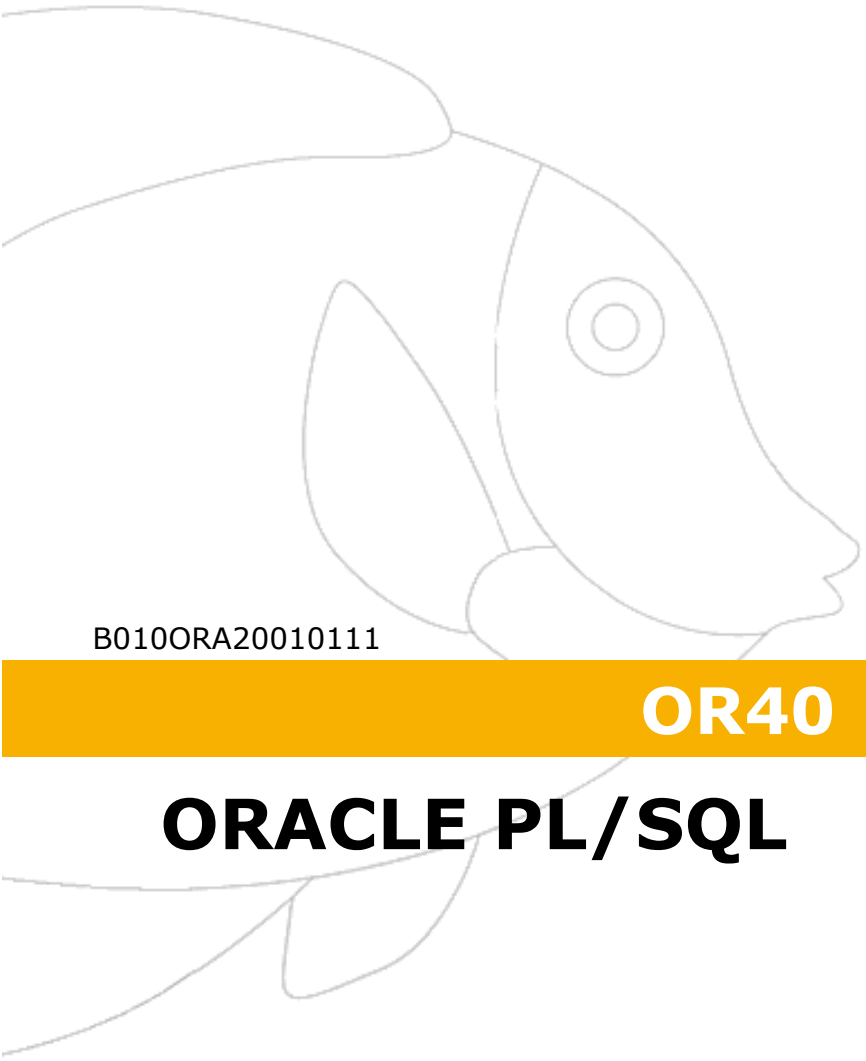


B010ORA20010111

OR40

ORACLE PL/SQL

Centre de formation d'Atos Origin
13, rue de Bucarest - 75008 Paris
Tél. : 01 55 30 62 00 - Fax : 01 55 30 62 30
Mail : formation@atosorigin.com

SOMMAIRE

LE LANGAGE PL/SQL

Objectif

Connaître le langage procédural d'Oracle. Appréhender ses mécanismes et savoir les mettre en œuvre.

Pré-requis : Connaître l'algorithmie de programmation et le Langage Oracle SQL

1. INTRODUCTION

1.1. Pourquoi PL/SQL ?

SQL est un langage complet d'accès à une Base de Données Relationnelle.

SQL est non procédural.

Le PL/SQL est un langage procédural. C'est une extension du langage SQL et il est donc capable d'envoyer au noyau ORACLE tous les ordres SQL :

- Les ordres LID/LMD :

  ⇒ SELECT, INSERT, UPDATE, DELETE

- La gestion des transactions :

  ⇒ COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION, LOCK TABLE

La partie procédurale comprend :

- Les curseurs (zones de contexte, zones de mémoire)

  ⇒ DECLARE, OPEN, FETCH, CLOSE

- Les boucles

  ⇒ LOOP, FOR, WHILE, EXIT, GOTO

- Les conditions

  ⇒ IF, THEN, ELSIF, ELSE, END IF,

- Les attributs

  ⇒ Définition de variables locales à une procédure.

```
DECLARE
   NOM VARCHAR2(30) := '';
```

⇒ Affectation de valeurs.

```
NOM := UPPER(NOM) ;
```

⇒ Calculs divers.

```
PRIX_FFR := PRIX_DEV * TAUX_DEV ;
```

- La gestion des erreurs

  (section EXCEPTION du bloc PL/SQL)

```
DECLARE
nom     VARCHAR2(30);
BEGIN
.....
EXCEPTION
WHEN NO_DATA_FOUND THEN ...
END;
```

- Création et appel de Fonctions et de Procédures stockées dans la base.

- Utilisation de fonctions prédéfinies

  ⇒ TO_DATE, TO_CHAR, TO_NUMBER, UPPER, LOWER, SUBSTR, ...

- Création de Packages

  ⇒ Encapsulation d'objets dans une même unité logique de traitement.

- Création de Triggers

  ⇒ Traitement procédural lié à une table et se déclenchant lors d'un événement survenu sur la table.

1.2. Documents de référence

PL/SQL User's Guide and Reference release 2.3

2. ENVIRONNEMENT PL/SQL

PL/SQL peut être utilisé au sein de différents outils :

⇒ SQL*PLUS

⇒ Précompilateurs (PRO*)

⇒ Developer FORMS

⇒ Developer REPORTS

⇒ Developer GRAPHICS

2.1. Fonctionnement

PL/SQL interprète des 'blocs' de commandes

⇒ Gain de transmission

⇒ Gain de performances

BLOC PL/SQL
SQL
IF **...**
THEN

APPLICATION **->** SQL **>** ORACLE

ELSE
SQL
END IF ;
...

PL/SQL est composé de deux "moteurs" :

- SQL STATEMENT EXECUTOR
  Ce moteur se trouve toujours dans le noyau ORACLE

- PROCEDURAL STATEMENT EXECUTOR
  Ce moteur se trouve soit :
  - ✓ - Dans le noyau ORACLE (RDBMS)
  - ✓ - Dans l'outil (FORMS par exemple).
  - ✓ - Ou dans les deux (architecture Client / Serveur)

2.2. Intégration dans le noyau

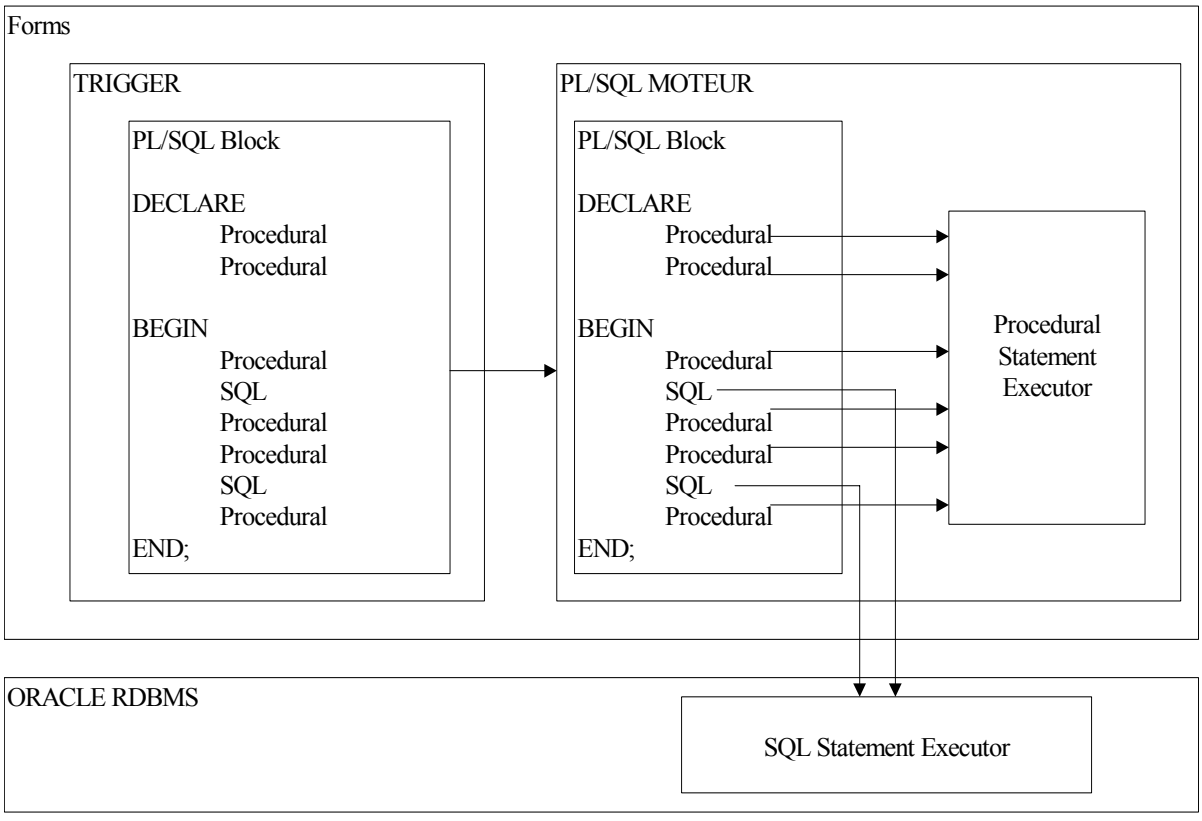
Utilisable avec :

- ✓ Les précompilateurs
- ✓ SQL*PLUS
- ✓ Server Manager

2.3. Intégration dans les outils

Utilisable avec :

- ✓ Developer FORMS
- ✓ Developer REPORTS
- ✓ Developer GRAPHICS



3. STRUCTURE D'UN BLOC PL/SQL

Un bloc PL/SQL est composé de trois parties

- Une partie déclarative (Facultative)
- Une partie exécutable (Obligatoire)
- Une partie exception (Facultative)

DECLARE

⇒ Déclarations de variables, constantes ,exceptions, curseurs

BEGIN

- ✦ Commandes SQL du langage de manipulation des données
- ✦ Utilisation de structures de contrôles (conditionnels, itératifs)
- ✦ Utilisation des curseurs
- ✦ Appels de fonctions, procédures, packages
- ✦ Utilisation de blocs PL/SQL imbriqués

EXCEPTION

- ✦ Traitement des exceptions (erreurs)

END ;
/

Remarques

Chaque instruction se termine par un point-virgule: « **;** »

Les commentaires sont possibles :

Un commentaire sur une ligne commence par : « -- »

-- *Le reste de la ligne est en commentaire*

Un commentaire multi - ligne commence par « ***/**** » et se termine par « ******* »

/ Début de commentaire .....*

..... fin de commentaire */

-On peut imbriquer les blocs

4. PL/SQL ET INSTRUCTION SELECT

Toute donnée extraite d'une table doit être obligatoirement réceptionnée dans une variable.
Il existe deux manières de réceptionner une donnée:

-Soit par un ordre SELECT simple:

```
SELECT nom
INTO nom_variable
FROM e_emp
WHERE no=25;
```

-Soit par l'intermédiaire d'un curseur qui permet de gérer les ordres SELECT qui ramènent plusieurs lignes.

☞ Lorsqu' une requête SELECT ne ramène aucune ligne, Oracle génère l'erreur « NO_DATA_FOUND »

5. LES VARIABLES

PL/SQL gère deux types de variables

Les variables locales :

Chaque variable et constante à un type de donnée associé (Datatype) qui spécifie son format de stockage, ses contraintes et son jeu valide de valeurs .

Les variables externes :

1) Les variables champs écrans FORMS

Les variables de lien (« bind » variables –variables SQL).

Les variables du langage hôte dans les langages PRO.

Elles sont toujours préfixées de ' :' lors de leur utilisation.

2) Les variables PL/SQL déclarées dans les packages.

Elles sont toujours préfixées du nom du package lors de leur utilisation.

5.1. Les Types de données

BINARY_INTEGER : est utilisé pour stocker des entiers signés compris dans l'intervalle [- 2147483647 .. 2147483647]

NUMBER : est utilisé pour stocker des nombres suivant la syntaxe NUMBER(precision, scale)

precision est un entier qui spécifie le nombre de chiffres.
Sa longueur maximale est de 38 positions. S'il n'est pas spécifié, sa valeur par défaut est la valeur maximale supportée par le système.

scale est un entier qui détermine la position de l'arrondi. Il varie entre -84 et 127. S'il n'est pas spécifié, sa valeur par défaut est zéro.

Par exemple, si scale = 2 alors la valeur 3.556 devient 3.5
si scale = -3 alors la valeur 5459 devient 5000
si scale = -2 alors la valeur 5459 devient 5500
et la valeur 5449 devient 5400

Une valeur de type BINARY_INTEGER requiert moins d'espace qu'une valeur de type NUMBER.

PLS_INTEGER : est utilisé pour stocker des entiers signés compris dans l'intervalle [-2147483647.. 2147483647].
Une valeur de type PLS_INTEGER requiert moins d'espace qu'une valeur de type NUMBER.

☞ L'utilisation de PLS_INTEGER est recommandée pour de meilleures performances par rapport aux types NUMBER et BINARY_INTEGER.

CHAR : est utilisé pour manipuler des chaînes de caractères de taille fixe suivant la syntaxe CHAR(taille_maximale)

taille_maximale à pour valeur maximale 32767 caractères

Il ne peut pas être une constante ni une variable. Seul un entier littéral est autorisé.
S'il n'est pas spécifié, sa valeur est par défaut 1.

LONG : est utilisé pour manipuler des chaînes de caractères de longueur variable.
La taille maximale d'une variable LONG est de 2 Go de caractères.

RAW : est utilisé pour stocker des données binaires.
Une variable RAW peut stocker 32767 octets.

LONGRAW : est utilisé pour stocker jusqu'à 2 Go de données binaires.

ROWID : est un type de données interne à Oracle qui permet de stocker des valeurs binaires appelées Rowids.
Celles-ci identifient de manière unique chaque enregistrement et fournissant ainsi un moyen d'accéder plus rapidement aux enregistrements.

VARCHAR2 : est utilisé pour manipuler des chaînes de caractères de longueur variable allant jusqu'à 32767 caractères.
La syntaxe est VARCHAR2(taille_maximale) où taille_maximale ne peut pas être une constante ou une variable. Seul un entier littéral est autorisé.

BOOLEAN : est utilisé pour stocké les valeurs TRUE ,FALSE et NULL.
Seules des variables peuvent avoir ce type de données.

DATE : est utilisé pour stocker des dates de taille fixe.

MSLABEL : Type de données utilisé seulement sur Oracle trusted, permet de stocker le label du système d'exploitation sur 4 octets.

5.2. Conversion des types de Données

5.2.1. Conversion explicite

La conversion explicite permet de convertir une valeur d'un certain type en un autre à l'aide des fonctions SQL telles que TO_DATE, TO_NUMBER, TO_CHAR, ...

TO_CHAR(DATE , FORMAT) → CHAINE DE CARACTERE

TO_CHAR('25-DEC-96','DD/MM/YYYY') = '25/12/1996'

TO_NUMBER(CHAINE , FORMAT) → NOMBRE

TO_NUMBER('125.35' , '999.99') = 125.35

TO_DATE(CHAINE , FORMAT) → DATE

TO_DATE('25/12/1996' , 'DD/MM/YYYY') = '25-DEC-1996'

Tableau des conversions explicites

Vers / De	CHAR	DATE	NUMBER	RAW	ROWID
CHAR		TO_DATE	TO_NUMBER	HEXTORAW	CHARTOROWID
DATE	TO_CHAR				
NUMBER	TO_CHAR	TO_DATE			
RAW	RAWTOHEX				
ROWID	ROWIDTOCHAR				

5.2.2. Conversion implicite

La conversion implicite est réalisée automatiquement par PL/SQL.

Si PL/SQL n'arrive pas à déterminer la conversion implicite nécessaire, la compilation engendre une erreur. Dans ce cas, on utilise la conversion explicite.

Le tableau suivant montre les conversions implicites acceptées par PL/SQL.

Vers / De	BINARY_ INTEGER	CHAR	VARCHAR2	LONG	NUMBER	PLS_ INTEGER	DATE	RAW	ROWID
BINARY_ INTEGER		Oui	Oui	Oui	Oui	Oui			
CHAR	Oui		Oui	Oui	Oui	Oui	Oui	Oui	Oui
VARCHAR2	Oui	Oui		Oui	Oui	Oui	Oui	Oui	Oui
LONG		Oui	Oui					Oui	
NUMBER	Oui	Oui	Oui	Oui		Oui			
PLS_ INTEGER	Oui	Oui	Oui	Oui	Oui				
DATE		Oui	Oui	Oui					
RAW		Oui	Oui	Oui					
ROWID		Oui	Oui						

```
DECLARE
   nombre      NUMBER(3):= '5'; -- conversion implicite
   ligne       VARCHAR2(25) := 6 ; -- conversion implicite
   variable1   CHAR(2) := '5' ;
   variable2   CHAR(2) := '2' ;

   BEGIN

      total := variable1 - variable2 ;     -- conversion implicite
      .....
```

5.3. Variables et constantes de type ORACLE

Les variables locales et les constantes sont définies dans la section DECLARE du bloc PL/SQL.
Déclarer une variable ou une constante consiste à lui allouer un espace pour stocker une valeur et à spécifier un type de donnée.
On peut aussi lui assigner une valeur par défaut et /ou spécifier la contrainte NOT NULL.

```
Num_employe     NUMBER(10) ;
Date_Jour       DATE   := SYSDATE ;   -- Initialisée à la date du jour
Logique         BOOLEAN ;
```

Il n'est pas permis de spécifier un type de donnée pour une liste de variables

```
A , B , C       PLS_INTEGER ;   --Incorrect
```

Les variables ne peuvent pas partager le même nom si leur type de donnée est différent.

```
Employe  VARCHAR2(20) ;
Employe  NUMBER ;          duplication incorrecte de la variable employe
```

Utilisation indifférente de majuscules ou de minuscules pour déclarer une variable

PL/SQL ne fait aucune différence pour les noms de variable suivants :

```
Employe     VARCHAR2(20) ;
emPLOye     VARCHAR2(20) ;
EMPLOYE     VARCHAR2(20) ;
```

Lorsque la **contrainte NOT NULL** est spécifiée, la variable ou constante doit être initialisée sinon une erreur survient à la compilation.
La valeur d'initialisation ne peut être NULL, sinon une erreur survient à l'exécution du programme.

```
Nom_Departement   VARCHAR2(15)   NOT NULL   := 'FINANCE'-- correct
Nom_Departement   VARCHAR2(15)   NOT NULL ;                -- incorrect
Nom_Departement   VARCHAR2(15)   NOT NULL   :=  NULL ;     -- incorrect
```

On peut déclarer une constante avec le mot réservé CONSTANT :

```
TVA  CONSTANT  NUMBER  := 20.6 ;
Pi   CONSTANT  REAL    := 3.1415 ;
```

5.4. Variable référencée à une colonne d'une table de la base

L'attribut %TYPE spécifie pour une variable le type d'une colonne de table.

Nom_Variable Table.colonne%TYPE ;

Si la colonne de la table est définie avec la contrainte NOT NULL, cette contrainte ne s'applique pas à la variable.

```
DECLARE

   num_emp e_emp.no%TYPE ;

BEGIN

   num_emp := NULL ;    -- Correct

END ;
```

Lors de l'utilisation de noms de variables identiques à ceux de colonnes de tables

La commande DELETE supprime l'ensemble des enregistrements de la table et non pas le seul employe dont le no est 10 :

```
DECLARE
   No   NUMBER(10) :=  10 ;
BEGIN
  DELETE FROM  e_emp
  WHERE  no = no ;
  .........
END ;
```

Pour éviter ces problèmes, soit on différencie les noms de variables de ceux des colonnes, soit on préfixe le nom des variables avec le label du bloc :

```
<<Nom_label>>
DECLARE
   no  NUMBER(10) := 10 ;
BEGIN
   DELETE FROM e_emp
   WHERE  no = nom_label.no ;
   ..............
END ;
```

5.5. Variable référencée à une table de la base

L'attribut %ROWTYPE spécifie pour une variable la structure d'une ligne de table.

5.5.1. Déclaration

Nom_Variable Table%ROWTYPE ;

```
DECLARE

   employe e_emp%ROWTYPE ;
```

5.5.2. Affectation de valeurs

-Par référence aux champs:

```
BEGIN

   employe.nom := 'DOLE' ;
   employe.prenom :=  'ERIC' ;
   employe.dt_entree := SYSDATE ;

END;
/
```

-En utilisant la commande SELECT ou FETCH (Voir chapitre sur les curseurs):

```
   BEGIN
   SELECT *
   INTO employe
   FROM e_emp
   WHERE nom ='DUMAS';

   -- autre exemple

   SELECT nom,dt_entree
   INTO employe.nom,employe.dt_entree
   FROM e_emp
   WHERE nom ='DUMAS';
END ;
```

5.6. Enregistrements prédéfinis (RECORDS)

On peut utiliser l'attribut %ROWTYPE pour déclarer un enregistrement du type d'un enregistrement d'une table de la base. Cependant, il est impossible de cette façon d'attribuer aux champs qui composent l'enregistrement nos propres spécifications.
L'implémentation du type de donnée RECORD lève cette restriction.

5.6.1. Déclaration d'un RECORD

La déclaration d'un RECORD se fait en deux étapes dans la section DECLARE du bloc PL/SQL

1) Déclarer le type du RECORD suivant la syntaxe :

```
TYPE  Type_record IS RECORD
        (Nom_champ1 {Type_champ | Variable%TYPE | Table.colonne%TYPE
                                    | Table%ROWTYPE } [NOT NULL] ,
        Nom_champ2 {Type_champ | Variable%TYPE | Table.colonne%TYPE
                                    | Table%ROWTYPE } [NOT NULL],

        Nom_champN {Type_champ | Variable%TYPE | Table.colonne%TYPE
                                    | Table%ROWTYPE } [NOT NULL]
    ) ;
```

- ✓ Le nom d'un champ est unique.
- ✓ Les champs déclarés avec la contrainte NOT NULL doivent être initialisés.
- ✓ Type_record peut être utilisé comme type de donnée d'un champ dans la déclaration d'un autre RECORD.

2) Déclarer les RECORD sur le type déclaré à la première étape

```
        Nom_record  Type_record ;
```

```
DECLARE
TYPE  Employe_type IS RECORD
      (Employe      e_emp.Nom%TYPE ,
      Service       e_Service.Nom%TYPE ,
      Sal           NUMBER(11,2) ,
      Date_Jour     DATE := SYSDATE ) ;
      Employe_record   Employe_type ;
      Employe2_record Employe_type ;
```

5.6.2. Affectation de valeurs

1) -Par référence aux Champs:

```
BEGIN

Employe_record.Employe := 'PEROS' ;
Employe_record.sal := 15000 ;
…
END;
/
```

2) -En assignant un record à un autre record de même type:

```
BEGIN
Employe2_record :=Employe_record;
…
END;
/
```

- même si deux records ont des champs aux propriétés identiques, ils doivent être de même type pour assigner l'un à l'autre.

3) -En utilisant la commande SELECT ou FETCH (voir chapitre sur les curseurs):

```
BEGIN
SELECT e.nom, s.nom, e.salaire, e.dt_entree
INTO Employe_record
FROM e_emp e ,e_service s
WHERE…

END ;
/
```

Ou

```
BEGIN
SELECT e.nom, s.nom
INTO Employe_record.Employe, Employe_record.service
From e_emp e, e_service s
WHERE…

END ;
/
```

5.7. Tables PL/SQL (Tableaux)

Les objets de type TABLE sont appelés tables PL/SQL ou tableaux.

Ce sont des tableaux de scalaires, à une dimension, indicés par un entier de type BINARY_INTEGER.

5.7.1. Déclarer un tableau

La déclaration d'un tableau se fait en deux étapes dans la section DECLARE du bloc PL/SQL.

1) Déclarer le type du tableau suivant la syntaxe :

```
TYPE  nom_type  IS TABLE OF
( Type_colonne  | Variable%TYPE | Table.colonne%TYPE |
  Type_record) [NOT NULL]
INDEX BY BINARY_INTEGER ;
```

2) Déclarer les tableaux sur le type déclaré à la première étape.

Déclarer le tableau nommé essais, constitué d'une colonne de type CHAR(5)

```
DECLARE

   TYPE  Type_table  IS TABLE OF CHAR(5)
   INDEX BY BINARY_INTEGER ;

   essais   Type_table ;

BEGIN
   .....
END ;
```

Déclarer le tableau nommé essais, de type Type_Table et constitué d'une colonne de même type que la colonne salaire de la table E_emp.

```
DECLARE

   TYPE  Type_table IS TABLE OF  E_emp.Salaire%TYPE
   INDEX BY BINARY_INTEGER ;

   essais   Type_table ;


BEGIN
....
END ;
```

5.7.2. Accès aux données du tableau

On accède aux données de la table en utilisant la clé primaire selon la syntaxe :

Tableau_Plsql(valeur_clé_primaire)

où valeur_clé_primaire est de type BINARY_INTEGER.
Elle peut donc prendre une valeur de l'intervalle $-2^{31}-1$ à $2^{31}-1$.

```
Table_sal(5)
Table_sal(-18)
```

5.7.3. Insertion de valeurs dans le tableau

La valeur de la clé primaire permet d'affecter une valeur dans le tableau :

Tableau_Plsql(valeur_clé_primaire) ;

```
Table_salaire(1) := 12000 ;
Table_salaire(2) :=  10000 ;
```

5.7.4. Attributs des tables PL/SQL ou tableaux

(seulement depuis la version 2.3 de PL/SQL)

EXISTS(n) retourne TRUE si le n^{ième} élément du tableau PL/SQL existe. Sinon, il retourne FALSE

IF tab_salaire .EXISTS(i) THEN
…

COUNT retourne le nombre d'éléments du tableau PL/SQL

IF tab_salaire.COUNT = 25 THEN
....

FIRST et LAST retournent respectivement la plus petite et la plus grande valeur de l'index

IF tab_salaire.FIRST < tab_salaire.LAST THEN
....

PRIOR(n) retourne la valeur de l'index qui précède l'index n

NEXT(n) retourne la valeur de l'index qui suit l'index n

DELETE supprime tous les élément du tableau PL/SQL

tab_salaire.DELETE ;

DELETE(n) supprime le n^{ième} élément du tableau PL/SQL.
Si le n^{ième} élément n'existe pas, aucune erreur ne survient.

tab_salaire.DELETE(3) ;

DELETE(m,n) supprime les éléments de m à n
si (m > n) ou (m est null) ou (n est null) alors DELETE(m,n) n'agit pas

tab_salaire.DELETE(5,10) ;

5.8. Variable référencée à une variable de même type

L'attribut %TYPE spécifie pour une variable le type de donnée de la variable qui sert de référence.

nom_variable nom_variable_ref%TYPE ;

```
DECLARE

   sal      NUMBER(11,2) ;
   comm  sal%TYPE ;

BEGIN
....
END ;
```

Faire attention à l'ordre dans lequel sont déclarées les variables et les constantes.

```
DECLARE

   comm    NUMBER(8,2)  :=  Sal  * 0.2  ;     -- Incorrect. Déclarer la
   sal        NUMBER(8,2 ) :=  10000 ;          -- variable sal avant usage

BEGIN
   ....
END ;
```

5.9. Visibilité des variables

Une variable déclarée dans un bloc est connue dans celui-ci, ainsi que dans ses blocs fils (blocs imbriqués). Par contre, elle ne l'est pas dans les autres blocs (pères et frères).

```
DECLARE
   A  VARCHAR2(6) ;
   B   NUMBER(3) ;
   C   NUMBER(5,2) ;

BEGIN
   /* Les identifiants valides dans ce bloc sont  A VARCHAR2(6) , B
      NUMBER(3), et C NUMBER(5,2) */

    DECLARE
    B VARCHAR2(10) ;
    BEGIN
    -- Les identifiants valides dans ce bloc sont A  VARCHAR2(6),
    -- B VARCHAR2(1) et C  NUMBER(5,2)
    ....
          DECLARE
          Y  REAL ;
          BEGIN
          -- Les identifiants valides dans ce bloc sont  A VARCHAR2(6) ,
          -- B VARCHAR2(10), C NUMBER(5,2), Y REAL
          .....
          END ;

    END ;

    DECLARE
    Z  BOOLEAN ;
    BEGIN
    -- Les identifiants valides dans ce bloc sont  A VARCHAR2(6) ,
    -- B NUMBER(3), C NUMBER(5,2) et Z BOOLEAN
    ....
    END ;
END ;
```

5.10. Les « bind » variables

Une fois déclarées elles sont visibles pour tous les programmes de la session.

Déclaration

SQL> VAR nom_variable [NUMBER | CHAR | CHAR (n) |
VARCHAR2 (n) | REFCURSOR]]

Utilisation

En préfixant avec '**:**'

Affectation de valeurs

SQL> EXECUTE **:**nom_variable :=valeur

Affichage

SQL> PRINT nom_variable

Exemple

SQL> VAR no_emp NUMBER
*SQL> SELECT **
FROM E_EMP
WHERE no = :no_emp;

6. STRUCTURES DE CONTROLE

6.1. Traitements Conditionnels

Les traitements conditionnels permettent de contrôler l'exécution d'instructions en fonction de conditions.

Les opérateurs utilisés dans les conditions sont les mêmes que dans SQL :
= , > , < , >= , <= , <> , != , IS NOT NULL , IN , LIKE, BETWEEN , AND , OR ...

6.1.1. IF…THEN...END IF

C'est la structure conditionnelle la plus simple

Syntaxe

```
IF Condition  THEN
      Sequence d'instructions;
END IF;
```

La séquence d'instructions est exécutée si la condition est évaluée à TRUE.

Dans le cas où la condition est évaluée à FALSE ou NULL, la séquence d'instructions n'est pas exécutée.

6.1.2. IF…THEN...ELSE...END IF

Syntaxe

```
IF Condition  THEN
      Sequence1 d'instructions;
ELSE
      Sequence2 d'instructions;
END IF;
```

La séquence1 d'instructions est exécutée si la condition est évaluée à TRUE.
La séquence2 d'instructions est exécutée si la condition est évaluée à FALSE ou NULL.

6.1.3. IF…THEN...ELSIF...ELSE...END IF

Syntaxe

```
IF   Condition1 THEN
        Sequence1 d'instructions;
ELSIF Condition2 THEN
        Sequence2 d'instructions;
ELSIF Condition N THEN
        SequenceN d'instructions;
ELSE
        Sequence d'instructions;
END IF;
```

La clause ELSE est optionnelle.

Si la condition1 est évaluée à TRUE ,la séquence1 d'instructions est exécutée.

Si la condition1 est évaluée à FALSE ou NULL, la clause ELSIF teste la condition2.

Si celle ci est aussi évaluée à FALSE ou NULL, la clause ELSIF teste la condition3 et ainsi de suite jusqu'à la conditionN.

Si une des conditions de la clause ELSIF est évaluée à TRUE ,la séquence d'instructions correspondante est exécutée. Les conditions des clauses ELSIF suivantes ne seront pas testées.

Si aucune des conditions ELSIF n'est évaluée à TRUE, alors la séquence d'instructions de la clause ELSE est exécutée.

6.2. Traitements itératifs

6.2.1. Boucle LOOP ... END LOOP

Boucle de base qui permet la répétition d'une séquence d'instructions.

Syntaxe

```
[<<LABEL>>]
LOOP
   .....
   EXIT LABEL WHEN condition ;
   ....
END LOOP  [LABEL] ;
```

<<LABEL>> est un identifiant non déclaré de la boucle. Utilisé, il doit apparaître au commencement de la boucle et optionnellement à la fin de celle-ci.

Utilisez la commande EXIT [LABEL] [WHEN Condition] pour sortir de la boucle :

EXIT force la sortie de la boucle sans conditions.

EXIT WHEN permet la sortie de la boucle si la condition est vérifiée.

EXIT <<LABEL>> force la sortie de la boucle identifiée par <<LABEL>>.

```
<<Nom_boucle>>
   LOOP
        LOOP
        ...
        EXIT Nom_Boucle  WHEN condition1 ;
        END LOOP;
   END LOOP Nom_boucle ;
```

Si la condition 1 est vérifiée , sortie des deux boucles
Les opérateurs utilisés dans les conditions sont les mêmes que dans SQL :
= , > , < , >= , <= , <> , != , IS NOT NULL , IN , LIKE, BETWEEN , AND , OR ...

Pour afficher à l'écran un résultat, utilisez la fonction PUT_LINE du package intégré DBMS_OUTPUT de la façon suivante :

```
DBMS_OUTPUT.PUT_LINE('Résultat='||variable) ;
```

Pour pouvoir utiliser ce package, il faut positionner la variable d'environnement SERVEROUTPUT sur ON de la façon suivante :

```
SET SERVEROUTPUT ON
```

Exemple : Afficher à l'écran les 10 premiers entiers positifs non nuls

```
SET SERVEROUTPUT ON

DECLARE
        nombre  NUMBER(2) := 1;
BEGIN
        LOOP
        DBMS_OUTPUT.PUT_LINE(nombre);
           nombre := nombre + 1;
            EXIT WHEN nombre > 10;
      END LOOP;
END;
/
```

6.2.2. Boucle WHILE … LOOP... END LOOP

Tant que la condition de la clause WHILE est vérifiée, le contenu de la boucle est exécuté.

Le nombre d'itérations de la boucle dépend de la condition et n'est donc pas connu jusqu'à l'achèvement de la boucle.

Les opérateurs utilisés dans les conditions sont les mêmes que dans SQL : = , > , < , >= , <= , <> , != , IS NOT NULL , IS NULL ,IN , LIKE, BETWEEN, AND, OR ...

Syntaxe

```
WHILE condition
LOOP
… ;
END LOOP;
```

Exemple : Déterminez N tel que la somme de N premiers entiers 1+2+3+...+N < 50 et afficher N à l'écran.

```
DECLARE

        total  NUMBER(4) := 0;
        N      NUMBER(4) := 0;

BEGIN
        WHILE (total + N + 1) < 50
           LOOP
              N := N + 1;
              total := total + N;
        END LOOP;
        DBMS_OUTPUT.PUT_LINE ('Valeur de N: '||N);
END;
/
```

6.2.3. Boucle FOR … LOOP ... END LOOP

Le nombre d'itérations est connu avant d'entrer dans la boucle.

Syntaxe

```
   FOR Compteur IN [REVERSE] Borne inférieure .. Borne_supérieure
   LOOP
… ;
   END LOOP;
```

Règles

La variable compteur est déclarée implicitement. Elle ne doit pas être définie dans la section DECLARE .

Il est interdit de modifier la variable compteur en lui assignant une valeur :

```
FOR  Compteur IN 1..20
   LOOP
         Compteur := 5;    -- Incorrect
END LOOP;
```

Borne_inférieure et borne_supérieure peuvent être des variables, des constantes ou des expressions.

```
Compteur IN –5 .. 10
Code IN ASCII('A') .. ASCII('Z')
Compteur IN Premier .. Dernier
```

Sans l'option REVERSE, la variable compteur varie de borne_inférieure à borne_supérieure avec un pas d'incrémentation de 1.

Avec l'option REVERSE, la variable compteur varie de borne_supérieure à borne_inférieure avec un pas d'incrémentation de -1.

PL/SQL n'a pas de structure pour un pas d'incrémentation différent de 1, mais il est facile d'en construire une :

Exemple

Pas d'incrémentation de 4

Construire à l'aide une boucle FOR compteur IN ... LOOP dont compteur varie de 4 à 32 , l'équivalent d'une boucle FOR ... LOOP avec un pas d'incrémentation de 4.

Afficher les différentes valeurs prises par la variable qui s'incrémente de 4 à chaque itération de la boucle FOR ... LOOP.

```
SET SERVEROUTPUT ON

BEGIN

        FOR compteur IN 4 .. 32
           LOOP
              IF MOD(compteur,4) = 0 THEN
                    DBMS_OUTPUT.PUT_LINE(compteur);
              END IF;
        END LOOP;
END;
/
```

La variable compteur n'est définie que pour la boucle.

```
FOR  Compteur  IN 1 .. 10
      LOOP
         ... ;
END LOOP;

Var :=  Compteur;  -- Incorrect
```

Deux boucles imbriquées peuvent utiliser le même nom de variable.
Pour référencer le nom de variable de la boucle extérieure, utiliser un label.

```
<< Nom_boucle>>
FOR  Compteur  IN 1 .. 30
   LOOP
        FOR compteur  IN 1..10
           LOOP
           ...
           IF Nom_boucle.Compteur  > 5  THEN
            ...
        END LOOP;
END LOOP Nom_boucle;
```

Exemple :

Calculer la factorielle de 12 = 1x2x3x4x5x...x12 et afficher le résultat à l'écran

```
SET SERVEROUTPUT ON
DECLARE

   N           NUMBER(2) := 12;
   Resultat    NUMBER(12) := 1;

BEGIN

   FOR   J  IN 1 .. N
   LOOP
         Resultat := Resultat * J ;
   END LOOP ;

   DBMS_OUTPUT.PUT_LINE (N||' ! = '|| resultat);

END ;
/
```

6.3. Traitements séquentiels

GOTO

PL/SQL permet de se « brancher » sur un label sans conditions avec l'instruction GOTO.

Syntaxe

GOTO LABEL

Règles

LABEL est unique et doit précéder une commande exécutable ou un bloc PL/SQL.

```
DECLARE
   Sortir  BOOLEAN ;
BEGIN

   FOR J IN 1 .. 10
      LOOP
         IF SortirTHEN
            GOTO Nom_label ;
         END IF ;
         ...
<<Nom_label>>-- Incorrect car Nom_label doit précéder une commande
                -- exécutable
   END LOOP ;
END;
```

Pour corriger l'exemple précédent, utilisez l'instruction NULL :

```
DECLARE
   Sortir  BOOLEAN ;
BEGIN
   FOR    J IN 1 .. 10
      LOOP
            IF SortirTHEN
               GOTO Nom_label ;
            END IF ;
            ....
            <<Nom_label>>--  Correct car suivi de la commande
                             --  exécutable NULL
            NULL ;
      END LOOP ;
   END ;
    /
```

7. LES CURSEURS

7.1. Définition et Type

Un curseur est une zone mémoire de taille fixe utilisée par le noyau pour analyser et interpréter tout ordre SQL (Définie dans le fichier Init.ora (option open_cursor)).

Il existe deux types de curseurs :

Les curseurs implicites :

Ils sont générés et gérés par le noyau pour chaque ordre SQL (SELECT, UPDATE, DELETE, INSERT) .

Les curseurs explicites :

Ils sont créés en programmation PL/SQL et utilisés par le développeur pour gérer les requêtes SELECT qui doivent rapporter plusieurs lignes.

Même si l'on est sûr qu'une requête ne ramènera qu'une seule ligne, il est conseillé d'utiliser systématiquement un curseur explicite plutôt que le SELECT …INTO au cas où cette requête ramènerait plusieurs lignes suite à une mise à jour, ce qui génèrerait l'erreur ORA-01422 (TOO_MANY_ROWS).

7.2. Utilisation des curseurs explicites

Ils sont définis dans la section DECLARE d'un bloc PL/SQL, d'une procédure, d'une fonction ou d'un package par son nom et sa requête spécifique.

Trois commandes permettent de gérer les curseurs explicites dans les programmes :

OPEN ouvre le curseur
FETCH exécute la requête en ramenant une ligne à la fois.
Pour ramener toutes les lignes de la requête, il faut exécuter la commande FETCH autant de fois qu'il y a de lignes à ramener.
CLOSE ferme le curseur

Plusieurs curseurs peuvent être utilisés en même temps.

7.2.1. La déclaration du curseur

Le curseur doit être déclaré dans la clause DECLARE avant toute utilisation suivant la syntaxe :

```
DECLARE

   CURSOR  Nom_curseur IS
   SELECT   nom_colonnes....
   FROM     nom_tables
   WHERE    condition(s) ;

BEGIN
   ....
END ;
/
```

DECLARE

CURSOR c_employe IS
SELECT no, nom, salaire
FROM e_emp
WHERE salaire > 1000 ;

BEGIN
.....
END ;
/

7.2.2. L'ouverture d'un curseur

L'ouverture d'un curseur s'effectue dans la section BEGIN.

Elle réalise l'allocation mémoire du curseur, l'analyse sémantique et syntaxique de l'ordre SELECT, le positionnement de verrous éventuels si l'option FOR UPDATE est utilisée avec l'ordre SELECT. (voir chapitre sur la gestion des transactions)

Syntaxe

OPEN Nom_curseur ;

💣 Il n'est pas permis d'ouvrir un curseur déjà ouvert.

```
DECLARE
   CURSOR c_employe  IS
   SELECT   no, nom, sal
   FROM     e_emp
   WHERE    sal > 1000 ;
BEGIN
   OPEN c_employe ;
   ....
END ;
```

7.2.3. La fermeture d'un curseur

La fermeture d'un curseur s'effectue dans les sections BEGIN ou EXCEPTION Elle libère les ressources utilisées par le curseur.

Syntaxe

CLOSE Nom_curseur ;

💣 Toujours fermer un curseur lors de la fin de son utilisation.

7.2.4. Traitement des lignes d'un curseur

Le traitement des lignes d'un curseur s'effectue dans la section BEGIN à l'aide de la commande FETCH .

La commande FETCH ramène une à une les lignes d'enregistrements. A chaque fois que la commande FETCH est exécutée, le curseur avance à la ligne suivante.

Syntaxe

FETCH Nom_curseur INTO Liste_de_variables ;

FETCH Nom_curseur INTO Variable_type_enregistrement ;

Il est nécessaire d'avoir déclaré et ouvert le curseur avant son utilisation**.**

7.3. Les attributs d'un curseur

Les attributs d'un curseur sont des variables gérées lors de l'exécution d'un ordre.
Ils peuvent être testés afin de modifier le déroulement d'une exécution.
Ces attributs sont %NOTFOUND, %FOUND, %ROWCOUNT, %ISOPEN.

Curseurs explicites : Ces attributs fournissent des informations sur l'exécution d'une requête ramenant plusieurs enregistrements**.**

Curseurs implicites : Oracle ouvre un curseur implicite pour gérer chaque ordre SQL qui n'est pas lié à un curseur déclaré explicitement.

Si on ne peut pas utiliser les ordres OPEN, FETCH et CLOSE pour un curseur implicite, on peut par contre utiliser les attributs pour accéder à des informations sur la **dernière commande** SQL exécutée.

7.3.1. L'attribut %FOUND

L'attribut %FOUND est de type booléen.

Il prend la valeur VRAI si la dernière commande FETCH a ramené une ligne.

L'attribut est utilisé avec une syntaxe différente suivant que le curseur est explicite ou implicite.

Curseur implicite : La syntaxe est SQL%FOUND

Curseur explicite : La syntaxe est Nom_curseur%FOUND

Exemple

Afficher à l'écran les noms, salaires et le nom du service des employés dont le salaire est supérieur à 1000.
Ordonner l'ensemble par salaire décroissant et par nom.

```
SET SERVEROUTPUT ON

   DECLARE

            CURSOR c_emp IS
            SELECT   e.nom,
                     e.salaire,
                     s.nom
            FROM     e_emp e, e_service s
            WHERE    e.service_no = s.no
            AND      e.salaire > 1000
            ORDER BY e.salaire DESC,e.nom;

            nom_emp   e_emp.nom%TYPE;
            sal_emp   e_emp.salaire%TYPE;
            nom_serv  e_service.nom%TYPE;

   BEGIN

            OPEN c_emp;

      LOOP
               FETCH c_emp
               INTO   nom_emp,
                      sal_emp,
                      nom_serv;

            IF c_emp%FOUND THEN
               DBMS_OUTPUT.PUT_LINE('Nom : '||nom_emp||' - Service : '
||nom_serv|| ' - Salaire : '||sal_emp);
            ELSE
                  EXIT ;
            END IF ;
      END LOOP;
      CLOSE c_emp;
   END;
/
```

7.3.2. L'attribut %NOTFOUND

L'attribut %NOTFOUND est de type booléen.

Il prend la valeur VRAI si la dernière commande FETCH n'a pas ramené de ligne.

L'attribut est utilisé avec une syntaxe différente suivant que le curseur est explicite ou implicite.

Curseur implicite : La syntaxe est SQL%NOTFOUND.

Curseur explicite : La syntaxe est Nom_curseur%NOTFOUND

Exemple

Mettre à jour le salaire d'un employé et afficher un message si cet employé n'existe pas.

```
SQL> ACCEPT p_nom PROMPT 'entrer un nom :'
entrer un nom : Martin
SQL> BEGIN
  2  UPDATE e_emp
  3  SET SALAIRE =2000
  4  WHERE nom ='&p_nom';
  5  IF SQL%NOTFOUND THEN
  6    DBMS_OUTPUT.PUT_LINE('Cet employé n''existe pas');
  7  END IF;
  8  END;
  9  /
```

7.3.3. L'attribut %ROWCOUNT

L'attribut %ROWCOUNT est de type numérique.

Contient le nombre de lignes ramenées par la ou les commandes FETCH dans le cas d'un curseur explicite, ou le nombre de lignes concernées par la dernière commande SQL implicite.

L'attribut est utilisé avec une syntaxe différente suivant que le curseur est explicite ou implicite.

Curseur implicite : La syntaxe est SQL%ROWCOUNT

Curseur explicite : La syntaxe est Nom_curseur%ROWCOUNT

Exemple

Pour les employés qui ont un salaire inférieur à 1500, augmenter celui-ci de 120. Afficher à l'écran le nombre d'employés qui ont été augmentés.

```
SET SERVEROUTPUT ON

BEGIN

   UPDATE  e_emp
   SET     salaire = salaire + 120
   WHERE   salaire < 1500 ;

   DBMS_OUTPUT.PUT_LINE('Nombre d''employés augmentés : '
||SQL%ROWCOUNT) ;

END ;
/
```

7.3.4. L'attribut %ISOPEN

L'attribut %ISOPEN est de type booléen.

Il est utilisé pour déterminer si un curseur est ouvert ou fermé

Curseur implicite : %ISOPEN est toujours évalué à **FALSE.**

Oracle ferme le curseur SQL automatiquement après l'exécution de la commande SQL associée au curseur.

Curseur explicite : La syntaxe est Nom_curseur%ISOPEN

7.4. Curseurs paramétrés

Il est possible de passer des paramètres au curseur. Cela permet de réutiliser le même traitement de curseur avec une valeur d'entrée différente.

La syntaxe de déclaration est la suivante :

```
DECLARE
    ...
    CURSOR  nom_curseur (param1  TYPE_param1
                         [,param2 TYPE_param2 [,.....])
       IS
       SELECT ..... (utilisant le(s) paramètre(s) )
    ...
```

Les noms des paramètres qui ont servi pour déclarer le curseur ne peuvent pas être utilisés ailleurs dans le programme.

Les types de paramètres LONG et LONGRAW sont invalides.

On ne doit pas spécifier la longueur maximale pour les autres types mais VARCHAR2, NUMBER, ...

L' utilisation est la suivante:

```
   BEGIN
      ...
      OPEN nom_curseur( valeur1 [,valeur2 [,....    ]);
…
FETCH nom_curseur
INTO … ;
CLOSE nom_curseur ;
```

Ou encore

```
   FOR compteur IN nom_curseur( valeur1 [,valeur2 [,....    ])

   LOOP
   …
   END LOOP ;
```

Exemple

```
DECLARE
CURSOR c_cs(p_no e_service.no%type) IS
SELECT *
FROM e_service
WHERE no >p_no
ORDER BY no;

BEGIN

FOR compteur IN c_cs(25)
LOOP
   DBMS_OUTPUT.PUT_LINE(compteur.no);
END LOOP;
END;
/
```

7.5. Boucles et Curseurs

L'utilisation d'un curseur dans une boucle FOR simplifie l'écriture des commandes qui permettent de gérer le curseur.

La boucle FOR gère la gestion du curseur : les commandes OPEN, FETCH et CLOSE sont alors inutiles.

Syntaxe

FOR enreg IN nom_curseur ;

LOOP

....

A chaque itération de la boucle FOR, les valeurs des colonnes du curseur nom_curseur sont 'fetchées' dans la variable enreg définie implicitement comme de type RECORD et équivalente à la variable qui serait explicitement déclarée de la manière suivante :

enreg nom_curseur%ROWTYPE ;

A la sortie de la boucle FOR, le curseur nom_curseur est automatiquement fermé, même si la sortie se fait prématurément avec la commande EXIT.

Exemple : *Affichez à l'écran le nom, le salaire et le nom du service des employés qui exercent la fonction de Magasinier. Ordonnez l'ensemble par salaire décroissant.*

```
SET SERVEROUTPUT ON

DECLARE

        CURSOR c_employe IS
        SELECT e.nom     nom_emp,
               e.salaire sal_emp,
               s.nom     serv
        FROM   e_emp e, e_service s
        WHERE  e.service_no = s.no
        AND    e.titre = 'Magasinier'
        ORDER BY e.salaire DESC;
BEGIN
        FOR c_rec IN c_employe
        LOOP
            DBMS_OUTPUT.PUT_LINE ('Nom : '||c_rec.nom_emp||
'-Service : ' ||c_rec.serv|| ' - Salaire : '||c_rec.sal_emp);
        END LOOP;
END;
/
```

7.6. La clause CURRENT OF

La clause WHERE CURRENT OF permet de faire référence au positionnement dans un curseur afin de traiter la ligne correspondante (UPDATE, DELETE).

Il est nécessaire de réserver la ligne lors de la déclaration du curseur par le positionnement d'un verrou d'intention (FOR UPDATE OF...).

Exemple : *Utilisation d'un curseur pour augmenter de 10% les salaires inférieurs à 2000.*

```
DECLARE
CURSOR c_emp IS
SELECT *
FROM e_emp
FOR UPDATE OF salaire;

BEGIN

FOR compteur IN c_emp
LOOP
IF  compteur.salaire<2000 THEN
   UPDATE e_emp
   SET salaire =salaire *1.1
   WHERE CURRENT OF c_emp;
END IF;
END LOOP;
END;
/
```

7.7. La clause RETURN d'un curseur

Grâce à cette clause il est possible de créer des curseurs dans des packages et d'en cacher l'implémentation aux développeurs (voir Packages)

Exemple

Spécification :

CURSOR c_cs(p_no e_service.no%TYPE) RETURN e_service%ROWTYPE ;

Implémentation (partie cachée) :

CURSOR c_cs(p_no e_service.no%TYPE) RETURN e_service%ROWTYPE IS
*SELECT **
FROM e_service
WHERE no >p_no ;

7.8. Variables de type REF CURSOR

(depuis la version PLSQL 2.3).

Une variable curseur, contrairement à un curseur, est dynamique car elle n'est pas rattachée à une requête spécifique.

7.8.1. Déclarer une variable Curseur

La création d'une variable curseur s'effectue en deux étapes

1) Définir un type REF CURSOR

TYPE ref_type_nom IS REF CURSOR RETURN type_return ;

type_return représente soit un record, soit une ligne de table basée.

Exemple : type_return représente une ligne de la table e_emp

TYPE emptype IS REF CURSOR RETURN e_emp%ROWTYPE ;

2) Déclarer une variable de type REF CURSOR

Déclarer la variable emp_cs

DECLARE

TYPE emptype IS REF CURSOR RETURN e_emp%ROWTYPE ;
emp_cs emptype ;

7.8.2. Gérer une variable curseur

On utilise les commandes OPEN-FOR, FETCH, CLOSE pour contrôler les variables curseurs.

```
OPEN {nom_var_curseur | :host_nom_var_curseur}
FOR ordre_select ;
……..
FETCH nom_var_curseur
INTO var_hôtes ;
……..
CLOSE nom_curseur ;
```

Exemple : *Utilisation d'un curseur référencé pour retourner des enregistrements choisis par l'utilisateur*

```
DECLARE
TYPE rec_type IS RECORD (no e_client.no%TYPE,nom e_client.nom%TYPE);
enreg rec_type;
TYPE  refcur_type IS REF CURSOR RETURN enreg%TYPE ;
curref  refcur_type ;
enreg_emp curref%ROWTYPE ;
v_choix NUMBER(1) := &choix ;

BEGIN
IF v_choix=1 THEN
   OPEN curref FOR
   SELECT no,nom
   FROM e_client;
ELSIF v_choix=2 THEN
   OPEN curref FOR
   SELECT no,nom
   FROM e_emp;
ELSIF v_choix=3 THEN
   OPEN curref FOR
   SELECT no,nom
   FROM e_service;
ELSIF v_choix=4 THEN
   OPEN curref FOR
   SELECT no,nom
   FROM e_continent;
ELSIF v_choix=5 THEN
   OPEN curref FOR
   SELECT no,nom
   FROM e_produit;
END IF;
LOOP
   EXIT WHEN v_choix NOT BETWEEN 1 AND 5;
   FETCH curref
   INTO enreg ;
   EXIT WHEN curref%NOTFOUND ;
   DBMS_OUTPUT.PUT_LINE('No : '||enreg.no||'. Nom : ' ||enreg.nom) ;
END LOOP ;
END ;
/
```

8. GESTION DES ERREURS

Les sources d'erreurs lors de l'exécution d'une application sont multiples :

- ✓ erreur de codage,
- ✓ erreur système,
- ✓ violation d' une règle Oracle,
- ✓ dépassement des limites du système d'exploitation.

Il est impossible d'anticiper toutes les erreurs qui pourraient survenir.

Néanmoins, PL/SQL met à la disposition de l'utilisateur un mécanisme de gestion des erreurs qui lui permet de planifier les traitements à effectuer (abandon, continuation des traitements) lorsque surviennent certains types d'erreurs.

On parle alors **de gestion des EXCEPTIONS.**

Chaque erreur Oracle générée par le noyau a un code erreur (SQLCODE) .

Un nom d'exception est :

- soit prédéfini en interne par le noyau Oracle
- soit défini par l' utilisateur. Il peut être associé à un code d'erreur.

Le développeur peut également définir ses propres erreurs qu'il doit déclencher explicitement et auxquelles il attribue un nom personnalisé.

Il y a deux méthodes pour gérer les erreurs:

- Si le nom d'exception existe:

```
BEGIN
……
EXCEPTION
  WHEN Nom_exception THEN

            Instructions;
END;
/
```

- Si le nom d'exception n'existe pas:

```
BEGIN
……
EXCEPTION
  WHEN OTHERS THEN

        IF SQLCODE =Numero_code1 THEN
           Instructions;
        END IF;

      IF SQLCODE =Numero_code2 THEN
           Instructions;
        END IF;

END;
/
```

☞ S'il y a d'autres exceptions, l'exception **OTHERS** doit figurer en dernière position. **OTHERS** concerne toutes les erreurs qui peuvent être générées et les inhibe à défaut d'instructions explicites.

Lorsque le traitement a été transféré dans la section EXCEPTION du bloc PL/SQL, il n'est pas possible de retourner dans le corps principal du bloc.

Pour pouvoir continuer, la solution consiste à utiliser des blocs imbriqués.

8.1. Les exceptions prédéfinies

Les erreurs les plus courantes ont été redéfinies dans PL/SQL.
Les autres pourront être gérées par l'utilisateur grâce aux gestionnaires OTHERS et EXCEPTION_INIT.

Liste des exceptions prédéfinies dans PL/SQL

Nom exception	Erreur Oracle	Valeur SQLCODE
CURSOR_ALREADY_OPEN	ORA-06511	- 6511
DUP_VAL_ON_INDEX	ORA-00001	- 1
INVALID_CURSOR	ORA_01001	- 1001
INVALID_NUMBER	ORA_01722	- 1722
LOGIN_DENIED	ORA_01017	- 1017
NO_DATA_FOUND	ORA-01403	+ 100
NOT_LOGGED_ON	ORA-01012	- 1012
PROGRAM_ERROR	ORA-06501	- 6501
ROWTYPE_MISMATCH	ORA-06504	- 6504
STORAGE_ERROR	ORA-06500	- 6500
TIMEOUT_ON_RESOURCE	ORA-00051	- 51
TOO_MANY_ROWS	ORA-01422	- 1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	- 1476

CURSOR_ALREADY_OPEN : tentative d'ouverture d'un curseur déjà ouvert. Vous devez d'abord fermer le curseur avant de l'ouvrir à nouveau.

DUP_VAL_ON_INDEX : violation de l'unicité lors d'une mise à jour (détectée au niveau de l'index UNIQUE).

INVALID_CURSOR : opération incorrecte sur un curseur, comme par exemple la fermeture d'un curseur qui n'a pas été ouvert.

INVALID_NUMBER : échec de la conversion d'une chaîne de caractères en numérique.

LOGIN_DENIED : connexion à la base échouée car le nom utilisateur / mot de passe est invalide.

NO_DATA_FOUND : déclenché si la commande SELECT INTO ne retourne aucun enregistrement ou si on fait référence à un enregistrement non initialisé d'un tableau PL/SQL.

NOT_LOGGED_ON : tentative d'accès à la base sans être connecté.

PROGRAM_ERROR : problème général dû à PL/SQL.

ROWTYPE_MISMATCH : survient lorsque une variable curseur d'un programme hôte retourne une valeur dans une variable curseur d'un bloc PL/SQL qui n'a pas le même type.

STORAGE_ERROR : problème de ressource mémoire dû à PL/SQL

TIMEOUT_ON_RESOURCE : dépassement de temps dans l'attente de libération des ressources (lié aux paramètres systèmes de la base).

TOO_MANY_ROWS : la commande SELECT INTO retourne plus d'un enregistrement.

VALUE_ERROR : erreur arithmétique, de conversion, de troncature, de contrainte de taille. Par exemple, si vous affectez une chaîne de caractère de taille plus grande que la taille de la variable qui la reçoit, PL/SQL abandonne l'affectation et déclenche l'exception VALUE_ERROR.

ZERO_DIVIDE : tentative de division par zéro. Le résultat est indéfini.

8.2. Définir une exception non associée à une erreur

La commande **RAISE** interrompt l'exécution normale du bloc PL/SQL et transfère la suite des opérations au gestionnaire d'exception.

Exemple : Gérer un seuil de stock :

```
DECLARE
   Stock_alert      EXCEPTION ;
   Stock_Seuil      NUMBER(4) ;
   Qte_stock        NUMBER(4) ;

BEGIN
   ....
   IF Qte_Stock  < Stock_seuil   THEN
      RAISE  Stock_alert ;
   END IF ;
   ...
EXCEPTION

   WHEN  Stock_alert  THEN
      ......
END ;
/
```

8.3. Définir une exception associée à une erreur

Une exception doit être déclarée dans la section DECLARE d'un bloc PL/SQL, une procédure, une fonction ou un package.

Son nom y est suivi du mot clé : **EXCEPTION.**

PRAGMA EXCEPTION_INIT est une directive compilée qui demande au compilateur d'associer le nom d'une exception avec le numéro d'une erreur Oracle suivant la syntaxe :

```
PRAGMA  EXCEPTION_INIT( Nom_exception, Numero_erreur_Oracle) ;
```

Exemple :

Déclarer l'exception Privileges_insuffisants et l'associer à l'erreur -1031:

```
DECLARE

   Privileges_insuffisants   EXCEPTION ;
   PRAGMA  EXCEPTION_INIT(Privileges_insuffisants , -1031) ;

BEGIN
   ....
EXCEPTION

   WHEN  Privileges_insuffisants  THEN    -- exception atteinte sans la
                                          -- commande explicite RAISE
      Traitements ;
END ;
/
```

8.4. Créer une erreur personnalisée

Il est possible de créer ses propres erreurs avec numéro de code et message associé en utilisant la syntaxe:

RAISE_APPLICATION_ERROR(numéro_erreur, message)

où numéro_erreur est compris entre -20000 et –20999.

Exemple

```
SET SERVEROUTPUT ON

DECLARE
nom_emp e_emp.nom%TYPE;
BEGIN
SELECT nom
INTO nom_emp
FROM e_emp
WHERE no =&p_no;
IF nom_emp = 'DUMAS' THEN
   RAISE_APPLICATION_ERROR(-20200,'C''EST DUMAS!');
ELSE
DBMS_OUTPUT.PUT_LINE(nom_emp) ;
END IF;
END;
/
```

8.5. Utilisation des fonctions SQLCODE et SQLERRM

Vous pouvez utiliser les fonctions SQLCODE et SQLERRM pour trouver quelle erreur a été générée par Oracle et le message qui lui est associé.

- ✓ La fonction SQLCODE retourne le numéro de l'erreur Oracle.
  Ce nombre est négatif sauf pour l'erreur NO_DATA_FOUND. (SQLCODE = +100).

- ✓ La fonction SQLERRM retourne le texte du message d'erreur.
  Ce message commence avec le code Oracle de l'erreur.

Exceptions définies par l'utilisateur avec EXCEPTION_INIT

SQLCODE retourne le numéro d'erreur défini.
SQLERRM retourne le message d'erreur défini.

Exceptions définies par l'utilisateur sans EXCEPTION_INIT

SQLCODE = + 1
SQLERRM = User-defined- Exception.

Aucune exception n'est survenue

SQLCODE = 0
SQLERRM = ORA-0000 : normal, successful completion

SQLERRM(numero erreur)

SQLERRM(numero erreur) = message de l'erreur

Si numero erreur est positif et différent de + 100
SQLERRM = User-defined Exception

SQLERRM(+ 100) = ORA-01403 : no data found

SQLERRM(0) = ORA-0000 : normal, successful completion

Exemple

Afficher à l'écran les messages pour les numéros d'erreurs allant de -20 à - 30

```
SET SERVEROUTPUT ON
   DECLARE
      message_erreur  VARCHAR2(100);

   BEGIN

      FOR Numero_erreur IN 20 .. 30
         LOOP
            message_erreur := SQLERRM( - Numero_erreur ) ;
            DBMS_OUTPUT.PUT_LINE(message_erreur) ;
      END LOOP ;
   END ;
   /
```

☞ **SQLCODE et SQLERRM** ne peuvent pas être utilisés directement dans un ordre SQL.
Vous devrez passer par l'intermédiaire de deux variables auxquelles seront assignées les valeurs SQLCODE et SQLERRM.

```
INSERT INTO erreurs VALUES( SQLCODE,SQLERRM) ; -- Incorrect
```

9. GESTION DES TRANSACTIONS

> Une transaction est un ensemble de mises à jour de la base dépendantes. Soit elles sont toutes validées, soit aucune ne l'est.

Une transaction débute avec la première commande SQL exécutée.

Par exemple, l'ouverture d'une session débute une transaction.

La première commande SQL exécutée qui suit la fin d'une transaction débute une nouvelle transaction.

Pour permettre à différents utilisateurs de travailler simultanément et partager les mêmes ressources, Oracle contrôle l'accès concurrentiel des utilisateurs aux mêmes données en utilisant des verrous (LOCKS).

Un verrou donne temporairement à un utilisateur la maîtrise des mises à jour des données sur une table ou enregistrements basés.

Tout autre utilisateur ne peut intervenir sur les mêmes données tant qu'elles ne sont pas déverrouillées.

La pose d'un verrou est implicitement effectuée par Oracle.

Cependant, chaque utilisateur a la possibilité de prendre à son compte cette opération et choisir un mode de verrouillage plus approprié tels que les modes ROW SHARE et EXCLUSIVE

Les principaux ordres SQL de gestion des transactions sont les suivants :

COMMIT	valide une transaction
ROLLBACK	annule une transaction
SAVEPOINT	débute une sous transaction
ROLLBACK TO	annule une sous transaction
LOCK TABLE	ordre de verrouillage explicite.

Exemple de transaction

→ début de session et de la transaction 1
CONNECT Username/password ;
.....
UPDATE (table) ;
.....
INSERT INTO (table) ;
.....
COMMIT ;
→ fin de la transaction 1
→ début de la transaction 2
.....
INSERT INTO (table) ;
.....
→ début sous-transaction 2
SAVEPOINT SV1 ;
.....
DELETE FROM (table) ;
.....
ROLLBACK TO SV1 ;
→Annulation sous-transaction 2
.....
COMMIT; *-- **Valide tous le reste de la transaction 2***
→fin de la transaction 2

9.1. Utilisation de la commande COMMIT

La commande COMMIT valide définitivement les mises à jour opérées dans la base lors d'une transaction.
Jusqu'à la validation définitive, les autres utilisateurs ne peuvent pas voir les données modifiées mais telles qu'elles étaient avant les changements (lecture cohérentes).

```
BEGIN
   .....
   UPDATE  e_service        --  pose d'un verrou sur les
   SET nom = 'Marketing'    --  données manipulées dans
   WHERE   no = 32 ;        --  cet ordre SQL

   .....
   COMMIT ;    -- Validation définitive de la transaction
               -- Suppression du verrou
END ;
```

9.2. Utilisation de la commande ROLLBACK

La commande ROLLBACK annule définitivement les mises à jour opérées dans la base lors d'une transaction.

```
DECLARE

   no_emp  INTEGER ;

BEGIN
   ...
   INSERT INTO e_emp(no_emp, .....) ;-- si l'ordre INSERT tente d'insérer une
                                     -- valeur no_emp déjà existante dans la
                                     -- table e_emp, l'exception
                                     -- DUP_VAL_ON_INDEX est déclenchée
....
EXCEPTION
   WHEN DUP_VAL_INDEX  THEN
      ROLLBACK ;
END ;
```

9.3. Utilisation de SAVEPOINT

L'ordre SAVEPOINT découpe une transaction en sous-transactions.
On peut ainsi gérer l'annulation des mises à jour opérées dans la base lors de sous-transactions sans pour autant annuler l'ensemble de la transaction.

L'ordre SAVEPOINT est utilisé conjointement à l'ordre ROLLBACK TO.

```
DECLARE

   Num_emp     e_emp.no%TYPE ;

BEGIN
   ....
   UPDATE  e_emp
   SET         salaire = salaire * 0.12
   WHERE    Titre = 'Magasinier' ;
   .....
   SAVEPOINT     inserer ;
   INSERT INTO e_emp VALUES( Num_emp, ......) ;
   .....
EXCEPTION

   WHEN DUP_VAL_ON_INDEX  THEN
      ROLLBACK TO inserer ;       -- Cette commande annule toute la partie
                                  -- de la transaction qui débute après la
                                  -- marque du SAVEPOINT  inserer
END ;
```

Les noms de savepoint sont des identifiants non déclarés et peuvent être réutilisés dans la transaction.
Ceci a pour effet de déplacer le savepoint de son ancienne position à la courante.

Exemple

```
DECLARE
   Num_emp     e_emp.no%TYPE ;

BEGIN
   ....
   SAVEPOINT  point ;
   UPDATE  e_emp
   SET        salaire = salaire * 0.12
   WHERE   Titre = 'Magasinier' ;
   .....
   SAVEPOINT     point;          -- déplace point à cette position
   INSERT INTO e_emp VALUES( Num_emp, ......) ;
   .....
EXCEPTION

   WHEN DUP_VAL_ON_INDEX  THEN
      ROLLBACK TO point;         -- Cette commande annule toute la partie
                                 -- de la transaction qui débute après la
                                 -- marque du SAVEPOINT point
END ;
```

Rollback implicite

Après l'exécution d'un ordre INSERT, UPDATE ou DELETE , Oracle marque implicitement un SAVEPOINT.
Si l'ordre ne s'exécute pas normalement, Oracle effectue un ROLLBACK jusqu'au SAVEPOINT.

10. Le schéma

10.1. Définition

Un schéma est le regroupement des objets d'un utilisateur dans une même unité logique.

Il permet de construire l'ensemble des structures d'une application en une seule opération.

Le contrôle des dépendances entre les objets est réalisé à la fin de la création de tous les objets.

Syntaxe

CREATE SCHEMA AUTHORIZATION nom_schéma
CREATE TABLE nom_table...
CREATE VIEW nom_vue...
GRANT liste_privileges ON {nom_table | nom_vue} TO {user/role}

...

Il n'y a qu'un seul caractère d'exécution (« ; » ou « / ») à la fin du dernier ordre. Il indique la fin de création du schéma.

Attention : le nom du schéma ne peut être que celui de l'utilisateur propriétaire des objets créés.

Exemple

```
CREATE SCHEMA AUTHORIZATION cours1
CREATE TABLE e_emp (no NUMBER(7) NOT NULL,
                    nom VARCHAR2(25) NOT NULL,
                    dt_entree DATE NOT NULL,
                    no_service   NUMBER(2),
CONSTRAINT e_emp_no_pk PRIMARY KEY (no))
TABLESPACE user_data
STORAGE ( INITIAL 10K NEXT 10K PCTINCREASE 0)
GRANT SELECT, UPDATE ON e_emp TO cours2
CREATE TABLE e_service (no  NUMBER(2) NOT NULL,
                        nom VARCHAR2(15) NOT NULL)
TABLESPACE user_data
STORAGE ( INITIAL 10K NEXT 10K PCTINCREASE 0)
GRANT SELECT ON e_service TO cours2
/
```

10.2. Intéret d'un schéma

- Faciliter la gestion des objets utilisateur : si une seule opération échoue lors de la création du schéma (création d'une table, par exemple), il y a un rollback général : toutes les opérations sont annulées, aucun objet n'est créé.
- Faciliter l'administration des statistiques sur les objets : au lieu d'exécuter la commande ANALYZE pour chaque objet (table, index), on peut le faire pour tous les objets appartenant au schéma grâce à la procédure standard :

$ORACLE_HOME/RDBMS/ADMIN/DBMS_UTILITY.ANALYZE SCHEMA()

☞ **Contrainte** : tout est validé ou rien n'est validé. Technique difficile avec 350 tables, 140 vues, 70 Packages , etc.

Syntaxe :

```
SQL>EXECUTE DBMS_UTILITY.ANALYZE_SCHEMA ('nom_schéma',
                                {'COMPUTE' | 'ESTIMATE' | 'DELETE'};
```

Exemple

Générer des statistiques complètes sur les tables du schéma COURS1

```
SQL>EXECUTE DBMS_UTILITY.ANALYZE_SCHEMA('cours1',
                                                'COMPUTE') ;
```

10.2.1.Modifier un élément du schéma

2 méthodes possibles :

1) Supprimer et recréer l'objet : DROP ..., CREATE...

   Dans ce cas, il faudra créer à nouveau les privilèges.

2) Renommer l'objet : RENAME ... TO ....

   L'objet conserve ses privilèges d'origine.

3) On ne peut renommer ni un traitement catalogué ni un synonyme public ni un cluster.

✍ Dans tous les cas, les objets dépendants seront invalides et seront recompilés (Cf. Chapitre « Dépendance »).

11. LES TRAITEMENTS STOCKES DANS LA BASE

11.1. Définitions

Certains traitements sur les données peuvent être stockés dans la base.

Les traitements sont définis avec le langage PL/SQL et sont stockés dans les tables du dictionnaire de données.

4 types de traitements :

1. La **procédure** dont l'unique rôle est d'effectuer un traitement.
2. La **fonction** qui effectue un traitement pour renvoyer en retour une valeur.
3. Le **package** qui regroupe un ensemble de procédures et /ou de fonctions.
4. Le **trigger (déclencheur)** qui est déclenché automatiquement lors d' une mise à jour sur une table.

A la création de la procédure, fonction ou package, le traitement est stocké sous deux formes différentes dans le dictionnaire de données :

- Sous la forme de code source
- Sous la forme de code compilé

Lorsque le traitement est appelé, sont stockés en mémoire le code source et le code compilé.

Lorsqu'un élément du package est appelé, c'est l'ensemble du package qui est stocké en mémoire (source et compilé).

11.2. Intérêts des traitements stockés

- ✓ Analyse des requêtes SQL et des blocs PL déjà effectuée à l'exécution (Temps CPU).
- ✓ Diminution du trafic de communication (client/serveur). Le client envoie un appel au serveur. Tout le traitement est effectué sur le serveur qui ne renvoie au client que le résultat final.
- ✓ Réutilisation des traitements. Toutes les applications peuvent accéder au traitement.
- ✓ Gestion aisée de la confidentialité et des contrôles de cohérence des données. Il est possible de donner le droit de lancer un traitement sans donner le droit d'accès direct aux données.

11.3. Les procédures et fonctions

Elles peuvent être appelées :

- en mode interactif
- dans d'autres procédures ou fonctions stockées
- dans un trigger (déclencheur)
- dans une application ORACLE FORMS
- dans programme hôte travaillant en langage PRO*xx

11.3.1.Les procédures

Syntaxe de création / modification

```
CREATE [OR REPLACE] PROCEDURE [nom_user.]nom_procédure
( arguments IN type [, argument IN type, ... ] )
{ IS | AS }
[Variable_locale type_variable_locale ;]
BEGIN
{ contenu du bloc PL }
END [ nom_procédure ] ;
/
```

où

Argument	nom du paramètre transmis ou/et renvoyé
IN	mode de transmission : ENTREE
OUT	mode de transmission : SORTIE
IN OUT	mode de transmission : ENTREE - SORTIE
Type	Type du paramètre (%TYPE, NUMBER, VARCHAR2, CURSOR, ROWTYPE...)

Exemple

Création d'une procédure qui compte le nombre de services pour un numéro de continent donné.

```
CREATE OR REPLACE PROCEDURE p_service
                        (p_continent_no IN e_service.continent_no%TYPE)
IS
v_no NUMBER ;
BEGIN
SELECT COUNT(no)
INTO v_no
FROM e_service
WHERE continent_no =p_continent_no ;
DBMS_OUTPUT.PUT_LINE('Nombre de services= '||v_no) ;
END ;
/
```

Recherche d' une procédure dans la base

SELECT object_name
FROM user_objects
WHERE object_type = 'PROCEDURE' ;

Recherche du code source d' une procédure dans la base

SELECT text
FROM user_source
WHERE name ='NOM_PROCEDURE'
ORDER BY line;

Exécution

✍ Il faut avoir le privilège objet : EXECUTE

- Exécution sous SQL*PLUS :

EXECUTE [nom_propriétaire.]nom_procedure {(liste arguments)} ;

- Exécution dans le corps d'un programme PL/SQL

BEGIN
[nom_propriétaire.]nom_procedure {(liste arguments)} ;
END ;

Suppression

DROP PROCEDURE [nom_user.]nom_procédure ;

11.3.2.Les fonctions

Syntaxe de création / modification

CREATE [OR REPLACE] FUNCTION [nom_user.]nom_function
(arguments IN type [,argument IN type_arg,...]) RETURN type_val
{IS | AS }

[Variable_locale type_variable_locale ;]

BEGIN
{ contenu du bloc PL }
RETURN variable_a_retourner ;
END [nom_function] ;
END ;

où

ARGUMENT	Nom du paramètre transmis ou/et renvoyé
IN	Mode de transmission : ENTREE
TYPE	Type du paramètre (%TYPE, NUMBER, CURSOR, ROWTYPE ...)

Exemple

Création d'une fonction qui compte le nombre d'employés qui travaillent dans un service donné.

```
CREATE OR REPLACE FUNCTION f_service (p_no IN
e_emp.service_no%TYPE) RETURN NUMBER
IS
v_no NUMBER(3) ;
BEGIN
SELECT COUNT(no)
INTO v_no
FROM e_emp
WHERE service_no =p_no ;
RETURN(v_no) ;
END ;
/
```

Recherche d' une fonction dans la base

SELECT object_name
FROM user_objects
WHERE object_type = 'FUNCTION' ;

Recherche du code source d' une fonction dans la base

SELECT text
FROM user_source
WHERE name ='NOM_FONCTION'
ORDER BY line;

Exécution

✍ Il faut avoir le privilège objet : EXECUTE

- Exécution sous SQL*PLUS :

Variable_externe := [nom_propriétaire.]nom_foncfion{(liste arguments)} ;

Ou

SELECT [nom_propriétaire.]nom_foncfion{(liste arguments)}
FROM SYS.DUAL ;

- Exécution dans le corps d'un programme PL/SQL

BEGIN
:Variable :=[nom_propriétaire.]nom_fonction {(liste arguments)} ;
END ;

Suppression

DROP FUNCTION [nom_user.]nom_function ;

11.4. Les packages

11.4.1.Description

Un package est l'encapsulation d'objets dans une même unité logique de traitement :

- procédures
- fonctions
- exceptions
- variables, curseurs, constantes
- types de variable

Avantages du package

- A l'appel d'un des objets, le package entier est chargé en mémoire (source et compilé) et disponible pour tous les utilisateurs : limitation des E/S.
- Il est possible d'accorder à des utilisateurs le droit d'exécution d'une procédure ou d'une fonction du package sans qu'ils puissent accéder au code source.
- Les variables, types de variable et curseurs déclarés dans la partie spécification se comportent comme des données globales.
- Le package permet de créer des procédures surchargées.

Un package comprend deux parties :

- La partie spécification
- La partie body

Spécification du package

La partie spécification contient la déclaration des objets auxquels peuvent accéder les utilisateurs, directement sous SQL*PLUS ou à partir d'une application, lorsque ces utilisateurs ont le droit EXECUTE sur le package.

On appelle également cette partie : **partie publique.**

Body (corps du package)

Le corps du package ou body, contient la définition de tous les objets cités dans la partie spécification (procédures, fonctions, curseurs, paramètres…) et de tous les objets qui ne sont appelables qu'à l'intérieur du corps du package.

On appelle également cette partie : **partie privée.**

Cette partie peut également inclure un bloc d'initialisation qui est exécuté lors du premier appel à un élément du package.

☞ On peut déclarer la spécification d 'un package sans déclarer son corps. Les procédures qui appellent les procédures / fonctions déclarées dans ce package peuvent alors être compilées (mais pas exécutées)

Spécification

```
CREATE [OR REPLACE] PACKAGE [nom_user.]nom_package
{ IS | AS }
    nom_exception EXCEPTION;
    PRAGMA EXCEPTION_INIT (nom_exception,-sqlcode);
    TYPE nom_type IS RECORD OF … :
    TYPE  nom_type IS TABLE OF ….;
    Variable type_variable ;
    CURSOR  nom_cursor is SELECT … ;
    CURSOR  nom_curseur2(p_no type_p_no)  RETURN [type_retour];
    TYPE  type_curseur IS REF CURSOR RETURN [type_retour];
    FUNCTION nom_fonction ( arguments IN type [,argument IN type,... ] )
        RETURN type ;
    PROCEDURE nom_procédure ( arguments IN type [,argument IN type, ... ] ) ;
END [nom_package ] ;
/
```

Body

```
CREATE [OR REPLACE] PACKAGE BODY [nom_user.]nom_package
{ IS | AS }
    CURSOR nom_curseur(p_no  type_p_no) RETURN [type_retour]
    IS SELECT…. ;
    FUNCTION nom_function ( arguments IN type  [, argument IN type, .. ])
                    RETURN type_variable IS
     BEGIN
        { Traitement bloc PL }
     RETURN valeur ;
     END [nom_fonction ] ;

     PROCEDURE nom_procédure( arguments IN type[,argument IN type,..])
     IS
     Variable type_variable ;
     BEGIN
      { Traitement bloc PL }
     END [ nom_procédure ] ;

/* Bloc d'initialisation facultatif*/
[BEGIN]
END [nom_package] ;
/
```

Recherche d'un package dans la base

```
SELECT object_name
FROM user_objects
WHERE object_type = 'PACKAGE' ;
```

Recherche du code source d'un package dans la base

```
SELECT text
FROM user_source
WHERE name ='NOM_PACKAGE'
ORDER BY line;
```

Appel à un élément d'un package

Il est possible de faire référence à tous les éléments du package déclarés dans la partie spécification.
L'appel à un élément du package se fait en préfixant son nom par le nom du package.

Exemple en mode interactif, sous SQL*PLUS

Pour une procédure :

```
EXECUTE nom_package.nom_procedure {(liste arguments)} ;
```

Pour une fonction :

```
Variable := nom_package.nom_fonction {(liste arguments)} ;
```

Ou

```
SELECT nom_package.nom_fonction {(liste arguments)}
FROM SYS.DUAL ;
```

Pour une variable :

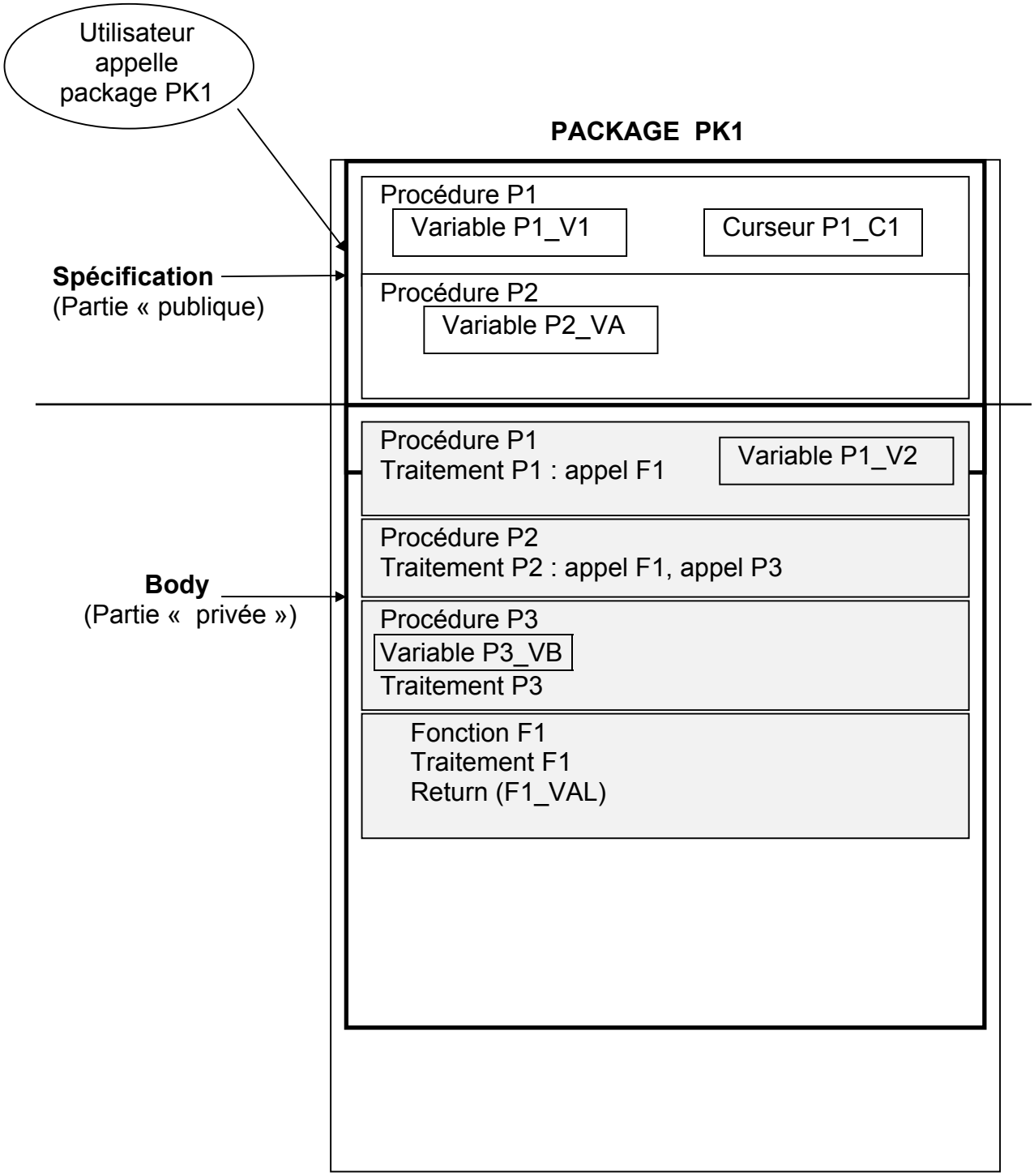
```
nom_package.nom_variable := valeur ;
```

✍ Il faut avoir le privilège objet : EXECUTE

Suppression

```
DROP PACKAGE nom_package ;
```

11.4.2.Description schématique



L'utilisateur peut exécuter P1 (ou P2), mais seule P1 (ou P2) exécutera F1 et /ou P3

Par contre un élément privé peut faire référence à un élément public.

```
CREATE OR REPLACE PACKAGE pk_emp AS
PROCEDURE p_emp_sup (p_num IN e_emp.no%TYPE);
FUNCTION  f_salaire ( p_num IN e_emp.no%TYPE) RETURN NUMBER;
PROCEDURE p_maj_service (p_num IN e_service.no%TYPE) ;
err_emp   EXCEPTION ;
END pk_emp;
/

CREATE OR REPLACE PACKAGE BODY pk_emp AS

PROCEDURE p_emp_sup (p_num IN e_emp.no%TYPE) IS
BEGIN
traitements..... ;
END p_emp_sup;

FUNCTION  f_salaire ( p_num IN e_emp.no%TYPE) RETURN NUMBER IS
BEGIN
traitements.... ;
END f_salaire ;

PROCEDURE p_maj_service (p_num IN e_service.no%TYPE)
IS
BEGIN
traitements... ;
END p_maj_service ;

END pk_emp ;
/
```

11.4.3.Validité des données

Déclaration associées aux objets du package (body)

Curseur, variable ou constante : lorsque ces éléments sont déclarés dans une procédure / fonction du package, ils sont créés dès leur initialisation dans la procédure / fonction.

Ils seront supprimés à la fin de l'exécution de la procédure / fonction.

Déclaration au niveau de la spécification du package ou non liés à un objet

Curseur, variable ou constante : lorsque ces éléments sont déclarés dans la partie spécification ou s'ils ne sont pas associés à une procédure / fonction, ils sont créés à leur initialisation et restent valides pendant toute la session.

S'il y a plusieurs sessions en cours, chaque session à sa propre image des curseurs et variables.

11.5. Gestion des packages / procédures / fonctions

Privilèges requis

L'utilisateur qui crée un package / procédure / fonction dans son propre schéma, doit avoir le privilège :

CREATE PROCEDURE

S'il doit créer un package / procédure / fonction dans n'importe quel schéma, il doit avoir le privilège :

CREATE ANY PROCEDURE

Faciliter l'utilisation

On peut faciliter l'accès au package / procédure / fonction en leur attribuant un synonyme public.

Syntaxe

```
CREATE PUBLIC SYNONYM nom_synonym
        FOR {nom_schéma.[nom_package|nom_procédure|nom_fonction]} ;
```

11.6. Les triggers stockés

11.6.1.Définition

Un trigger stocké est un traitement procédural lié à une table et une seule, et donc répertorié dans le dictionnaire de données.

- Il se déclenche automatiquement lors d'un événement intervenu sur la table dont il dépend : insertion, suppression ou mise à jour.
- Il reste valide tant que la table existe.
- Il peut être actif ou inactif

Par exemple, on peut créer un trigger sur la table e_emp qui vérifiera lors de chaque création ou mise à jour que la date d'entrée de l'employé n'est pas NULL.

Le trigger stocké se déclenchera quel que soit l'origine de la mise à jour : SQL*PLUS, application, programme en langage PRO*, ...

11.6.2.Caractéristiques

3 cas de mise à jour :

- INSERT
- UPDATE
- DELETE

2 types de trigger :

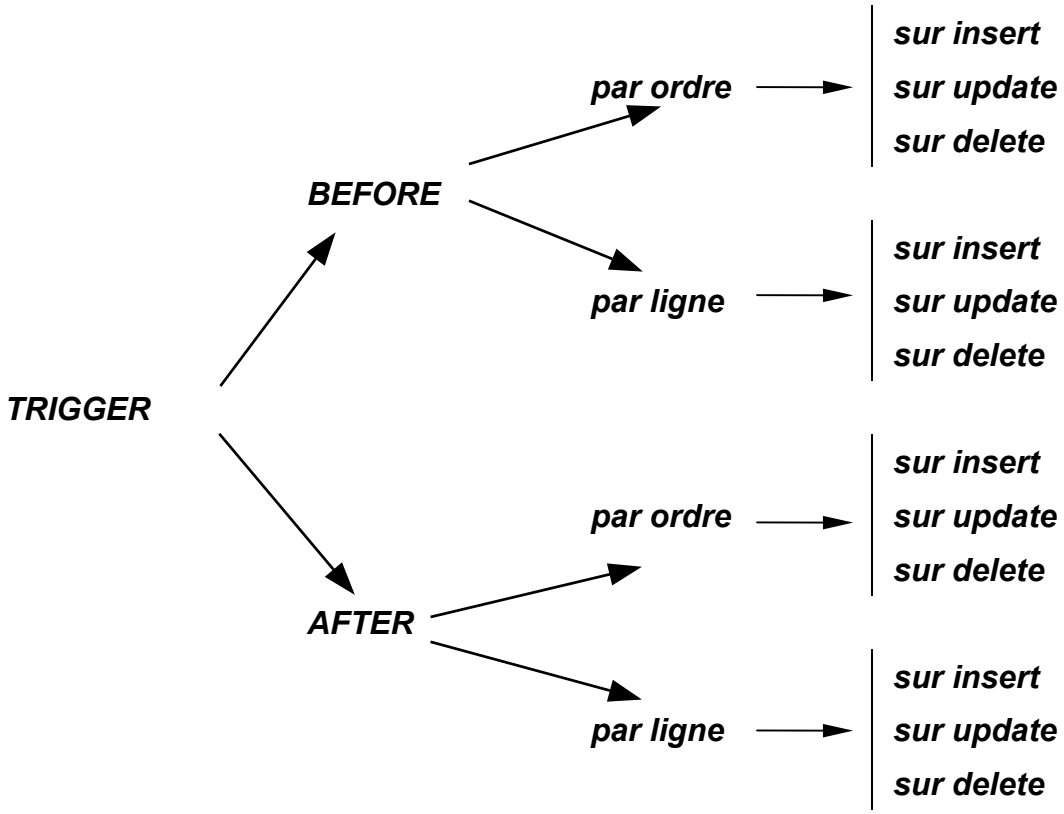
- Déclenchement sur chaque ligne mise à jour
- Déclenchement une seule fois pour la mise à jour

2 séquencements :

- Avant la mise à jour
- Après la mise à jour

= 12 triggers possibles sur une table

Récapitulatif des triggers possibles



Syntaxe

Création / Modification du trigger

```
CREATE [ OR REPLACE ] TRIGGER [nom_user.]nom_trigger
{ BEFORE | AFTER } { INSERT | UPDATE [ OF col1…] | DELETE }
[ OR { INSERT | UPDATE | DELETE } ]
[ OR { INSERT | UPDATE | DELETE } ]
ON nom_table
[ FOR EACH ROW [ WHEN (prédicat) ] ]
DECLARE
--Déclaration de variables locales au trigger ;

BEGIN
{ contenu du bloc PL }
END ;
/
```

On peut également, si le trigger est activé par plusieurs événements, utiliser des prédicats booléens prédéfinis tels que INSERTING, DELETING ou UPDATING :

```
CREATE OR REPLACE TRIGGER t_emp
BEFORE INSERT OR UPDATE OR DELETE ON e_emp
FOR EACH ROW
BEGIN
 IF INSERTING THEN ...
Traitement spécifique à l'insertion ;
 END IF ;
 IF UPDATING THEN
    Traitement spécifique à la modification ;
 END IF;
IF DELETING THEN
Traitement spécifique à la suppression ;
END IF ;
Traitements communs aux trois événements ;
END ;
```

Désactivation d'un trigger

ALTER TRIGGER [nom_user.]nom_trigger DISABLE;

Réactivation d'un trigger

ALTER TRIGGER [nom_user.]nom_trigger ENABLE;

Désactivation de tous les triggers sur une table

ALTER TABLE Nom_table DISABLE ALL TRIGGERS;

Réactivation de tous les triggers sur une table

ALTER TABLE nom_table ENABLE ALL TRIGGERS;

Suppression du trigger

DROP TRIGGER [nom_user.]nom_trigger;

Restrictions d'utilisation

1-Jamais de COMMIT dans un trigger.
2-Pas de consultation ni de mise à jour dans le trigger de la table sur laquelle se déclenche le trigger (*table mutating cf. § Triggers en cascade*).

11.6.3.Utilisation des variables « OLD. » et « NEW.»

Elles sont utilisables uniquement dans les triggers FOR EACH ROW. Dans le bloc PL/SQL, on peut faire référence aux colonnes en les préfixant avec :NEW.nom_colonne ou :OLD.nom_colonne avec les restrictions suivantes :

:NEW uniquement dans les triggers INSERT ou UPDATE

(On ne peut pas modifier la valeur de :NEW.nom_colonne dans les triggers AFTER).

:OLD uniquement dans les triggers UPDATE ou DELETE

(On ne peut pas modifier la valeur de :OLD.nom_colonne).

Dans la clause WHEN, les colonnes sont préfixées avec NEW ou OLD sans les ' **:**'

	OLD	NEW
INSERT	NULL	Valeur créée
DELETE	Valeur avant suppression	NULL
UPDATE	Valeur avant modification	Valeur après modification

Exemples :

1) Avant chaque suppression de ligne dans la table e_emp :

```
CREATE OR REPLACE TRIGGER t_emp_del
BEFORE DELETE ON e_emp
FOR EACH ROW
BEGIN
  INSERT into tab_mvts VALUES
  (:OLD.no, :OLD.nom, SYSDATE) ;
END ;
/
```

2) Lors de chaque insertion dans la table e_service :

```
 CREATE OR REPLACE TRIGGER t_serv_ins
BEFORE INSERT ON e_service
FOR EACH ROW
BEGIN
  :NEW.no :=... ;
END ;
 /
```

Recherche d'un trigger dans la base

```
SELECT  trigger_body
FROM  user_triggers
WHERE trigger_name ='NOM_TRIGGER' ;
```

11.6.4.Cas du trigger INSTEAD OF

Depuis la **version 8 d'Oracle**, on peut créer une vue à partir d'une requête qui comporte une jointure, mais les informations mises à jour ne peuvent être que des colonnes de la table de niveau le plus bas.

Le trigger INSTEAD OF permet l'insertion, la modification et la suppression des enregistrements de plusieurs tables à travers une vue multi-table. Il se déclare uniquement sur des vues.

Exemple

Création d'une vue :

```
CREATE OR REPLACE VIEW v_es (no_emp,nom_emp,no_serv,nom_serv)
  AS SELECT e.no,e.nom,s.no,s.nom
      FROM e_emp e,e_service s
       WHERE e.service_no =s.no;
```

Création du trigger INSTEAD OF :

```
CREATE OR REPLACE TRIGGER ins_v_es
      INSTEAD OF INSERT ON v_es
BEGIN
   INSERT INTO e_service(no,nom)
   VALUES(:new.no_serv,:new.nom_serv);

INSERT INTO e_emp(no,nom,service_no)
VALUES(:new.no_emp,:new.nom_emp, :new.no_serv);
END;
/
```

Insertion d'une ligne

```
INSERT INTO v_es
VALUES(36,'DUPONT',67,'Informatique');
```

11.6.5.Triggers en cascade

L'exécution d'un trigger peut entraîner l'exécution d'un autre trigger sur la table en cours de modification par son exécution.

Dans ce cas, quelques précautions s'imposent pour éviter l'interruption de la transaction en cours et le message « table mutating » ou « objet mutant ».

Précautions

- Aucun ordre ne doit consulter ou modifier une table déjà utilisée en modification par un autre utilisateur.
  Pour éviter ce type de collision, créez une fonction qui testera l'état de la table (récupération du message d'erreur), la gestion dépendra du contexte : abandon ou attente et affichage d'un message d'erreur.

- Aucun ordre ne doit modifier une colonne déclarée en PRIMARY, UNIQUE ou FOREIGN KEY.

11.7. Les dépendances

Il y a dépendance des objets (procédures, fonctions, packages, vues) lorsqu'ils font référence à des objets de la base tels qu'une vue (qui fait elle-même référence à une ou plusieurs tables), une table, une autre procédure, etc…

Si l'objet en référence est modifié, il est nécessaire de recompiler les objets **dépendants.**

Le comportement d'un objet dépendant varie selon son type :

- procédure et / ou fonction
- package

11.7.1. Dépendance des procédures / fonctions

Deux types de dépendance

- **Dépendance directe**

Le traitement fait explicitement référence à l'objet modifié qui peut être

⇒ une table
⇒ une vue
⇒ une séquence
⇒ un synonyme
⇒ un autre traitement (procédure, fonction)

Exemple

Une table T1 sur laquelle travaille une procédure P1 : il y dépendance directe.

- **Dépendance indirecte**

  Il y a dépendance indirecte lorsque le traitement fait indirectement référence à un autre objet.

  ⇒ une vue liée à une ou plusieurs tables

  ⇒ une vue liée à une autre vue

  ⇒ un objet au travers d'un synonyme

  ⇒ ..

  Exemple

  Une table T1 sur laquelle porte une vue V1. La procédure P1 travaille à partir de V1 : il y a dépendance indirecte vis à vis de T1.

Deux cas peuvent se présenter :

- **Dépendance locale**
  Les procédures et fonctions sont sur la même base que les objets auxquels elles font référence.

- **Dépendance distante**
  Les procédures et fonctions sont sur une base différente de celle des objets auxquels elles font référence.

Informations sur les dépendances

- Connaître les dépendances directes : consulter les tables

  USER | ALL | DBA_DEPENDENCIES

- Connaître les dépendances indirectes : utiliser la procédure

  $ORACLE_HOME/RDBMS/ADMIN/DEPTREE_FILL liée aux vues DEPTREE et IDEPTREE.

11.8. Impacts et gestion des dépendances

Chaque fois qu'un objet référencé par une procédure / fonction est modifié, le statut du traitement dépendant passe à INVALIDE : Il est alors nécessaire de le recompiler.

- Consulter le statut d'un objet : USER | ALL | DBA_OBJECTS

✍ Une vue suit les mêmes règles qu'une procédure / fonction. Elle est recompilée si sa table source est modifiée.

Objets sur la même base locale :	ORACLE vérifie le statut des objets dépendants et les recompile automatiquement.
Objets sur des bases différentes :	ORACLE n'intervient pas, la recompilation doit être manuelle.

✍ Cas d'une dépendance locale : même si ORACLE recompile automatiquement les procédures INVALIDES, il est conseillé de la faire manuellement.

Gains de performance : évite les contentions sur un même objet, chargement plus rapide du traitement en mémoire.

11.8.1.Procédure / fonction

Syntaxe

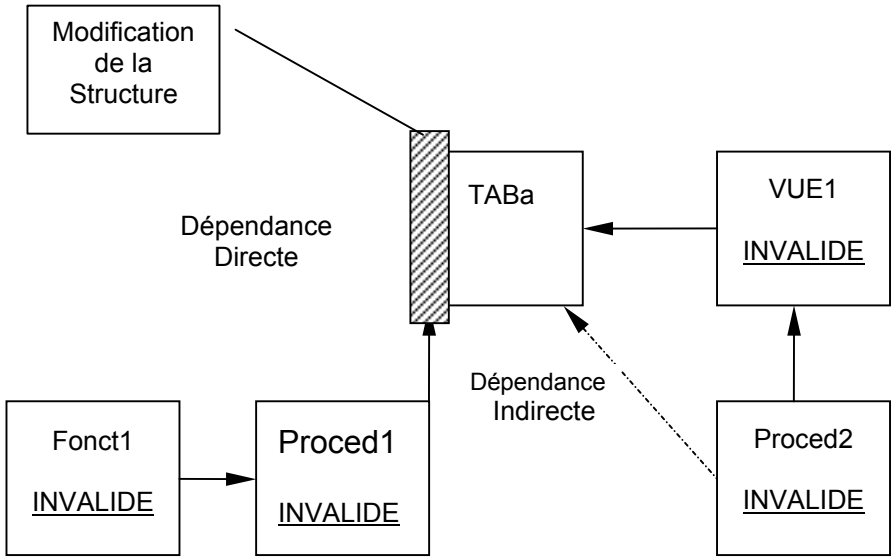
ALTER {PROCEDURE | FUNCTION | VIEW } nom_objet COMPILE ;

Exemple

On a ajouté la colonne DT_MODIF à la table des auteurs AUT.
Proced1 et vue1 dépendent directement de la table AUT.
Fonct1 et proced2 dépendent indirectement de la table AUT.

```
ALTER PROCEDURE proced1 COMPILE ;
ALTER VIEW      vue1    COMPILE ;
ALTER PROCEDURE proced2 COMPILE ;
ALTER FUNCTION  fonct1  COMPILE ;
```

Schéma dépendance procédure / fonction



11.8.2.Package

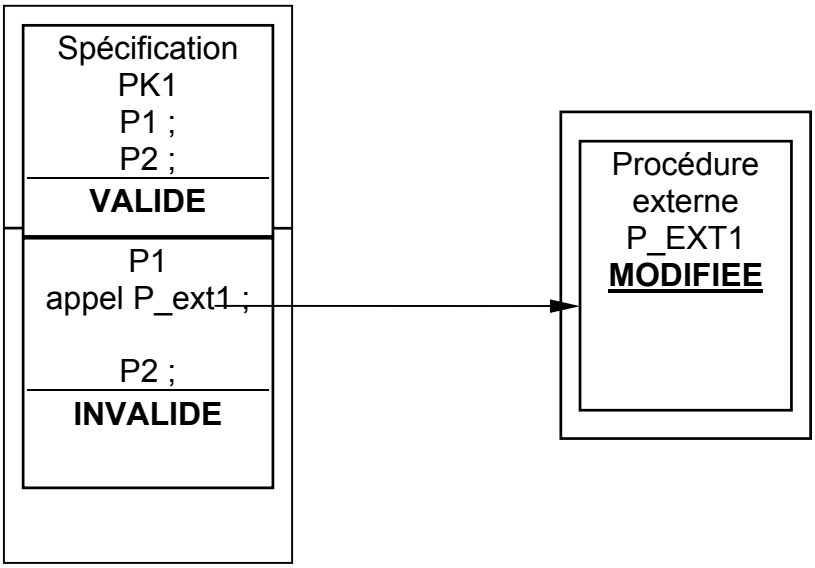
La gestion des dépendances pour les packages sont plus simples

1) On modifie une procédure externe au package : il faut seulement recompiler le corps du package.

Syntaxe

ALTER PACKAGE BODY nom_package COMPILE ;

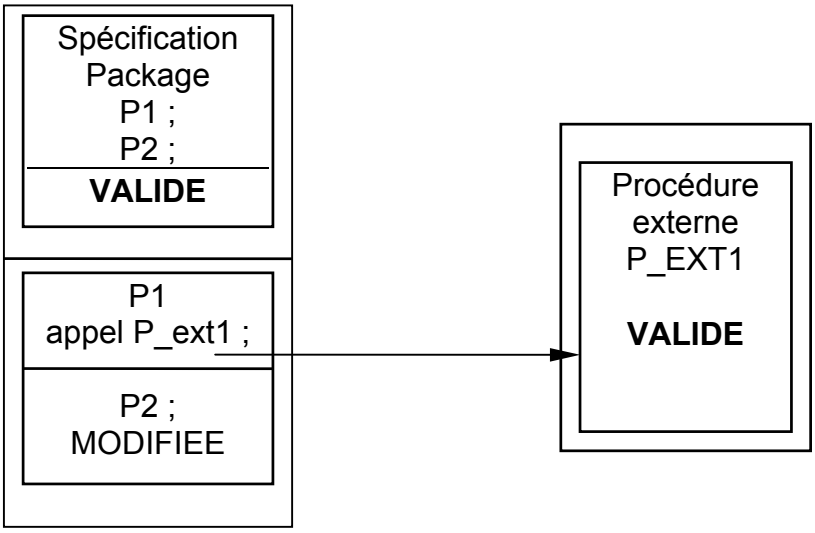
Exemple




```
CREATE OR REPLACE PROCEDURE P_EXT1 ...IS
BEGIN
.....
END ;
ALTER PACKAGE BODY pk1 COMPILE ;
```

2) On modifie un élément dans le corps du package, sans rien modifier dans la partie spécification, il n'y a pas besoin de recompiler la procédure externe.

Exemple




```
CREATE OR REPLACE PROCEDURE P2 IS ...
BEGIN
.....
END ;
/
```

12. LES PACKAGES INTEGRES

12.1. Le package DBMS_OUTPUT

Le package DBMS_OUTPUT permet de stocker de l'information dans un tampon avec les modules PUT et PUT_LINE.

On peut récupérer ces informations en appelant GET et GET_LINE.

Les informations stockées dans le tampon de sortie peuvent permettre de tracer les programmes PL/SQL.

Les procédures de DBMS_OUTPUT :

- GET_LINE (ligne OUT VARCHAR2, statut OUT INTEGER) ;

  Extrait une ligne du tampon de sortie.

- GET_LINES (lignes OUT VARCHAR2, n IN OUT INTEGER);

  Extrait, à partir du tampon de sortie, un tableau de n lignes.

- NEW_LINE;

  Place un marqueur de fin de ligne dans le tampon de sortie.

- PUT (variable | constante IN {VARCHAR2|NUMBER|DATE}) ;

  Place la valeur spécifiée dans le tampon de sortie.

- PUT_LINE (variable | constante IN {VARCHAR2|NUMBER|DATE});

  Combinaison de PUT et NEW_LINE.

- ENABLE (taille tampon IN INTEGER DEFAULT 2000) ;

  Permet de mettre en route le mode trace dans une procédure ou une fonction.

- DISABLE

  Permet de désactiver le mode trace dans une procédure ou une fonction.

12.2. Le package UTL_FILE

Le package UTL_FILE permet aux programmes PL/SQL d'accéder à la fois en lecture et en écriture à des fichiers systèmes.

On peut appeler UTL_FILE à l'intérieur de programmes stockés sur le serveur ou à partir de modules résidents sur la partie cliente de l'application, comme ceux développés avec Oracle FORMS.

Les modules de UTL_FILE

- Fonction FOPEN(location IN VARCHAR2,
  nom_fichier IN VARCHAR2,
  mode_ouverture IN VARCHAR2)
  RETURN UTL_FILE.FILE_TYPE ;

Cette fonction ouvre un fichier et renvoie un pointeur de type UTL_FILE.FILE_TYPE sur le fichier spécifié.

- ✓ -location est l'emplacement du fichier sur le poste serveur,
- ✓ -nom_fichier est le nom du fichier avec son extension,
- ✓ -mode_ouverture spécifie ouverture en lecture 'R',
  en écriture-lecture en mode remplacement 'W'
  en écriture-lecture en mode ajout 'A'.

Il faut avoir le droit d'ouvrir un fichier dans le répertoire spécifié. Pour cela, il faut accéder au paramètre utl_file_dir dans le fichier init___.ora.

La fonction peut générer les exceptions

INVALID_PATH ,INVALID_MODE ,INVALID_OPERATION.

- Procédure GET_LINE(pointeur_fichier IN UTL_FILE.FILE_TYPE,
  ligne OUT VARCHAR2) ;

  Cette procédure lit une ligne du fichier spécifié, s'il est ouvert, dans la variable ligne. Lorsqu' elle atteint la fin du fichier l'exception NO_DATA_FOUND est déclenchée.

- Procédure PUT_LINE (pointeur_fichier IN UTL_FILE.FILE_TYPE,
  ligne OUT VARCHAR2) ;

  Cette procédure insère des données dans un fichier et ajoute automatiquement une marque de fin de ligne. Lorsqu'elle atteint la fin du fichier, l'exception NO_DATA_FOUND est déclenchée.

- Procédure PUT(pointeur_fichier UTL_FILE.FILE_TYPE,
  item IN {VARCHAR2|NUMBER|DATE})

  Cette procédure permet d'ajouter des données dans le fichier spécifié.

- Procédure NEW_LINE(pointeur_fichier UTL_FILE.FILE_TYPE);

  Cette procédure permet d'ajouter une marque de fin de ligne à la fin de la ligne courante.

- Procédure PUTF(pointeur_fichier UTL_FILE.FILE_TYPE,
  format IN VARCHAR2,
  item1 IN VARCHAR2,
  [item2 IN VARCHAR2,.....]);

  Cette procédure insère des données dans un fichier suivant un format.

- Procédure FCLOSE (pointeur_fichier IN UTL_FILE.FILE_TYPE) ;

  Cette procédure permet de fermer un fichier.

- Procédure FCLOSE_ALL ;

  Cette procédure permet de fermer tous les fichiers ouverts.

- Fonction IS_OPEN(pointeur_fichier IN UTL_FILE.FILE_TYPE)
  RETURN BOOLEAN ;

  Cette fonction renvoie TRUE si pointeur_fichier pointe sur un fichier ouvert.

12.3. le package DBMS_SQL

Le package DBMS_SQL permet d'accéder dynamiquement au SQL à partir du PL/SQL .

Les requêtes peuvent être construites sous forme de chaînes de caractères au moment de l'exécution puis passées au moteur SQL.

Ce package offre notamment la possibilité d'exécuter des commandes DDL dans le corps d'un programme.

- Fonction OPEN_CURSOR RETURN INTEGER ;

Cette fonction ouvre un curseur et renvoie un INTEGER.

- Procédure PARSE (pointeur IN INTEGER, requête_sql IN VARCHAR2, DBMS.NATIVE) ;

Cette procédure analyse la chaîne 'requête_sql' suivant la version sous laquelle l'utilisateur est connecté.

- Fonction EXECUTE (pointeur IN INTEGER) RETURN INTEGER ;

Cette fonction exécute l'ordre associé au curseur et renvoie le nombre de lignes traitées dans le cas d'un INSERT, DELETE ou UPDATE.

- Procédure CLOSE_CURSOR (pointeur IN OUT INTEGER) ;

Cette procédure ferme le curseur spécifié, met l'identifiant du curseur à NULL et libère la mémoire allouée au curseur.

Exemple1 : Création d'une procédure qui supprime une table

```
CREATE OR REPLACE PROCEDURE dr (p_table IN VARCHAR2) IS

poignee INTEGER;
exec INTEGER;

BEGIN
poignee :=DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE(poignee,'drop table '||p_table,DBMS_SQL.NATIVE);
exec:=DBMS_SQL.EXECUTE(poignee);
DBMS_SQL.CLOSE_CURSOR(poignee);
END;
/

SQL>EXECUTE dr('e_resultat') ;
```

L 'exécution de la procédure DR entraîne la suppression de la table p_table.

Exemple2 : Procédure mettant à jour une colonne d'une table pour une ligne donnée

```
CREATE OR REPLACE PROCEDURE up(p_salaire IN NUMBER, p_no IN
NUMBER,p_table IN VARCHAR2) IS

pointeur INTEGER;
exec INTEGER;

BEGIN
pointeur :=DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE(pointeur,'update '||p_table||' set salaire ='||p_salaire||' where
no ='||p_no,DBMS_SQL.V7);
exec:=DBMS_SQL.EXECUTE(pointeur);
DBMS_SQL.CLOSE_CURSOR(pointeur);
END;
/

SQL>EXECUTE up(1500,10,'e_emp') ;
```

Cette commande met à jour le salaire de l'employé no 10 de la table e_emp.

- Procédure BIND_VARIABLE(pointeur IN INTEGER, nom_variable_substitution IN VARCHAR2, valeur_variable_substitution IN {NUMBER|VARCHAR2|DATE}) ;

Lorsqu'on envoie une requête SQL comportant une variable de substitution ,

```
UPDATE e_emp
SET salaire = 1300
WHERE no = :NUMERO ;
```

L'analyse de la syntaxe est effectuée avant que l'affectation d'une valeur à :NUMERO ne soit réalisée.

C'est la procédure BIND_VARIABLE qui permet de réaliser cette affectation après le parsing et avant l'exécution du curseur.

Exemple3

```
CREATE OR REPLACE PROCEDURE up2 (p_no  IN NUMBER) IS

pointeur INTEGER;
exec INTEGER;

BEGIN
pointeur := DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE(pointeur,'update e_emp set salaire =1300 where no
=:glob',DBMS_SQL.NATIVE);
DBMS_SQL.BIND_VARIABLE(pointeur,'glob',p_no);
exec:=DBMS_SQL.EXECUTE(pointeur);
DBMS_SQL.CLOSE_CURSOR(pointeur);
END;
/

SQL> EXECUTE up(25);
```

Cette commande met à jour le salaire de l'employé no 25 de la table e_emp.

- Procédure DEFINE_COLUMN(pointeur IN INTEGER, position IN INTEGER, variable IN {DATE|NUMBER|VARCHAR2}) ;

Dans le cas où l'ordre SQL est un SELECT, il faut associer les colonnes ou expressions du SELECT à des variables locales avant l'exécution du curseur. 'position' est la position de la colonne dans l'ordre SELECT.

- Fonction FETCH_ROWS(pointeur IN INTEGER) RETURN INTEGER ;

Cette fonction intervient dans le cas d'un SELECT après l'exécution du curseur et correspond à la commande FETCH pour les curseurs PL/SQL classiques. Elle renvoie 0 lorsqu'il n'y a plus d'enregistrements à ramener.

- Procédure COLUMN_VALUE(pointeur IN INTEGER, position IN INTEGER, variable IN {DATE|NUMBER|VARCHAR2}) ;

Cette procédure passe une valeur du curseur dans une variable pour chaque ligne retournée.

13. DEBOGAGE SOUS SQL*PLUS

Pour afficher le contenu de variables à l'écran ainsi que des messages il faut utiliser les procédures du package DBMS_OUTPUT. La procédure PUT_LINE permet de stocker des données dans un tampon de sortie . Pour visualiser ces données, il faut positionner sur ON la variable d'environnement SERVEROUTPUT :

SQL>SET SERVEROUTPUT ON

Exemple

```
SQL> SET SERVEROUTPUT ON
SQL>DECLARE
   Nom_emp VARCHAR2(25) ;
   BEGIN
   SELECT nom
   INTO Nom_emp
   WHERE no =15 ;
   DBMS_OUTPUT.PUT_LINE(Nom_emp) ;
   END ;
   /
```

Visualisation des erreurs de compilation

SQL> SET ECHO ON

Visualisation des erreurs de compilation des objets stockés

SQL>SHOW ERRORS

INDEX

O

P

R

S

T

U

V

W