en exécution.
- Facilite la maintenance.

Variable typée dynamiquement

36

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
    v_ename                                emp.ename%TYPE;
BEGIN
    SELECT  ename
    INTO    v_ename
    FROM    emp
    WHERE   ROWNUM = 1;
    DBMS_OUTPUT.PUT_LINE(v_ename);
EXCEPTION
    WHEN OTHERS
    THEN NULL;
END;
/
```

Déclaration %ROWTYPE

37

- Une variable peut contenir toutes les colonnes d'une ligne d'une table:

```
employe emp%ROWTYPE;
```

- déclare que la variable employe contiendra une ligne de la table emp.

Exemple d'utilisation

38

```
employee emp%ROWTYPE;  
nom emp.name.%TYPE;  
select * INTO employee  
from emp  
where matr = 900;  
nom := employee.name;  
employee.dept := 20;  
-----  
insert into emp  
values employee;
```

Attribut %TYPE & %ROWTYPE

39

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
```

```
    rec  emp%ROWTYPE;
```

```
    address VARCHAR2(64);
```

```
    income emp.sal%TYPE; -- rec.sal%TYPE;
```

```
BEGIN
```

```
SELECT *
```

```
    INTO rec FROM emp
```

```
    WHERE ROWNUM = 1;
```

```
    income := rec.sal*12;
```

```
    address := rec.ename || CHR(10) ||
```

```
        income || CHR(10) ||
```

```
        TO_CHAR(rec.hiredate,'DD/MM/YYYY');
```

```
    DBMS_OUTPUT.PUT_LINE(address);
```

```
END;
```

```
/
```

```
SMITH
```

```
9600
```

```
17/12/1980
```

Manipulation dans une
variable PL

Affectation

40

- Plusieurs façons de donner une valeur à une variable:
 - `:=`
 - par la directive INTO de la requête SELECT
- Exemples :
 - `dateNaissance := '10/10/2004';`
 - `select name INTO nom from emp where matr = 509;`

Les opérateurs

41

PL/SQL supporte les opérateurs suivants :

- Arithmétique : $+$, $-$, $*$, $/$,
- Concaténation : $||$
- Parenthèses (contrôle des priorités entre opérations): $()$
- Comparaison : $=$, \neq , $<$, $>$, \leq , \geq , **IS NULL**, **LIKE**, **BETWEEN**, **IN**
- Logique : **AND**, **OR**, **NOT**
- Affectation: $:=$

Portée

42

- Les instructions peuvent être imbriquées là où les instructions exécutables sont autorisées.
- La section **EXCEPTION** peut contenir des blocs imbriqués.
- Les boucles possèdent chacune une portée
 - les incréments y sont définis
- Un identifiant est visible dans les régions où on peut référencer cet identifiant :
 - Un bloc voit les objets du bloc de niveau supérieur.
 - Un bloc ne voit pas les objets des blocs de niveau inférieur.

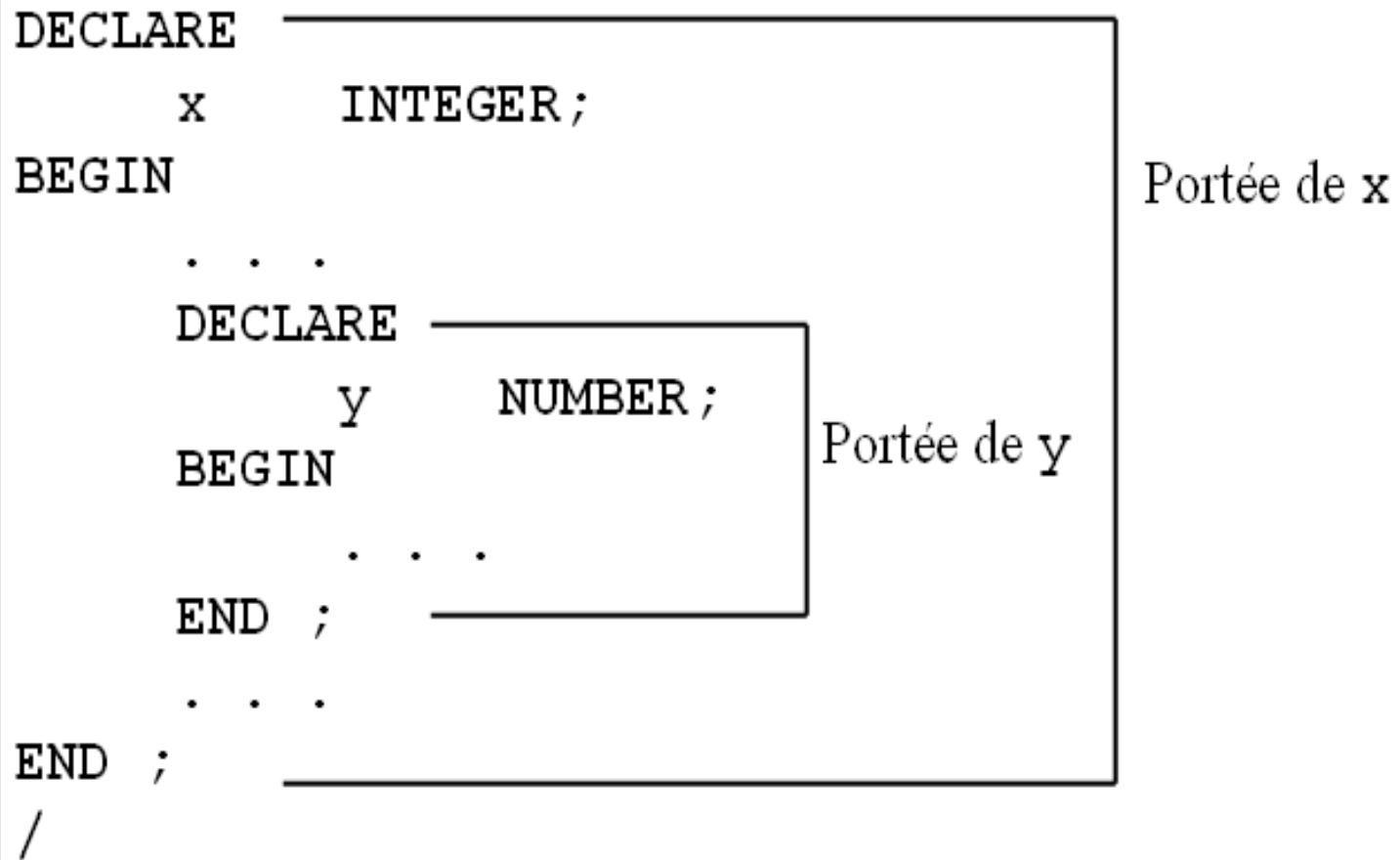
Portée

43

```
DECLARE
    x    INTEGER;
BEGIN
    . . .
    DECLARE
        y    NUMBER;
    BEGIN
        . . .
    END ;
    . . .
END ;
/
```

Portée de x

Portée de y



Règles générales

44

- Commenter le code
- Adopter une convention de casse
- Développer une convention d'appel pour les identifiants et autres objets
- Indenter le code

Instruction SQL en majuscules

Mots-clés en majuscules

Type de données en majuscules

Identifiant et paramètres en minuscules

Tables & colonnes en minuscules

Conventions d'appel possible

45

Identifiant	Nom	Exemple
Variable	v_name	v_sal
Constante	c_name	c_pi
Cursor	name_cursor	Emp_cursor
Exception	e_name	E_too_many
Table type	name_table_type	sum_table_type
Table	name_table	emp_tot_table
Record Type	name_record_type	emp_record_type
Record	name_record	emp_record
Substitution	p_name	p_sal
Globale	g_name	g_deptno

Bases de données avancées

46

STRUCTURES DE CONTRÔLE

Les branchements conditionnels

47

```
IF condition  
THEN  
instructions;  
END IF;
```

```
IF condition  
THEN  
instructions1;  
ELSE  
instructions2;  
END IF;
```

```
IF condition1 THEN  
instructions1;  
ELSEIF condition2 THEN  
instructions2;  
ELSEIF ...  
...  
ELSE  
instructionsN;  
END IF;
```

Les branchements conditionnels (2)

48

• Exemples:

```
IF l_date > '11-APR-03' THEN  
    l_salaire := l_salaire * 1.15;  
END IF;
```

```
IF l_date > '11-APR-03' THEN  
    l_salaire := l_salaire * 1.15;  
ELSE  
    l_salaire := l_salaire * 1.05;  
END IF;
```

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


Choix

49

```
CASE expression  
WHEN expr1 THEN instructions1;  
WHEN expr2 THEN instructions2;  
...  
ELSE instructionsN;  
END CASE;
```

- expression peut avoir n'importe quel type simple

Les itérations

50

- PL/SQL offre la possibilité d'écrire des boucles à l'aide d'instructions LOOP, WHILE ou FOR.

Les itérations (2)

51

1ere forme:

```
LOOP  
    instructions  
EXIT WHEN expression  
END LOOP ;
```

2eme forme:

La boucle **FOR** permet des boucles sur un ensemble fini comme un ensemble d'entiers ou un ensemble d'enregistrements issus d'une requête.

```
FOR i IN 1..10 LOOP  
    DBMS_OUTPUT . PUT_LINE ( 'iteration ' || i ) ;  
END LOOP;
```

Les itérations (3)

52

3eme forme:

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

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

Les itérations (4)

53

- 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;
```

Affichage

54

- Activer le retour écran
 - `set serveroutput on size 10000`
- Affichage
 - `dbms_output.put_line(chaine);`
 - Utilise `||` pour faire une concaténation

Affichage (2)

55

```
DECLARE
compteur number(3);
i number(3);
BEGIN
select count(*) into compteur from clients;
FOR i IN 1..compteur LOOP
    dbms_output.put_line('Nombre : ' || i );
END LOOP;
END;
```

Affichage (3)

56

```
DECLARE
    i number(2);
BEGIN
    FOR i IN 1..5 LOOP
        dbms_output.put_line('Nombre : ' || i );
    END LOOP;
END;
```


Exemple de bloc PL Macro

57

```
SET SERVEROUTPUT ON
DECLARE
    x VARCHAR (5) ;
    y VARCHAR2 (5) ;
BEGIN
    x := 'toto' ; y := 'toto' ;
    IF ( x = y ) THEN
        DBMS_OUTPUT.PUT_LINE('equals') ;
    ELSE
        DBMS_OUTPUT.PUT_LINE('e not equals') ;
    END IF ;
    DBMS_OUTPUT.PUT_LINE(LENGTH(x)) ;
    DBMS_OUTPUT.PUT_LINE(LENGTH(y)) ;
END ;
/
```

Macro intégrée

Autre exemple de macro: CONCAT

58

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
    nom          VARCHAR2 (10) ;
    salaire      NUMBER (7,2) ;
BEGIN
    SELECT ename, sal INTO nom, salaire
    FROM emp WHERE ROWNUM = 1;
    DBMS_OUTPUT.PUT_LINE (nom || salaire) ;
    DBMS_OUTPUT.PUT_LINE (
        CONCAT (nom, salaire) ) ;
END ;
/
```

Une autre façon de concaténer des chaînes de caractères

Un bloc peut contenir plusieurs requêtes

59

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE

        nom                VARCHAR2(10) ;
        salaire            NUMBER(7,2) ;

BEGIN

        SELECT ename INTO nom
        FROM emp WHERE ROWNUM = 1;
        DBMS_OUTPUT.PUT_LINE(nom) ;

        SELECT sal INTO salaire
        FROM emp WHERE ROWNUM = 1;
        DBMS_OUTPUT.PUT_LINE(salaire) ;

EXCEPTION
        WHEN OTHERS
                THEN NULL;

END ;

/
```

Variables de substitution

60

- Passage de paramètres en entrée d'un bloc PL/SQL:
 - Identification du paramètre par le symbole & préfixé au nom de la variable de substitution
 - Pas de déclaration de la variable à faire
 - Directive SQL*Plus ACCEPT si on veut afficher un message d'invite (sinon, SQL*Plus demandera simplement la valeur du paramètre identifiée par son nom)

```
ACCEPT s_number PROMPT 'Entrez le numero du vol : '  
DECLARE TYPE Type_vol is RECORD (num VARCHAR(10),  
depart DATE, arrivee DATE);  
vol_particulier Type_vol;  
BEGIN  
vol_particulier.num := '&s_number' ;  
END;
```

Bases de données avancées

61

LES CURSEURS

Introduction

62

- Manipulation de tuples:
 - Les directives INSERT INTO, UPDATE et DELETE FROM peuvent être utilisées sans restriction avec des variables PL/SQL (scalaires, %ROWTYPE)
- Lecture de tuples Chargement d'une variable à partir de la lecture d'un unique enregistrement dans la base (exception si 0 ou plusieurs enregistrements en réponse)

```
DECLARE
heure_depart Vols.depart%TYPE;
BEGIN
SELECT Vols.depart INTO heure_depart
FROM Vols WHERE Vols.id = 'AF3517' ;
END;
```

Pourquoi utiliser les curseur

63

- Les instructions de type `SELECT ... INTO ...` manquent de souplesse, elles ne fonctionnent que sur des requêtes retournant une et une seule valeur. Ne serait-il pas intéressant de pouvoir placer dans des variables le résultat d'une requête retournant plusieurs lignes ?

Fonctionnalités

64

- Toutes les requêtes SQL sont associées à un curseur.
- Ce curseur représente la zone mémoire utilisée pour *parser et exécuter la requête*.
- Le curseur peut être implicite (pas déclaré par l'utilisateur) ou explicite.
- Les curseurs explicites servent à retourner plusieurs lignes avec un select.

Attributs des curseurs

65

- Tous les curseurs ont des attributs que l'utilisateur peut utiliser:
 - `n%ROWCOUNT` : nombre de lignes traitées par le curseur
 - `n %FOUND` : vrai si au moins une ligne a été traitée par la requête ou le dernier fetch
 - `n%NOTFOUND` : vrai si aucune ligne n'a été traitée par la requête ou le dernier fetch
 - `n %ISOPEN` : vrai si le curseur est ouvert (utile seulement pour les curseurs explicites)

Curseur implicite

66

- Les curseurs implicites sont tous nommés SQL
- Exemple:

```
DECLARE
nb_lignes integer;
BEGIN
delete from emp
where dept = 10;
nb_lignes := SQL%ROWCOUNT;
...
```

Déclaration des curseurs

67

- Un curseur se déclare dans une section DECLARE :

```
CURSOR / nomcurseur / IS / r e q u e t e / ;
```

- Exemple:

```
CURSOR emp_cur IS  
    SELECT FROM EMP ;
```

Ouverture

68

- Lors de l'ouverture d'un curseur, la requête du curseur est évaluée, et le curseur contient toutes les données retournées par la requête. On ouvre un curseur dans une section BEGIN :

```
OPEN / nomcurseur / ;
```

- Exemple:

```
DECLARE  
CURSOR emp_cur IS  
SELECT FROM EMP ;  
BEGIN  
OPEN emp_cur ;  
/ Utilisation du curseur /  
END;
```

Lecture d'une ligne

69

- Une fois ouvert, le curseur contient toutes les lignes du résultat de la requête. On les récupère une par une en utilisant le mot-clé **FETCH** :

```
FETCH / nom curseur / INTO / list e variables / ;
```

- La liste de variables peut être remplacée par une structure de type `nom curseur%ROWTYPE`. Si la lecture de la ligne échoue, parce qu'il n'y a plus de ligne à lire, l'attribut `%NOTFOUND` prend la valeur vrai.

Lecture d'une ligne (2)

70

```
DECLARE
CURSOR emp_cur IS
SELECT FROM EMP ;
ligne emp_cur%rowtype
BEGIN
OPEN emp_cur ;
LOOP
FETCH emp_cur INTO ligne ;
EXIT WHEN emp_cur%NOTFOUND ;
DBMS_OUTPUT . PUT_LINE ( ligne . ename ) ;
END LOOP ;
/.../
END;
```

Fermeture du curseur

71

- Après utilisation, il convient de fermer le curseur.

```
CLOSE / nomcurseur / ;
```

- Complétons l'exemple:

```
DECLARE  
CURSOR emp_cur IS  
SELECT FROM EMP ;  
ligne emp_cur%rowtype ;  
BEGIN  
OPEN emp_cur ;  
LOOP  
FETCH emp_cur INTO ligne ;  
EXIT WHEN emp_cur%NOTFOUND ;  
DBMS_OUTPUT.PUT_LINE ( ligne . ename ) ;  
END LOOP ;  
CLOSE emp_cur ;  
END;  
/
```

Boucle For

72

- Il existe une boucle FOR se chargeant de l'ouverture, de la lecture des lignes du curseur et de sa fermeture.

```
FOR ligne IN emp_cur LOOP  
/ Traitement /  
END LOOP ;
```


Boucle For: exemple

73

```
DECLARE
CURSOR emp_cur IS
SELECT FROM EMP ;
ligne emp_cur%ROWTYPE ;
BEGIN
FOR ligne IN emp_cur LOOP
DBMS_OUTPUT.PUT_LINE ( ligne . ename ) ;
END LOOP ;
END;
/
```

Curseurs paramétrés

74

- Définition: Un curseur paramètre est un curseur dont la requête contient des variables dont les valeurs ne seront fixées qu'à l'ouverture.
- Déclaration: On précise la liste des noms et des type des paramètres entre parenthèses après le nom du curseur :

```
CURSOR / nom / ( / liste des parametres / ) IS  
/ r e q u e t e /
```

Curseurs paramétrés: Exemple

75

- Créons une requête qui, pour une personne donnée, nous donne la liste des noms et prénoms de ses enfants :

```
CURSOR enfants ( numparent NUMBER) IS  
SELECT  
FROM PERSONNE  
WHERE pere = numparent  
OR mere = numparent ;
```

Curseurs paramétrés: Ouverture

76

- On ouvre un curseur paramètre en passant en paramètre les valeurs des variables :

```
OPEN / nom / ( / l i s t e des parametres / )
```

- Exemple:

```
OPEN enfants ( 1 ) ;
```

Curseurs paramétrés: Boucle for

77

- La boucle pour se charge de l'ouverture, il convient donc de placer les paramètre dans l'entete de la boucle,

```
FOR / variable / IN / nom / ( / liste parametres / ) LOOP  
/ instructions /  
END LOOP ;
```

- Exemple:

```
FOR e IN enfants ( 1 ) LOOP  
DBMS_OUTPUT . PUT_LINE ( e . nompers jj ' ' jj e . prenompers ) ;  
END LOOP ;
```

Exemple récapitulatif

78

```
DECLARE
CURSOR parent IS
SELECT
FROM PERSONNE ;
p parent%rowtype ;
CURSOR enfants ( numparent NUMBER) IS
SELECT
FROM PERSONNE WHERE pere = numparent
OR mere = numparent ;
e enfants%rowtype ;
BEGIN
FOR p IN parent LOOP
DBMS_OUTPUT.PUT_LINE ( ' Les enf ant s de ' || p . prenom || ' ' ||p . nom ||' sont : ' ) ;
FOR e IN enfants ( p . numpers ) LOOP
DBMS_OUTPUT.PUT_LINE ( ' * ' || e . Prenom || ' ' ||e . nom ) ;
END LOOP ;
END LOOP ;
END;
/
```

Bases de données avancées

79

COLLECTIONS ET ENREGISTREMENTS

Les collections

80

- Une collection est un ensemble ordonné d'éléments de même type.
- Elle est indexée par une valeur de type numérique ou alphanumérique.
- Elle ne peut avoir qu'une seule dimension (mais en créant des collections de collections on peut obtenir des tableaux à plusieurs dimensions).
- Il s'agit d'un concept général qui englobe des listes, des tableaux et d'autres types de données familières. Chaque élément a un indice unique qui détermine sa position dans la collection.

Les collections

81

On peut distinguer trois types différents de collections :

- Les tables (**INDEX-BY TABLES**) qui peuvent être indicées par des variables numériques ou alpha-numériques.
- Les tables imbriquées (**NESTED TABLES**) qui sont indicées par des variables numériques et peuvent être lues et écrites directement depuis les colonnes d'une table.
- Les tableaux de type **VARRAY**, indicés par des variables numériques, dont le nombre d'éléments maximum est fixé dès leur déclaration et peuvent être lus et écrits directement depuis les colonnes d'une table.

Les collections

82

- Les collections de type **NESTED TABLE** et **VARRAY** doivent-êre initialisées après leur déclaration, à l'aide de leur constructeur qui porte le même nom que la collection (elles sont assignées à NULL lors de leur déclaration. Il est donc possible de tester leur nullité)

Les enregistrements

83

- Un enregistrement ressemble à une structure d'un L3G.
- Il est composé de champs qui peuvent être de type différent

Déclarations et initialisation

84

Les collections de type NESTED TABLE et INDEX-BY TABLES:

- Elles sont de taille dynamique et il n'existe pas forcément de valeur pour toutes les positions
- Déclaration d'une collection de type nested table
`TYPE nom type IS TABLE OF type élément [NOT NULL] ;`
- Déclaration d'une collection de type index by
`TYPE nom type IS TABLE OF type élément [NOT NULL]
INDEX BY index_by_type ;`

Déclarations et initialisation (2)

85

index_by_type représente l'un des types suivants :

- BINARY_INTEGER
- PLS_INTEGER(9i)
- VARCHAR2(taille)
- LONG

Déclarations et initialisation (3)

86

```
declare
  -- collection de type nested table
  TYPE TYP_NES_TAB is table of varchar2(100) ;
  -- collection de type index by
  TYPE TYP_IND_TAB is table of number index by binary_integer ;
  tab1 TYP_NES_TAB ;
  tab2 TYP_IND_TAB ;
Begin
    tab1 := TYP_NES_TAB('Lundi','Mardi','Mercredi','Jeudi' ) ;
    for i in 1..10 loop
        tab2(i):= i ;
    end loop ;
End;
/
Procédure PL/SQL terminée avec succès.
```

Les collections de type VARRAY

87

- Ce type de collection possède une dimension maximale qui doit être précisée lors de sa déclaration.
- Elle possède une longueur fixe et donc la suppression d'éléments ne permet pas de gagner de place en mémoire Ses éléments sont numérotés à partir de la valeur 1.
- Déclaration d'une collection de type VARRAY:
`TYPE nom type IS VARRAY (taille maximum) OF type élément [NOT NULL] ;`

88

01/12/2016

Les enregistrements

89

- Déclaration:

```
TYPE nom_type IS RECORD (  
    nom_champ type_élément [[ NOT NULL ] := expression ] [,  
    ....] ) ;  
Nom_variable nom_type ;
```

- Comme pour la déclaration des variables, il est possible d'initialiser les champs lors de leur déclaration

Les enregistrements (2)

90

```
declare
-- Record
TYPE T_REC_EMP IS RECORD (
  Num emp.empno%TYPE,
  Nom emp.ename%TYPE,
  Job emp.job%TYPE );

R_EMP T_REC_EMP ; -- variable enregistrement de type T_REC_EMP
Begin
    R_EMP.Num := 1 ;
    R_EMP.Nom := 'Scott' ;
    R_EMP.job := 'SUPPORT' ;

End;
/
Procédure PL/SQL terminée avec succès.
```

Les enregistrements (3)

91

- Bien sûr il est possible de gérer des tableaux d'enregistrements

```
declare
  -- Record
  TYPE T_REC_EMP IS RECORD (
    Num emp.empno%TYPE,
    Nom emp.ename%TYPE,
    Job emp.job%TYPE );
  -- Table de records –
  TYPE TAB_T_REC_EMP IS TABLE OF T_REC_EMP index by binary_integer ;
  t_rec TAB_T_REC_EMP ; -- variable tableau d'enregistrements
Begin
  t_rec(1).Num := 1 ;
  t_rec(1).Nom := 'Scott' ;
  t_rec(1).job := 'GASMAN' ;
  t_rec(2).Num := 2 ;
  t_rec(2).Nom := 'Smith' ;
  t_rec(2).job := 'CLERK' ;
End; /
```

92

Initialisation des collections (2)

93

- Il n'est pas obligatoire d'initialiser tous les éléments d'une collection. On peut même n'en initialiser aucun. Dans ce cas l'appel de la méthode constructeur se fait sans argument .

```
tab1 TYP_VAR_TAB := TYP_VAR_TAB();
```

- Cette collection n'a aucun élément initialisé. On dit qu'elle est vide.
- Une collection non initialisée n'est pas vide mais NULL.

Initialisation des collections (3)

94

- L'initialisation d'une collection peut se faire dans la section instructions, mais dans tous les cas, elle ne pourra pas être utilisée avant d'avoir été initialisée.

```
Declare
TYPE TYP_VAR_TAB is VARRAY(30) of varchar2(100);
tab1 TYP_VAR_TAB;
-- collection automatiquement assignée à NULL
Begin
-- La collection est assignée à NULL mais n'est pas manipulable –
    If Tab1 is null Then -- Test OK
        ...
    End if;
    Tab1 := TYP_VAR_TAB("","","","","","");
    -- La collection est manipulable
-- End ;
```

Accès aux éléments d'une collection

95

- La syntaxe d'accès à un élément d'une collection est la suivante :
Nom_collection(indice)
- L'indice doit être un nombre valide compris entre $-2\ 147\ 483\ 647$ et $+2\ 147\ 483\ 647$.
- Pour une collection de type NESTED TABLE, l'indice doit être un nombre valide compris entre 1 et $2\ 147\ 483\ 647$.
- Pour une collection de type VARRAY, l'indice doit être un nombre valide compris entre 1 et la taille maximum du tableau.
- Dans le cas d'une collection de type INDEX-BY Varchar2 ou Long, l'indice représente toute valeur possible du type concerné.

Accès aux éléments d'une collection(2)

96

- Indice peut être un littéral, une variable ou une expression

```
Declare
Type TYPE_TAB_EMP IS TABLE OF Varchar2(60) INDEX BY BINARY_INTEGER ;
emp_tab TYPE_TAB_EMP ;
i pls_integer ;
Begin
    For i in 0..10 Loop
        emp_tab( i+1 ) := 'Emp ' || ltrim( to_char( i ) ) ;
    End loop ;
End ;
/
```


Accès aux éléments d'une collection(3)

97

Declare

```
Type TYPE_TAB_JOURS IS TABLE OF INTEGER INDEX BY VARCHAR2(20) ;  
jour_tab TYPE_TAB_JOURS ;
```

Begin

```
    jour_tab( 'LUNDI' ) := 10 ;  
    jour_tab( 'MARDI' ) := 20 ;  
    jour_tab( 'MERCREDI' ) := 30 ;
```

End ;

/

Procédure PL/SQL terminée avec succès.

Accès aux éléments d'une collection(4)

98

- Il est possible d'assigner une collection à une autre à condition qu'elles soient de même type

Declare

Type `TYPE_TAB_EMP` IS TABLE OF `EMP%ROWTYPE` INDEX BY `BINARY_INTEGER` ;

Type `TYPE_TAB_EMP2` IS TABLE OF `EMP%ROWTYPE` INDEX BY `BINARY_INTEGER` ;

tab1 `TYPE_TAB_EMP` := `TYPE_TAB_EMP`(...);

tab2 `TYPE_TAB_EMP` := `TYPE_TAB_EMP`(...);

tab3 `TYPE_TAB_EMP2` := `TYPE_TAB_EMP2`(...);

Begin

tab2 := tab1 ; -- OK

tab3 := tab1 ; -- Illégal : types différents ...

End ;

Méthodes associées aux collections

99

- Les méthodes sont des fonctions ou des procédures qui s'appliquent uniquement aux collections.
- L'appel de ces méthodes s'effectue en préfixant le nom de la méthode par le nom de la collection.

`Nom_collection.nom_méthode[(paramètre, ...)]`

- Les méthodes ne peuvent pas être utilisées à l'intérieur de commandes SQL.

Méthodes associées aux collections (2)

100

EXISTS(indice):

- Cette méthode retourne la valeur TRUE si l'élément indice de la collection existe et retourne la valeur FALSE dans le cas contraire.
- Cette méthode doit être utilisée afin de s'assurer que l'on va réaliser une opération conforme sur la collection.
- Le test d'existence d'un élément qui n'appartient pas à la collection ne provoque pas l'exception SUBSCRIPT_OUTSIDE_LIMIT mais retourne simplement FALSE

Méthodes associées aux collections (3)

101

EXISTS(indice): exemple

```
If ma_collection.EXISTS(10) Then  
    Ma_collection.DELETE(10) ;  
End if ;
```

Méthodes associées aux collections (4)

102

COUNT:

- Cette méthode retourne le nombre d'éléments de la collection y compris les éléments NULL consécutifs à des suppressions.
- Elle est particulièrement utile pour effectuer des traitements sur l'ensemble des éléments d'une collection.

```
Declare
LN_Nbre integer ;
Begin
    LN_Nbre := ma_collection.COUNT ;
End ;
```

103

Méthodes associées aux collections (6)

104

FIRST:

- Cette méthode retourne le plus petit indice d'une collection.
- Elle retourne NULL si la collection est vide.
- Pour une collection de type VARRAY cette méthode retourne toujours 1

Méthodes associées aux collections (7)

105

LAST:

- Cette méthode retourne le plus grand indice d'une collection.
- Elle retourne NULL si la collection est vide.
- Pour une collection de type VARRAY cette méthode retourne la même valeur que la méthode COUNT

Méthodes associées aux collections (8)

106

PRIOR(indice) :

- Cette méthode retourne l'indice de l'élément précédent l'indice donné en argument.
- Elle retourne NULL si indice est le premier élément de la collection

```
LN_i := ma_collection.LAST ;  
While LN_i is not null Loop ...  
    LN_I := ma_collection.PRIOR(LN_I) ;  
End loop ;
```

Méthodes associées aux collections (9)

107

NEXT(indice):

- Cette méthode retourne l'indice de l'élément suivant l'indice donné en argument.
- Elle retourne NULL si indice est le dernier élément de la collection.

```
LN_i := ma_collection.FIRST ;  
While LN_i is not null Loop ...  
    LN_I := ma_collection.NEXT(LN_I) ;  
End loop ;
```

Méthodes associées aux collections(10)

108

EXTEND:

- Un seul élément NULL est ajouté à la collection

```
SQL> declare
  2   TYPE TYP_TAB is table of varchar2(100) ;
  3   tab TYP_TAB ;
  4   Begin
  5   tab := TYP_TAB( 'lundi','mardi','mercredi' ) ;
  6   tab.EXTEND ;
  7   tab(4) := 'jeudi' ;
  8   End;
  9   /
```

Procédure PL/SQL terminée avec succès.

EXTEND(n):

- n éléments NULL sont ajoutés à la collection

```
SQL> declare
  2     TYPE TYP_TAB is table of varchar2(100) ;
  3     tab  TYP_TAB ;
  4  Begin
  5     tab := TYP_TAB( 'lundi','mardi','mercredi' ) ;
  6     tab.EXTEND(4) ;
  7     tab(4) := 'jeudi' ;
  8     tab(5) := 'vendredi' ;
  9     tab(6) := 'samedi' ;
 10     tab(7) := 'dimanche' ;
 11  End;
 12  /
```

Procédure PL/SQL terminée avec succès.

EXTEND(n,i):

- n éléments sont ajoutés à la collection. Chaque élément ajouté contient une copie de la valeur contenue dans l'élément d'indice i

```
SQL> set serveroutput on
SQL> declare
  2   TYPE TYP_TAB is table of varchar2(100) ;
  3   tab TYP_TAB ;
  4   Begin
  5     tab := TYP_TAB( 'lundi','mardi','mercredi' ) ;
  6     tab.EXTEND(4,1) ;
  7     For i in tab.first..tab.last Loop
  8       dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') = ' || tab(i) ) ;
  9     End loop ;
 10   End;
 11   /
tab(1) = lundi
tab(2) = mardi
tab(3) = mercredi
tab(4) = lundi
tab(5) = lundi
tab(6) = lundi
tab(7) = lundi

Procédure PL/SQL terminée avec succès.
```

- **TRIM:**

Le dernier élément de la collection est supprimé

```
SQL> declare
  2   TYPE TYP_TAB is table of varchar2(100) ;
  3   tab TYP_TAB ;
  4   Begin
  5     tab := TYP_TAB( 'lundi','mardi','mercredi' ) ;
  6     tab.EXTEND(4,1) ;
  7     For i in tab.first..tab.last Loop
  8       dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') = ' || tab(i) ) ;
  9     End loop ;
 10     tab.TRIM ;
 11     For i in tab.first..tab.last Loop
 12       dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') = ' || tab(i) ) ;
 13     End loop ;
 14   End;
 15   /
tab(1) = lundi
tab(2) = mardi
tab(3) = mercredi
tab(4) = lundi
tab(5) = lundi
tab(6) = lundi
tab(7) = lundi
tab(1) = lundi
tab(2) = mardi
tab(3) = mercredi
tab(4) = lundi
tab(5) = lundi
tab(6) = lundi
```

Procédure PL/SQL terminée avec succès.

- **TRIM(n) :**

Les n derniers éléments de la collection sont supprimés

```
SQL> Declare
  2   TYPE TYP_TAB is table of varchar2(100) ;
  3   tab TYP_TAB ;
  4   Begin
  5     tab := TYP_TAB( 'lundi','mardi','mercredi' ) ;
  6     tab.EXTEND(4,1) ;
  7     For i in tab.first..tab.last Loop
  8       dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') = ' || tab(i) ) ;
  9     End loop ;
 10     tab.TRIM(4) ;
 11     dbms_output.put_line( 'Suppression des 4 derniers éléments' ) ;
 12     For i in tab.first..tab.last Loop
 13       dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') = ' || tab(i) ) ;
 14     End loop ;
 15   End;
 16 /
tab(1) = lundi
tab(2) = mardi
tab(3) = mercredi
tab(4) = lundi
tab(5) = lundi
tab(6) = lundi
tab(7) = lundi
Suppression des 4 derniers éléments
tab(1) = lundi
tab(2) = mardi
tab(3) = mercredi

Procédure PL/SQL terminée avec succès.
```


- Si le nombre d'éléments que l'on veut supprimer est supérieur au nombre total d'éléments de la collection, une exception **SUBSCRIPT_BEYOND_COUNT** est générée

```
SQL> Declare
2   TYPE TYP_TAB is table of varchar2(100) ;
3   tab  TYP_TAB ;
4   Begin
5   tab := TYP_TAB( 'lundi','mardi','mercredi' ) ;
6   tab.EXTEND(4,1) ;
7   For i in tab.first..tab.last Loop
8       dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') = ' || tab(i) ) ;
9   End loop ;
10  tab.TRIM(8) ;
11  dbms_output.put_line( 'Suppression des 8 derniers éléments' ) ;
12  For i in tab.first..tab.last Loop
13      dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') = ' || tab(i) ) ;
14  End loop ;
15  End;
16  /
tab(1) = lundi
tab(2) = mardi
tab(3) = mercredi
tab(4) = lundi
tab(5) = lundi
tab(6) = lundi
tab(7) = lundi
declare
*
ERREUR à la ligne 1 :
ORA-06533: Valeur de l'indice trop grande
ORA-06512: à ligne 10
```

DELETE:

- Suppression de tous les éléments d'une collection

```
SQL> Declare
2   TYPE TYP_TAB is table of varchar2(100) ;
3   tab TYP_TAB ;
4   Begin
5   tab := TYP_TAB( 'lundi','mardi','mercredi' ) ;
6   tab.EXTEND(4,1) ;
7   For i in tab.first..tab.last Loop
8       dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') = ' || tab(i) ) ;
9   End loop ;
10  tab.DELETE ;
11  dbms_output.put_line( 'Suppression de tous les éléments' ) ;
12  dbms_output.put_line( 'tab.COUNT = ' || tab.COUNT ) ;
13  End;
14  /
tab(1) = lundi
tab(2) = mardi
tab(3) = mercredi
tab(4) = lundi
tab(5) = lundi
tab(6) = lundi
tab(7) = lundi
Suppression de tous les éléments
tab.COUNT = 0

Procédure PL/SQL terminée avec succès.
```

• DELETE(n)

```
SQL> Declare
2   TYPE TYP_TAB is table of varchar2(100) ;
3   tab TYP_TAB ;
4   Begin
5   tab := TYP_TAB( 'lundi','mardi','mercredi' ) ;
6   tab.EXTEND(4,1) ;
7   For i in tab.first..tab.last Loop
8       dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') = ' || tab(i) ) ;
9   End loop ;
10  tab.DELETE(5) ;
11  dbms_output.put_line( 'Suppression de l''élément d''indice 5' ) ;
12  dbms_output.put_line( 'tab.COUNT = ' || tab.COUNT ) ;
13  For i in tab.first..tab.last Loop
14      dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') = ' || tab(i) ) ;
15  End loop ;
16  End;
17  /
tab(1) = lundi
tab(2) = mardi
tab(3) = mercredi
tab(4) = lundi
tab(5) = lundi
tab(6) = lundi
tab(7) = lundi
Suppression de l'élément d'indice 5
tab.COUNT = 6
tab(1) = lundi
tab(2) = mardi
tab(3) = mercredi
tab(4) = lundi
declare
*
ERREUR à la ligne 1 :
ORA-01403: Aucune donnée trouvée
ORA-06512: à ligne 14
```

- On peut observer que l'élément d'indice 5 de la collection, une fois supprimé, ne peut plus être affiché.
- Il convient, lorsque l'on supprime un ou plusieurs éléments d'une collection des tester l'existence d'une valeur avant de la manipuler

```

SQL> Declare
2   TYPE TYP_TAB is table of varchar2(100) ;
3   tab TYP_TAB ;
4   Begin
5   tab := TYP_TAB( 'lundi','mardi','mercredi' ) ;
6   tab.EXTEND(4,1) ;
7   For i in tab.first..tab.last Loop
8       dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') = ' || tab(i) ) ;
9   End loop ;
10  tab.DELETE(5) ;
11  dbms_output.put_line( 'Suppression de l''élément d''indice 5' ) ;
12  dbms_output.put_line( 'tab.COUNT = ' || tab.COUNT ) ;
13  For i in tab.first..tab.last Loop
14      If tab.EXISTS(i) Then
15          dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') = ' || tab(i) ) ;
16      Else
17          dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') inexistant ' ) ;
18      End if ;
19  End loop ;
20  End;
21  /
tab(1) = lundi
tab(2) = mardi
tab(3) = mercredi
tab(4) = lundi
tab(5) = lundi
tab(6) = lundi
tab(7) = lundi
Suppression de l'élément d'indice 5
tab.COUNT = 6
tab(1) = lundi
tab(2) = mardi
tab(3) = mercredi
tab(4) = lundi
tab(5) inexistant
tab(6) = lundi
tab(7) = lundi

```

Procédure PL/SQL terminée avec succès.

- Il est important de noter le décalage entre la valeur retournée par la méthode COUNT et celle retournée par la méthode LAST

Dans l'exemple précédent LAST retourne la plus grande valeur d'indice de la collection soit 7, alors que COUNT retourne le nombre d'éléments de la collection soit 6

Méfiez-vous de l'erreur facile consistant à penser que
COUNT = LAST

DELETE(n,m):

- Suppression des l'éléments dont les indices sont compris entre n et m (inclus) Si m est plus grand que n, aucun élément n'est supprimé .

```

SQL> Declare
2     TYPE TYP_TAB is table of varchar2(100) ;
3     tab  TYP_TAB ;
4 Begin
5     tab := TYP_TAB( 'lundi','mardi','mercredi' ) ;
6     tab.EXTEND(4,1) ;
7     For i in tab.first..tab.last Loop
8         dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') = ' || tab(i) ) ;
9     End loop ;
10    tab.DELETE(4,6) ;
11    dbms_output.put_line( 'Suppression des élément d''indice 4, 5 et 6' ) ;
12    dbms_output.put_line( 'tab.COUNT = ' || tab.COUNT ) ;
13    For i in tab.first..tab.last Loop
14        If tab.EXISTS(i) Then
15            dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') = ' || tab(i) ) ;
16        Else
17            dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') inexistant ' ) ;
18        End if ;
19    End loop ;
20 End;
21 /
tab(1) = lundi
tab(2) = mardi
tab(3) = mercredi
tab(4) = lundi
tab(5) = lundi
tab(6) = lundi
tab(7) = lundi
Suppression des élément d'indice 4, 5 et 6
tab.COUNT = 4
tab(1) = lundi
tab(2) = mardi
tab(3) = mercredi
tab(4) inexistant
tab(5) inexistant
tab(6) inexistant
tab(7) = lundi

```

Procédure PL/SQL terminée avec succès.

- Pour les collections de type VARRAY on ne peut supprimer que le dernier élément.
- Si l'élément à supprimer n'existe pas, aucune exception n'est générée.
- L'espace mémoire assigné aux éléments supprimés est conservé. Il est tout à fait permis de réassigner une nouvelle valeur à ces éléments.

Principales exceptions

122

Declare

```
TYPE TYP_TAB is table of varchar2(100) ;  
tab TYP_TAB ;  
lc$ valeur varchar2(100) ;
```

Begin

```
tab(1) := 'Lundi' ; -- ORA-06531: Référence à un ensemble non initialisé  
tab := TYP_TAB( 'lundi','mardi','mercredi' ) ;  
tab.EXTEND(4,1) ;  
tab.DELETE(4,6) ;  
lc$ valeur := tab(4) ; -- ORA-01403: Aucune donnée trouvée  
tab(0) := 'lunmanche' ; -- ORA-06532: Indice hors limites  
tab(22) := 'marcredi' ; -- ORA-06533: Valeur de l'indice trop grande  
lc$ valeur := tab(999999999999999999) ; -- ORA-01426: dépassement numérique  
lc$ valeur := tab(NULL) ; -- ORA-06502: PL/SQL : erreur numérique ou erreur sur une valeur:  
    la valeur de clé de la table d'index est NULL.
```

End ;

Bases de données avancées

123

LES EXCEPTIONS

Présentation

124

- Un exception est une erreur ou un avertissement, prédéfinie par Oracle ou défini par le programmeur.
- 4 catégories d'exceptions :
 - nommées prédéfinies par Oracle : Oracle les déclenche, l'utilisateur peut les récupérer (WHEN nomException)
 - nommées définies par l'utilisateur : l'utilisateur les déclare, les lève dans un programme PL/SQL, et peut les récupérer (WHEN nom...)
 - anonymes prédéfinies par Oracle : Oracle les déclenche sans les nommer, l'utilisateur peut quand même les récupérer (WHEN OTHERS ...)
 - anonymes définies par l'utilisateur : idem, mais c'est l'utilisateur qui les lève avec RAISE APPLICATION ERROR

Rappel de la structure d'un bloc

125

```
DECLARE  
    -- définitions de variables  
BEGIN  
    -- Les instructions à exécuter  
EXCEPTION  
    -- La récupération des erreurs  
END;
```

Saisir une exception

126

- Une exception ne provoque pas nécessairement l'arrêt du programme si elle est saisie par un bloc (dans la partie « EXCEPTION »)
- Une exception non saisie remonte dans la procédure appelante (où elle peut être saisie)

Exceptions prédéfinies

127

- **NO_DATA_FOUND**
- **TOO_MANY_ROWS**
- **VALUE_ERROR (erreur arithmétique)**
- **ZERO_DIVIDE**

Traitement des exceptions

128

```
BEGIN
```

```
...
```

```
EXCEPTION
```

```
    WHEN NO_DATA_FOUND THEN
```

```
        . . .
```

```
    WHEN TOO_MANY_ROWS THEN
```

```
        . . .
```

```
    WHEN OTHERS THEN -- optionnel
```

```
        . . .
```

```
END;
```

On peut utiliser les 2 variables
prédéfinies **SQLCODE** et **SQLERRM**

Traitement des exceptions

129

Exemple: utilisation de SQLCODE et SQLERRM

```
DECLARE
    name employees.last_name%TYPE;
    v_code NUMBER;
    v_errm VARCHAR2(64);
BEGIN
    SELECT last_name INTO name FROM employees WHERE employee_id = 1000;
    EXCEPTION
        WHEN OTHERS THEN
            v_code := SQLCODE;
            v_errm := SUBSTR(SQLERRM, 1 , 64);
            DBMS_OUTPUT.PUT_LINE('The error code is ' || v_code || '- ' || v_errm);
END;
/
```

Exceptions utilisateur

130

- Elles doivent être déclarées avec le type **EXCEPTION**.
- On les lève avec l'instruction **RAISE**.

Exemple 1

131

```
DECLARE
    salaire numeric(8,2);
    salaire_trop_bas EXCEPTION;
BEGIN
    select sal into salaire from emp
        where matr = 50;
    if salaire < 300 then
        raise salaire_trop_bas;
    end if;
    -- suite du bloc
EXCEPTION
    WHEN salaire_trop_bas THEN . . .;
    WHEN OTHERS THEN
        dbms_output.put_line(SQLERRM);
END;
```

Exceptions prédéfinies anonymes (1)

132

```
EXCEPTION
```

```
...
```

```
WHEN OTHERS THEN
```

```
    DECLARE – Bloc des autres erreurs
```

```
        codeErreur NUMBER := SQLCODE;
```

```
    BEGIN
```

```
        IF codeErreur = -02291 THEN
```

```
            – Contrainte d'intégrité
```

```
            RAISE_APPLICATION_ERROR (-20001, 'Client inexistant');
```

```
        ELSIF...
```

```
        ...
```

```
        END IF;
```

```
    END; – Bloc des autres erreurs
```

```
END; – Programme
```

Exceptions prédéfinies anonymes (2)

133

```
DECLARE
    ...
    monException EXCEPTION
    PRAGMA EXCEPTION_INIT (monException, -2400)
BEGIN
    ...
EXCEPTION
    WHEN monException THEN...
END;
```

Propagation d'une exception

134

- Où va une exception?:
- Une exception est déclenchée :
 - PL/SQL cherche un gestionnaire dans la partie Exception du bloc courant (WHEN adéquat).
 - Si pas de gestionnaire : on cherche dans le bloc englobant...
 - Si aucun : PL/SQL envoie un message d'exception non gérée à l'application appelante

Exemple 2

135

```
DECLARE
    cpt NUMBER := 0;    monException EXCEPTION;
BEGIN
    ...
    IF cpt < 0 THEN
        RAISE monException;
    END IF;
    ...
EXCEPTION
    WHEN monException THEN
        DBMS_OUTPUT.PUT_LINE('cpt ne doit pas être négatif');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Je ne connais pas cette erreur');
END;
```

Exceptions prédéfinies



Exception Name	Error	Description
ACCESS_INTO_NULL	ORA-06530	Attempted to assign values to the attributes of an uninitialized (NULL) object.
CASE_NOT_FOUND	ORA-06592	None of the choices in the WHEN clauses of a CASE statement is selected and there is no ELSE clause.
COLLECTION_IS_NULL	ORA-06531	Attempt to apply collection methods other than EXISTS to an uninitialized (NULL) PL/SQL table or VARRAY .
CURSOR_ALREADY_OPEN	ORA-06511	Exactly what it seems to be. Tried to open a cursor that was already open
DUP_VAL_ON_INDEX	ORA-00001	An attempt to insert or update a record in violation of a primary key or unique constraint
INVALID_CURSOR	ORA-01001	The cursor is not open, or not valid in the context in which it is being called.
INVALID_NUMBER	ORA-01722	It isn't a number, even though you are treating it like one to trying to turn it into one.
LOGIN_DENIED	ORA-01017	Invalid name and/or password for the instance.
NO_DATA_FOUND	ORA-01403	The SELECT statement returned no rows or referenced a deleted element in a nested table or referenced an initialized element in an Index-By table.
NOT_LOGGED_ON	ORA-01012	Database connection lost.
PROGRAM_ERROR	ORA-06501	Internal PL/SQL error.
ROWTYPE_MISMATCH	ORA-06504	The rowtype does not match the values being fetched or assigned to it.

Exceptions prédéfinies

137

Exception Name	Error	Description
SELF_IS_fs	ORA-30625	Program attempted to call a MEMBER method, but the instance of the object type has not been initialized. The built-in parameter SELF points to the object, and is always the first parameter passed to a MEMBER method.
STORAGE_ERROR	ORA-06500	A hardware problem: Either RAM or disk drive.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Reference to a nested table or varray index higher than the number of elements in the collection.
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Reference to a nested table or varray index outside the declared range (such as -1).
SYS_INVALID_ROWID	ORA-01410	The conversion of a character string into a universal rowid fails because the character string does not represent a valid rowid.
TIMEOUT_ON_RESOURCE	ORA-00051	The activity took too long and timed out.
TOO_MANY_ROWS	ORA-01422	The SQL INTO statement brought back more than one value or row (only one is allowed).
USERENV_COMMITSCN_ERROR	ORA-01725	Added for USERENV enhancement, bug 1622213.
VALUE_ERROR	ORA-06502	An arithmetic, conversion, truncation, or size-constraint error. Usually raised by trying to cram a 6 character string into a VARCHAR2(5) variable
ZERO_DIVIDE	ORA-01476	Not only would your math teacher not let you do it, computers won't either. Who said you didn't learn anything useful in primary school?

Bases de données avancées

138

LES PROCÉDURES ET LES FONCTION STOCKÉES

Fonctions et procédures stockées

139

- Pourquoi?
 - Pour enregistrer des programmes dans le noyau d'Oracle.
 - Comme une table ou une vue, elles peuvent être utilisées par d'autres utilisateurs, s'ils ont les droits voulus.
 - Stockées sous forme de pseudo-code : pas de nouvelle compilation ! Efficace.

Déclaration procédure stockée

140

Syntaxe

```
CREATE [OR REPLACE] PROCEDURE nom_procedure  
  [(liste paramètres formels)]  
  AS | IS  
  [partie déclaration]  
  BEGIN  
    ...  
  [EXCEPTION  
    ...]  
  END [nom_procedure];
```

partie déclaration: similaire à celle d'un bloc PL/SQL

Déclaration procédure stockée: paramètres

141

Syntaxe

```
nom paramètre [IN |  
               OUT [NOCOPY]]  
               IN OUT [NOCOPY]] type_paramètre  
               [ := | DEFAULT expression ]
```

type_paramètre : un type PL/SQL

IN : paramètre en entrée, non modifié par la procédure

OUT : paramètre en sortie, peut être modifié par la procédure,
transmis au programme appelant

IN OUT : à la fois en entrée et en sortie

par défaut : **IN**

NOCOPY : pour passer des références et non des valeurs (mais le
compilateur décide!)

Exemple de procédure stockée

142

- On cherche les réalisateurs qui ont joué dans un certain nombre de leur film...

```
CREATE PROCEDURE realActeursProc (nbFilms NUMBER) IS
  nbRealAct NUMBER(5);
  singulierException EXCEPTION;
BEGIN
  SELECT COUNT(distinct A.numIndividu) INTO nbRealAct
    FROM Film F, Acteur A
   WHERE A.numIndividu = realisateur
     AND F.numFilm=A.numFilm;
  IF nbRealAct > nbFilms THEN
    DBMS_OUTPUT.PUT_LINE(nbRealAct||' réalisateurs ont joué
      dans plus de '||nbFilms||'de leurs films');
  ELSE DBMS_OUTPUT.PUT_LINE('Aucun réalisateur n'a joué
    dans plus de '||nbFilms||'de ses films');
  END IF;
END;
```

Déclaration fonction stockée

143

Syntaxe

```
CREATE [OR REPLACE] FUNCTION nom_fonction
  [(liste paramètres formels)]
  RETURN typeRetour    AS | IS
  [partie déclaration]
  BEGIN
    ...
    RETURN valeurRetout
    ...
  [EXCEPTION ...]
  END [nom_fonction];
```

partie déclaration: similaire à celle d'un bloc PL/SQL

typeRetour : le type PL/SQL de valeurRetour retournée par la fonction.

liste de paramètres : idem procédures, mais IN préférable dans les fonctions!!!

Exemple de fonction stockée

144

- On cherche toujours les réalisateurs qui ont joué dans plus de nbFilms de leurs films...

```
CREATE FONCTION nbRealActeurFonc (nbFilms NUMBER)
RETURN NUMBER IS
    nbRealAct NUMBER(5) := 0 ;
BEGIN
    SELECT COUNT(distinct A.numIndividu) INTO nbRealAct
    FROM Film F, Acteur A
    WHERE A.numIndividu = realisateur
    AND F.numFilm=A.numFilm;
    RETURN nbRealAct;
END;
```


Appel de procédures et de fonctions stockées

145

Appel à une procédure dans un programme PL/SQL

```
nom_procedure [(liste de paramètres effectifs)];
```

Appel à `realActeursProc`

```
nbFilms:=20;  
...  
realActeursProc (nbFilms);
```

Appel à une fonction dans un programme PL/SQL

```
nomVariable := nom_fonction [(liste de paramètres effectifs)];
```

Appel à `nbRealActeurFonc`

```
nbFilms:=20;  
...  
nbGdActeursReals := nbRealActeurFonc (nbFilms);
```

Récessivité

146

Exemple de factorielle!

```
CREATE FUNCTION facto (N IN NUMBER(3))  
  RETURN NUMBER IS  
BEGIN  
  IF (N<=1) THEN  
    RETURN 1;  
  ELSE  
    RETURN N * facto(N - 1);  
  END IF;  
END;
```

Compilation et suppression

147

Compilation

Pour enregistrer une procédure ou une fonction stockée, il faut

- avoir le droit **CREATE PROCEDURE**
- donner l'ordre de création à Oracle.

Si pas d'erreur : → stockée sous forme compilée

Sinon : message d'erreur + erreurs stockées dans des vues générales (**USER_ERRORS**, **ALL_ERRORS**, **DBA_ERRORS**)

→ les consulter pour corriger (**REPLACE PROCEDURE...**)

Consultation

```
SELECT * FROM USER_ERRORS [WHERE NAME=nom];
```

Suppression

```
DROP {PROCEDURE | FUNCTION} nom_procedure_ou_fonction
```

Exemple

148

- On cherche les acteurs ayant joué dans plus de N films... Et on veut leurs noms!

```
CREATE PROCEDURE nomsGdsActeurs (nbFilms NUMBER)
  Cursor lesActeurs IS
    SELECT nomIndividu
    FROM Individu
    WHERE numIndividu IN
      (SELECT numActeur
      FROM acteur
      GROUP BY numIndividu
      HAVING Count(numFilm)>nbFilms);
```

...→

Exemple (suite)

149

```
....←  
BEGIN  
  DBMS_OUTPUT.PUT_LINE('Voici les acteurs ayant  
    joué dans plus de '||nbFilms||' films : ')  
  FOR ligneCurseur IN lesActeurs ;  
  LOOP  
    DBMS_OUTPUT.PUT_LINE(ligneCurseur.nomIndividu)  
  END LOOP;  
  DBMS_OUTPUT.PUT_LINE('Voici le nombre d'acteurs  
    ayant joué dans plus de '||nbFilms||' films : '  
    || lesActeurs%rowCount);  
END;
```



Triggers

Déclencheur (Trigger)



- C'est un traitement implicite déclenché par un événement.
- Utilisé pour implémenter des règles de gestion complexes et pour étendre les règles d'intégrité référentiel associées aux tables.
- Un trigger est défini au moyen de PL/SQL.

Caractéristiques d'un Trigger



- Son code est stocké dans la base de données.
- Il est déclenché sur un événement complété par un prédicat.
- Un déclencheur peut être actif ou non.
- Si un déclencheur aboutit, la transaction qui l'a appelé peut se poursuivre.

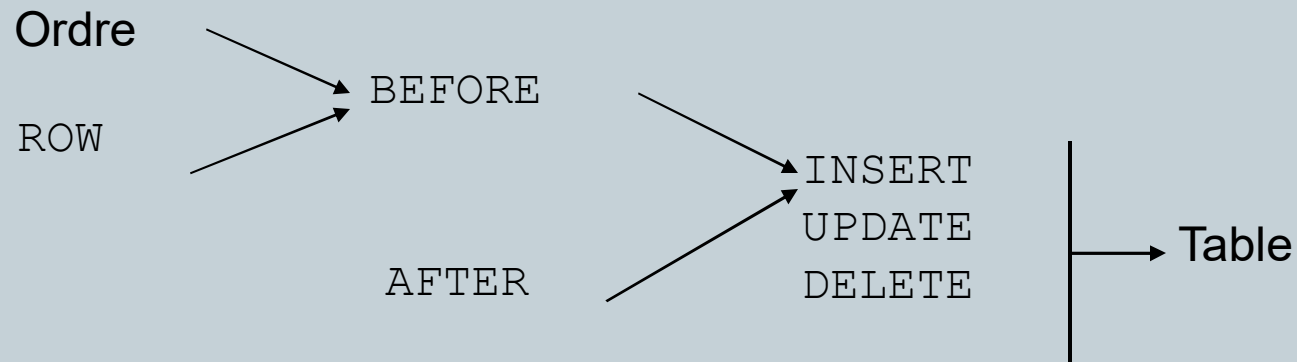
Description d'un trigger



- Le traitement peut s'appliqué :
 - À l'ordre
 - ✦ Le trigger ne s'applique qu'une fois
 - À chaque ligne de la table concernée par l'événement générateur
 - ✦ Le trigger s'applique autant de fois que nécessaire

Structure d'un déclencheur

- Événement



Éléments constitutifs



```
1 CREATE OR REPLACE TRIGGER myFirstTrigger
2   BEFORE UPDATE OF ename ON emp
3   FOR EACH ROW
4 BEGIN
5     DBMS_OUTPUT.ENABLE(20000);
6     DBMS_OUTPUT.PUT_LINE(' :NEW.ENAME || ' ' || :OLD.ENAME );
7 EXCEPTION
8     WHEN OTHERS THEN
9         DBMS_OUTPUT.PUT_LINE(SQLERRM);
10 END;
```

```
SQL> UPDATE emp
        SET ename = 'Durand'
        WHERE empno = 7839;
SQL> Durand Sami
```

Résolution multi-événements



- Un Trigger peut répondre à plusieurs événements. Dans ce cas, il est possible d'utiliser les prédicats intégrés **INSERTING**, **UPDATING** ou **DELETING** pour exécuter une séquence particulière du traitement en fonction du type d'événement.

Résolution multi-événements

```
CREATE OR REPLACE TRIGGER myFirstTrigger
AFTER UPDATE OR INSERT ON emp
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.ENABLE(20000);
    IF INSERTING THEN
        DBMS_OUTPUT.PUT_LINE(' INSERT ');
    END IF;
    IF UPDATING('ENAME') THEN
        DBMS_OUTPUT.PUT_LINE(' UPDATED ' || :NEW.ENAME);
    END IF;
END;
/
SQL> UPDATE emp SET ename = 'TOTO' WHERE empno = 7839;
UPDATED TOTO
1 row updated.
```

Typologie d'un Trigger



BEFORE	AFTER
BEFORE UPDATE ligne	AFTER UPDATE ligne
BEFORE DELETE ligne	AFTER DELETE ligne
BEFORE INSERT ligne	AFTER INSERT ligne
BEFORE UPDATE ordre	AFTER UPDATE ordre
BEFORE DELETE ordre	AFTER DELETE ordre
BEFORE INSERT ordre	AFTER INSERT ordre

Mise hors-service d'un Trigger



```
ALTER TRIGGER myTrigger DISABLE;
```

```
ALTER TABLE maTable  
    DISABLE ALL TRIGGERS;
```

```
ALTER TRIGGER tonTrigger ENABLE  
ALTER TRIGGER noTrigger  
    ENABLE ALL TRIGGERS;
```

```
DROP TRIGGER ceTrigger;
```

USER_TRIGGERS



```
SQL> desc user_triggers
```

Name	Null?	Type
-----	-----	-----
TRIGGER_NAME		VARCHAR2 (30)
TRIGGER_TYPE		VARCHAR2 (16)
TRIGGERING_EVENT		VARCHAR2 (227)
TABLE_OWNER		VARCHAR2 (30)
BASE_OBJECT_TYPE		VARCHAR2 (16)
TABLE_NAME		VARCHAR2 (30)
COLUMN_NAME		VARCHAR2 (4000)
REFERENCING_NAMES		VARCHAR2 (128)
WHEN_CLAUSE		VARCHAR2 (4000)
STATUS		VARCHAR2 (8)
DESCRIPTION		VARCHAR2 (4000)
ACTION_TYPE		VARCHAR2 (11)
TRIGGER_BODY		LONG