Oracle PL/SQL

par SheikYerbouti

Date de publication : Avril 2004

Dernière mise à jour : Juillet 2004

Découvrez le langage procédural d'Oracle

Introduction

Tutoriel au format PDF

Téléchargez le tutoriel au format PDF

PL/SQL est le langage procédural d'Oracle. Il est une extension du SQL qui est un langage ensembliste.

PL/SQL permet de gérer des traitements qui utilisent les instructions SQL dans un langage procédural.

Les instructions de manipulation des données, de description des données, de contrôle des transactions,

les fonctions SQL peuvent être utilisées avec la même syntaxe.

La gestion des variables et des structures de contrôle (tests, boucles) augmente la capacité

de traitement des données

La gestion des curseurs et du traitement des erreurs accroît les possibilités de traitement

Les instructions sont regroupées dans une unité appelée bloc qui ne génère qu'un accès à la base

Les blocs ou procédures PL/SQL sont compilés et exécutés par le moteur PL/SQL.

Ce moteur est intégré au moteur de la base de données et dans un certain nombre d'outils (Forms, Report).

En résumé, PL/SQL permet de construire des applications

Indication au lecteur

Cet ouvrage se situe entre le tutorial et le guide de référence

Il n'a pas pour vocation de se substituer à un ouvrage de formation à l'usage du débutant

La compréhension de cet article sous-entend des connaissances préalables en développement(en général) et en SQL(en particulier)

Versions des logiciels utilisés

L'intégralité des exemples présentés dans l'article a été effectuée avec la configuration suivante

OS : Windows 2000 5.00.2195 service pack 3

Noyau Oracle : Oracle9i Enterprise Edition Release 9.2.0.1.0 - Production

Sql*Plus : SQL*Plus: Release 9.2.0.1.0 - Production

Ce document traite des fonctionnalités PL/SQL actuellement en cours.

Dans la mesure du possible les nouveautés apparues avec la version 9i et 10g sont indiquées **(9i)** ou **(10g)**

En aucun cas, ce travail n'a été effectué entre les versions 7 et 8.

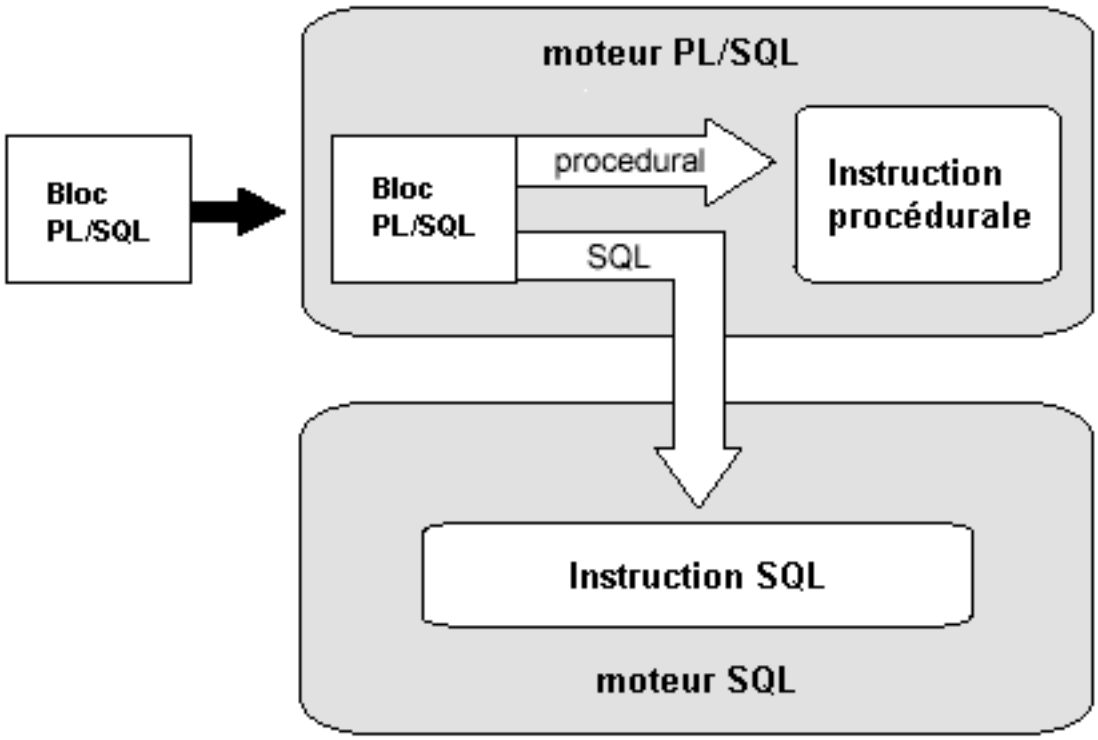
Il est donc tout à fait possible que certains exemples tirés de cet article provoquent des erreurs de compilation ou d'exécution s'ils sont testés sur une version inférieure à 9.2

1 - Le bloc PL/SQL

PL/SQL est un langage structuré en blocs, constitués d'un ensemble d'instructions.

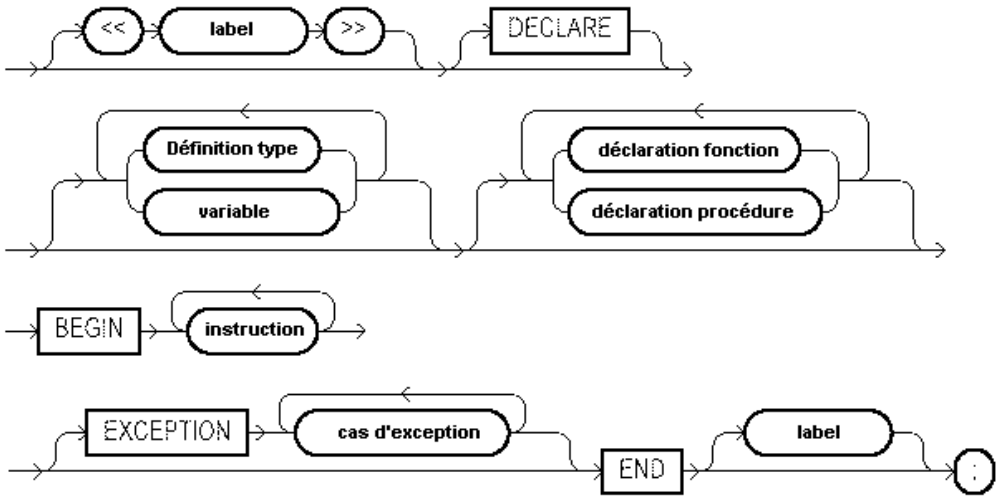
Un bloc PL/SQL peut être "externe", on dit alors qu'il est anonyme, ou alors stocké dans la base de données sous forme de procédure, fonction ou trigger.

un bloc PL/SQL est intégralement envoyé au moteur PL/SQL, qui traite chaque instruction PL/SQL et sous-traite les instructions purement SQL au moteur SQL, afin de réduire le trafic réseau.

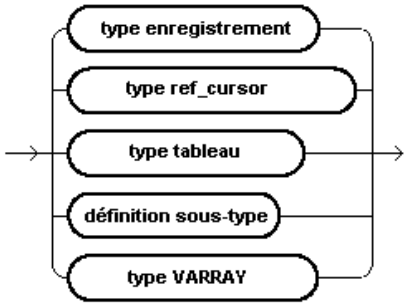


Syntaxe d'un bloc PL/SQL

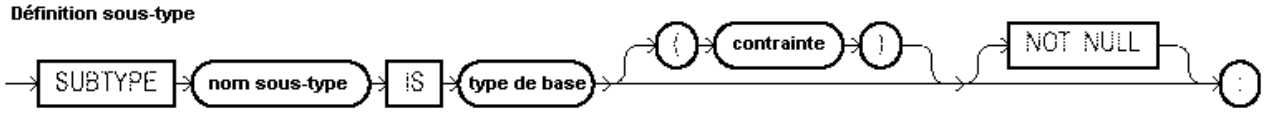
Bloc PL/SQL



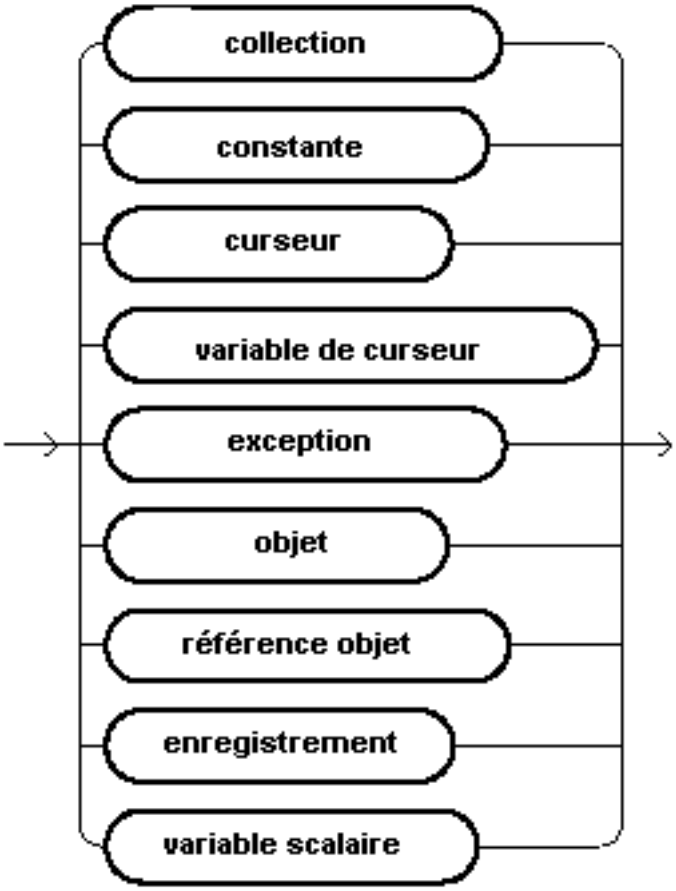
Définition type

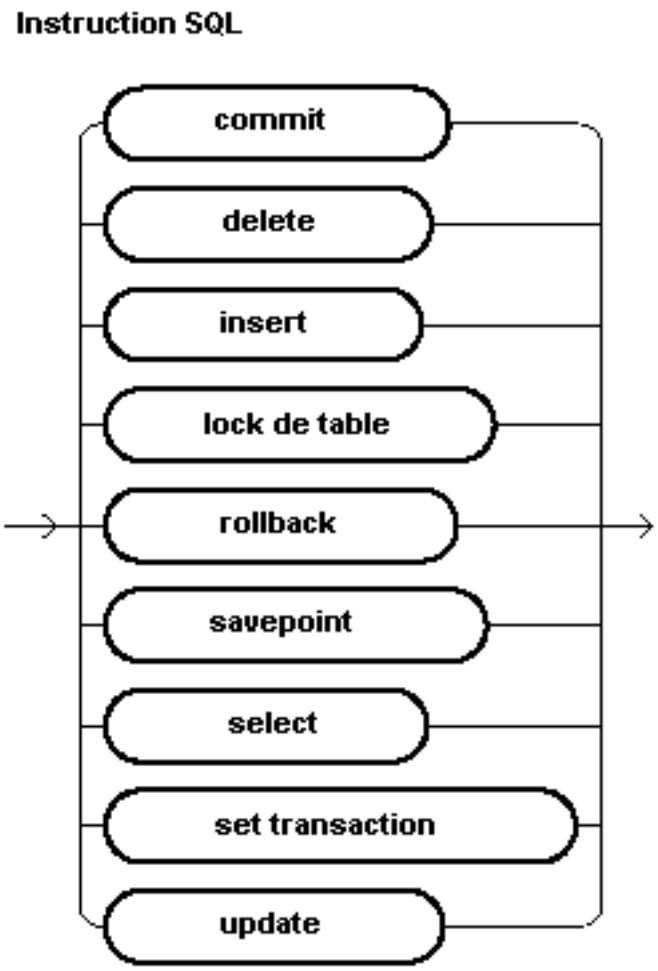


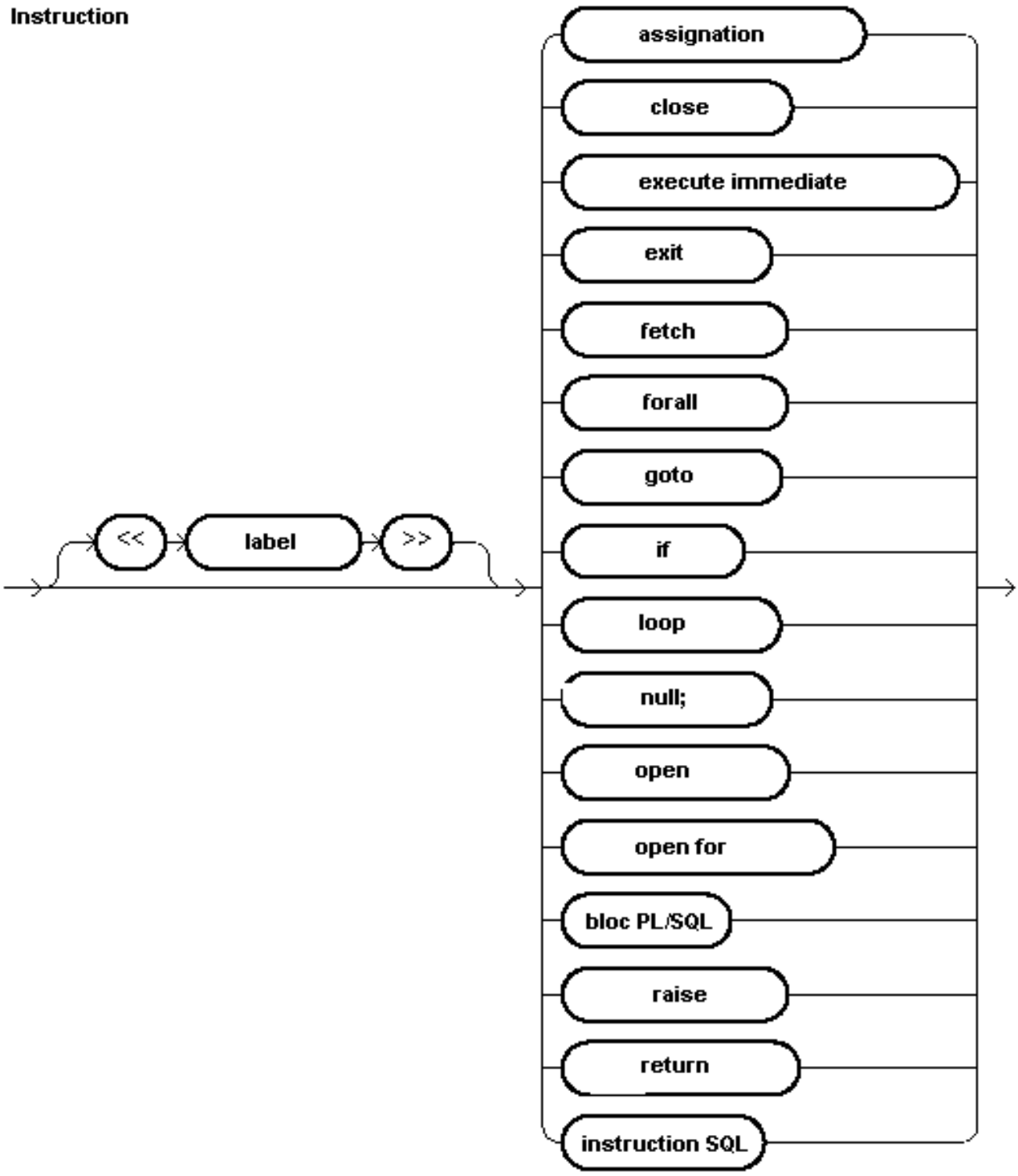
Définition sous-type



Variable







Chaque bloc PL/SQL peut être constitué de 3 sections :

- **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**

[**DECLARE**

déclarations et initialisation]

BEGIN

instructions exécutables

[**EXCEPTION**

interception des erreurs]

END;

Un bloc PL/SQL minimum peut être représenté de la façon suivante :

```
BEGIN
   Null ;
END ;
```

Le mot clé **BEGIN** détermine le début de la section des instructions exécutables

Le mot clé **END;** indique la fin de la section des instructions exécutables

Une seule instruction figure dans ce bloc : **Null;** qui ne génère aucune action

Ce bloc PL/SQL ne fait donc absolument rien !

La section déclarative (facultative) d'un bloc débute par le mot clé **DECLARE**

Elle contient toutes les déclarations des variables qui seront utilisées localement par la section exécutable, ainsi que leur éventuelle initialisation.

```
DECLARE
   LC$Chaine   VARCHAR2(15) := 'Salut Monde' ;
BEGIN
   DBMS_OUTPUT.PUT_LINE( LC$Chaine ) ;
END ;
```

Une variable LC$Chaine est déclarée de type VARCHAR2(15) et initialisée avec la valeur 'Salut Monde' ;

Dans la section exécutable, cette variable est transmise à la fonction DBMS_OUTPUT() pour être affichée à l'écran

Cette section ne peut pas contenir d'instructions exécutables. Toutefois, il est possible de définir dans cette section des procédures ou des fonctions contenant une section exécutable.

Toute variable doit avoir été déclarée avant de pouvoir être utilisée dans la section exécutable.

La section de gestion des erreurs (facultative) débute par le mot clé **EXCEPTION**

Elle contient le code exécutable mis en place pour la gestion des erreurs

Lorsqu'une erreur intervient dans l'exécution, le programme est stoppé et le code erreur est transmis à cette section

```
DECLARE
   LC$Chaine   VARCHAR2(15) := 'Hello World' ;
BEGIN
   DBMS_OUTPUT.PUT_LINE( LC$Chaine ) ;
EXCEPTION
   When OTHERS then
         Null ;
END ;
```

Les erreurs doivent être interceptées avec le mot clé **WHEN** suivi du code erreur ciblé. Ici, le code **OTHERS** qui définit toutes les erreurs non interceptées individuellement par les clauses **WHEN** précédentes.

Cette section peut elle-même contenir d'autres blocs PL/SQL

Les blocs PL/SQL peuvent être imbriqués les uns dans les autres

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

   ##..
END ;
```

1.1 - La section déclarative

Vous pouvez déclarer dans cette section tous les types, variables et constantes nécessaires à l'exécution du bloc.

Ces variables peuvent être de n'importe quel type SQL ou PL/SQL (voir le chapitre Variables, types et littéraux).

Leur initialisation, facultative, s'effectue avec l'opérateur **:=**

```
DECLARE
   LN$Nbre   NUMBER(3) := 0 ;
   LD$Date   DATE := SYSDATE ;
   LC$Nom    VARCHAR2(10) := 'PL/SQL' ;
```

Une constante est une variable dont l'initialisation est obligatoire et dont la valeur ne pourra pas être modifiée en cours d'exécution

Elle est déclarée avec le mot clé : CONSTANT qui doit précéder le type

```
DECLARE
   LN$Pi  CONSTANT NUMBER := 3. 1415926535 ;
```

PL/SQL n'est pas sensible à la casse. Pour lui les expressions suivantes sont équivalentes :

NOM_VARIABLE NUMBER ;

Nom_Variable Number ;

nom_variable number ;

1.2 - La section exécution

Délimitée par les mots clé **BEGIN** et **END;** elle contient les instructions d'exécution du bloc PL/SQL, les instructions de contrôle et d'itération,

l'appel des procédures et fonctions, l'utilisation des fonctions natives, les ordres SQL, etc.

Chaque instruction doit être suivi du terminateur d'instruction **;**

Voici la liste des instructions que cette section peut contenir

1.2.1 - Assignation

L'assignation d'une valeur à une variable peut être faite de 2 façons différentes

- En utilisant l'opérateur :=

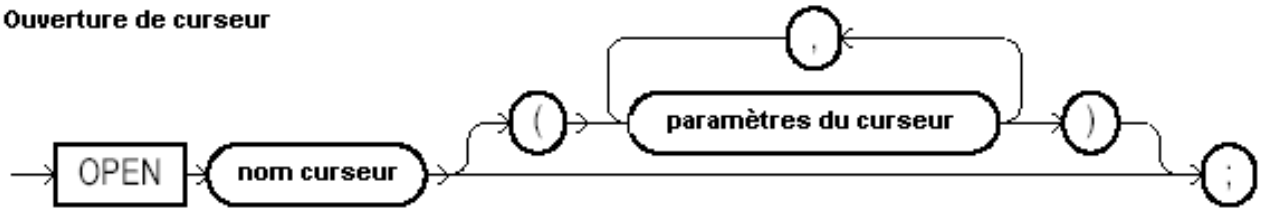
```
Ma_variable := 10 ;
Ma_chaine   := 'Chaîne de caractères' ;
```

- Par l'intermédiaire d'un ordre **SELECT # INTO** ou **FETCH # INTO**

```
Declare
   LC$Nom_emp  EMP.ENAME%Type ;

   Cursor C_EMP Is
   Select
        ename
   From
        EMP
   Where
        Empno = 1014
   ;

Begin
   Select
        ename
   Into
        LC$Nom_emp
   From
        EMP
   Where
        Empno = 1014
   ;

   Open  C_EMP ;
   Fetch C_EMP Into LC$Nom_emp ;
   Close C_EMP ;
End ;
```

1.2.2 - OPEN

Ouverture d'un curseur SQL



nom curseur représente le nom donné au curseur qui permettra de le référencer dans les instructions suivantes

paramètres du curseur représente la liste des paramètres transmis au curseur

le curseur doit avoir été préalablement défini dans la section déclarative

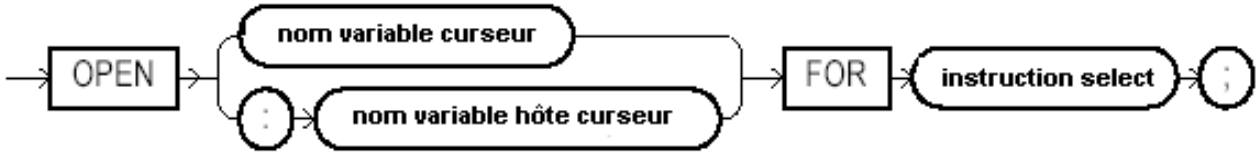
```
Declare
   LC$Nom_emp  EMP.ENAME%Type ;

   Cursor C_EMP ( LN$Numemp IN EMP.EMPNO%Type ) Is
   Select
        ename
   From
        EMP
   Where
        Empno = LN$Numemp
   ;
```

```
Begin
   Open  C_EMP ( 1024 );
   Fetch C_EMP Into LC$Nom_emp ;
   Close C_EMP ;
End ;
```

1.2.3 - OPEN FOR

Ouverture d'un curseur SQL incluant l'ordre select correspondant.

La déclaration préalable du curseur dans la section déclarative n'est pas nécessaire




```
Declare
   LC$Nom_emp  EMP.ENAME%Type ;
Begin
   Open  C_EMP For 'Select ename From EMP Where empno = 1024' ;
   Fetch C_EMP Into LC$Nom_emp ;
   Close C_EMP ;
End ;
```

1.2.4 - CLOSE

Cette instruction est utilisée pour fermer un curseur préalablement ouvert avec l'instruction OPEN

CLOSE(nom_curseur)

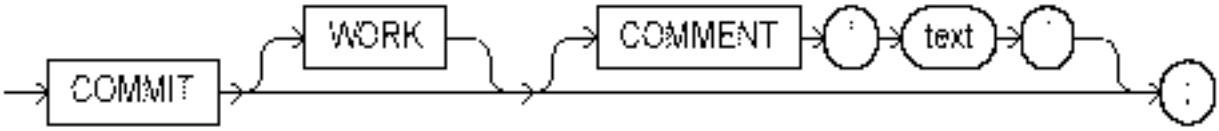
Après cette instruction, le curseur n'est plus valide et toute instruction s'y reportant génèrera une erreur

(voir exemple précédent)

1.2.5 - COMMIT

Cette instruction permet d'enregistrer en base toutes les modifications effectuées au cours de la transaction

Instruction COMMIT

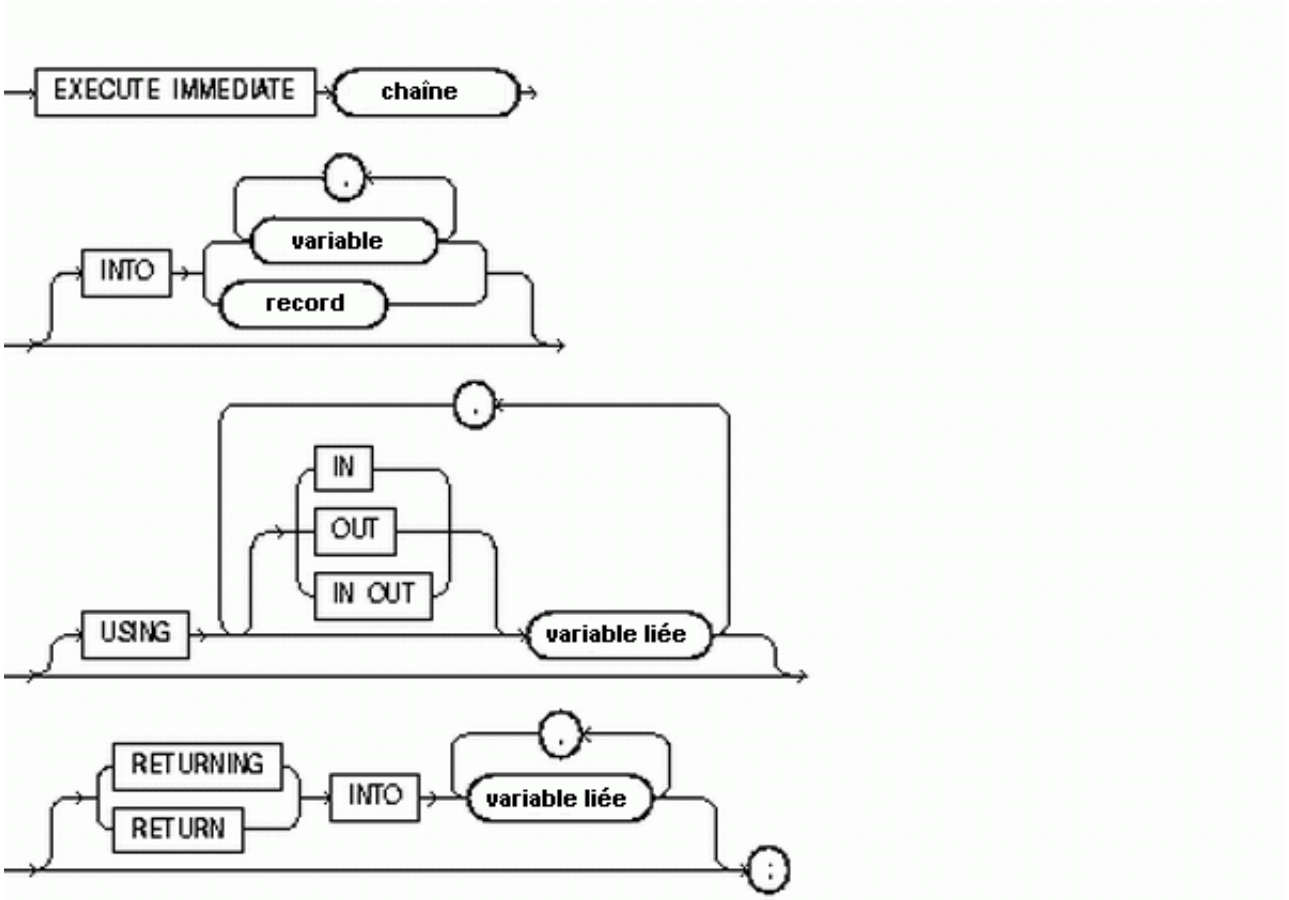


Le mot clé **WORK** est facultatif et n'a aucun effet particulier

Un commentaire d'un maximum de 50 caractères peut apparaître entre apostrophes derrière le mot clé **COMMENT**

1.2.6 - EXECUTE IMMEDIATE

Cette instruction permet d'exécuter du SQL dynamique



-> Pour une explication détaillée de cette instruction, reportez-vous à l'article sur le sql dynamique natif

1.2.7 - EXIT

Cette instruction permet de quitter une structure itérative

label facultatif permet de nommer précisément la structure dont on veut sortir.

expression booléenne permet de spécifier une condition de sortie

Exit saute à l'instruction suivant le mot clé **END LOOP;**

Dans le cas de boucles imbriquées, l'indication d'un label permet de quitter tout ou partie des boucles imbriquées

```
SQL> Declare
  2    LN$Num pls_integer := 0 ;
  3  Begin
  4    Loop
  5      LN$Num := LN$Num + 1 ;
  6      dbms_output.put_line( to_char( LN$Num ) ) ;
  7      EXIT WHEN LN$Num > 3 ; -- sortie de la boucle lorsque LN$Num est supétieur à 3
  8    End loop ;
  9  End ;
 10  /
1
2
3
4

Procédure PL/SQL terminée avec succès.
```

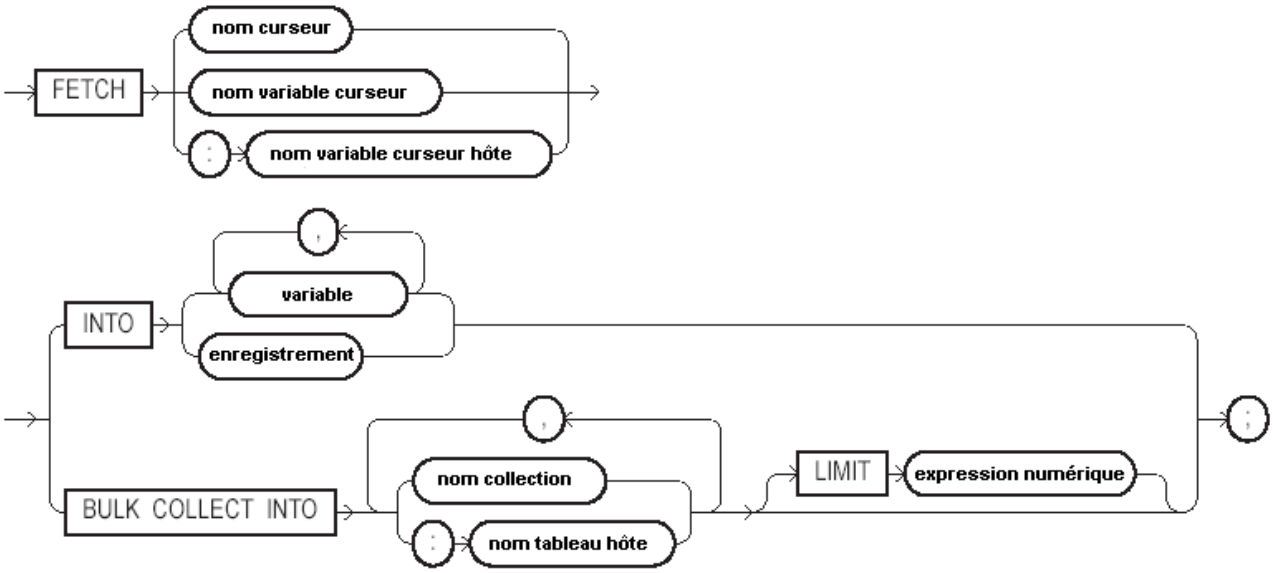
Lorsque le test effectué (LN$Num > 3) est vérifié (TRUE), la boucle Loop # End loop est quittée

```
SQL> Declare
  2    LN$I pls_integer := 0 ;
  3    LN$J pls_integer := 0 ;
  4  Begin
  5    <<boucle1>>
  6    Loop
  7      LN$I := LN$I + 1 ;
  8      Loop
  9         LN$J := LN$J + 1 ;
 10         dbms_output.put_line( to_char( LN$I ) || ',' || to_char( LN$J ) ) ;
 11         EXIT boucle1 WHEN LN$J > 3 ;
 12      End loop ;
 13    End loop ;
 14  End ;
 15  /
1,1
1,2
1,3
1,4

Procédure PL/SQL terminée avec succès.
```

Dans cet exemple, l'instruction **EXIT** suivie du label boucle1 permet de quitter les deux boucles imbriquées

1.2.8 - FETCH

Cette instruction permet de ramener une ligne d'un curseur préalablement ouvert avec l'instruction OPEN ou OPEN FOR



nom curseur représente le nom d'un curseur préalablement ouvert avec l'instruction OPEN ou OPEN FOR

nom variable curseur représente le nom d'une variable curseur

nom variable curseur hôte représente le nom d'une variable curseur transmise par un programme tiers (ex : Pro*C, Pro*Cobol, etc.)

variable représente le nom d'une variable préalablement définie dans la section déclarative, qui doit être du même type que la colonne ramenée par l'instruction Select

enregistrement représente le nom d'un enregistrement préalablement défini dans la section déclarative qui doit être du même type que la ligne ramenée par l'instruction Select

nom collection représente le nom d'une collection préalablement définie dans la section déclarative

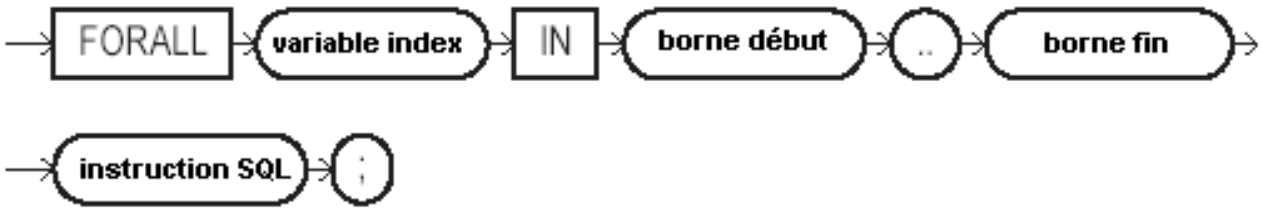
nom tableau hôte représente le nom du tableau transmis par un programme tiers

Si l'instruction **FETCH** ne ramène plus de ligne (fin du curseur) l'attribut %NOTFOUND prend la valeur TRUE et l'attribut %FOUND prend la valeur FALSE

-> Pour voir des exemples, reportez-vous à la section Les curseurs explicites (1.2.22)

1.2.9 - FORALL

Cette instruction permet de générer des ordres SQL de masse basés sur le contenu d'une collection



variable index représente l'indice de la collection sur laquelle porte l'instruction FORALL

borne début représente la valeur d'indice de départ

borne fin représente la valeur d'indice de fin

instruction sql doit être un ordre SQL de type INSERT, UPDATE ou DELETE

(10g)

FORALL i IN INDICES OF nom_collection

permet de ne traiter que les indices valorisés de la collection (non forcément consécutifs comme il était impératif dans les versions précédentes)

Pour voir des exemples, reportez-vous au chapitre 5 Collections et enregistrements

1.2.10 - GOTO

Cette instruction permet d'exécuter un saut dans le code vers le label precisé

Etiquette

Instruction GOTO

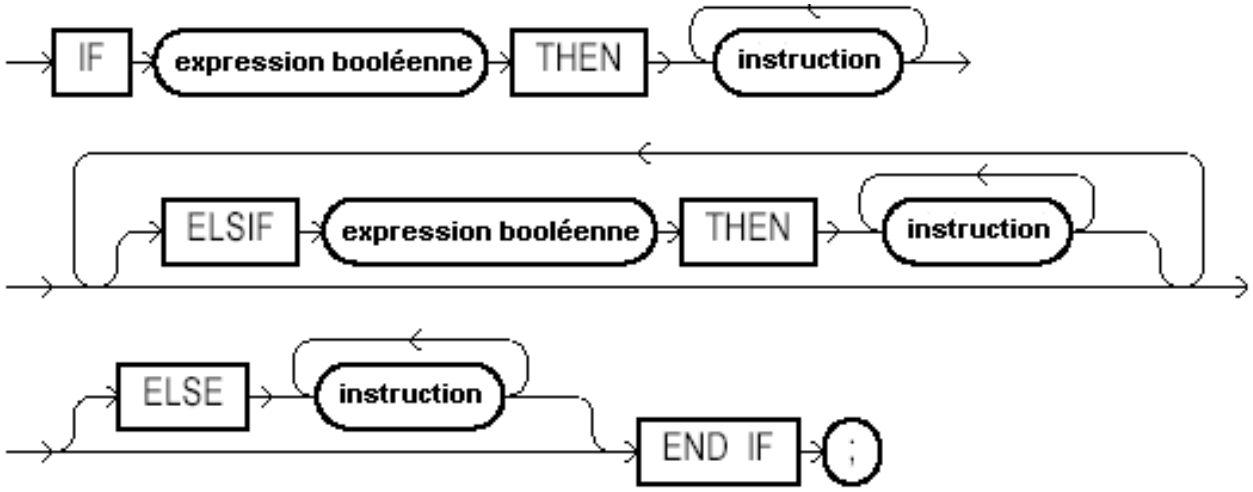
Une instruction valide doit suivre la déclaration du label

```
SQL> Declare
```

```
  2    LN$I pls_integer := 0 ;
  3    LN$J pls_integer := 0 ;
  4  Begin
  5    Loop
  6      LN$I := LN$I + 1 ;
  7      Loop
  8        LN$J := LN$J + 1 ;
  9        dbms_output.put_line( to_char( LN$I ) || ',' || to_char( LN$J ) ) ;
 10        If LN$J > 3  Then GOTO sortie ; End if ;
 11      End loop ;
 12    End loop ;
 13    <<sortie>>
 14    null ;
 15  End ;
 16  /
1,1
1,2
1,3
1,4

Procédure PL/SQL terminée avec succès.
```

1.2.11 - IF

Cette instruction permet de faire des tests conditionnels



expression booléenne représente un test générant un booléen TRUE ou FALSE

Seuls les mots clé **IF** et **END IF;** sont obligatoires. Les clauses **ELSIF** et **ELSE** sont facultatives

```
SQL> Declare
  2    LN$I pls_integer := 0 ;
  3    LN$J pls_integer := 0 ;
  4  Begin
  5    Loop
  6      LN$I := LN$I + 1 ;
  7      Loop
  8        LN$J := LN$J + 1 ;
  9        If LN$J = 1 Then
 10           dbms_output.put_line( '1' ) ;
 11        Elsif LN$J = 2 Then
 12           dbms_output.put_line( '2' ) ;
 13        Else
 14           dbms_output.put_line( '3' ) ;
```

```
 15           goto sortie ;
 16         End if ;
 17       End loop ;
 18     End loop ;
 19     <<sortie>>
 20     null ;
 21   End ;
 22   /
1
2
3

Procédure PL/SQL terminée avec succès.
```

1.2.12 - CASE

Cette instruction permet de mettre en place des structures de test conditionnel de type IF .. ELSE .. END IF,

à la grande différence qu'elle est utilisable dans les requêtes SQL

2 syntaxes sont possibles

- **CASE simple**

[<<label>>] CASE opérateur { WHEN contenu_opérateur THEN { instruction;} ... }... [ELSE { instruction;}...] END CASE [label];

- **CASE de recherche**

[<<label>>] CASE { WHEN expression_booléenne THEN { instruction;} ... }... [ELSE { instruction;}...] END CASE [label];

opérateur peut être n'importe quel type PL/SQL à l'exception des objets suivants :

- BLOB
- BFILE
- Type objet
- Enregistrement
- Collection (NESTED TABLE, INDEX-BY TABLE, VARRAY)

Pour le CASE simple, chaque mot clé **WHEN** vérifie l'égalité entre opérateur et contenu_opérateur. Dans l'affirmative, l'instruction suivant le mot clé **THEN** est exécutée, puis la structure **CASE** est quittée et l'exécution du programme est reprise après le mot clé **END CASE;**

```
SQL> Declare
  2    LN$Num pls_integer := 0 ;
  3  Begin
  4     Loop
  5       LN$Num := LN$Num + 1 ;
  6       CASE LN$Num
  7         WHEN 1 Then dbms_output.put_line( '1' ) ;
  8         WHEN 2 Then dbms_output.put_line( '2' ) ;
  9         WHEN 3 Then dbms_output.put_line( '3' ) ;
 10         ELSE
 11            EXIT ;
 12       END CASE ;
 13     End loop ;
 14  End ;
 15  /
1
2
3

Procédure PL/SQL terminée avec succès.
```

Exemple de CASE de recherche

```
SQL> Declare
  2    LN$Num pls_integer := 0 ;
  3  Begin
  4     Loop
  5       LN$Num := LN$Num + 1 ;
  6       CASE
  7         WHEN LN$Num between 1 and 3  Then dbms_output.put_line( To_char( LN$Num ) || ' -> 1-3'
) ;
  8         WHEN LN$Num < 5  Then dbms_output.put_line( To_char( LN$Num ) || ' < 5' ) ;
  9         ELSE dbms_output.put_line( To_char( LN$Num ) || ' >= 5' ) ;
 10       END CASE ;
 11       exit when LN$Num = 5 ;
 12     End loop ;
 13  End ;
 14  /
1 -> 1-3
2 -> 1-3
3 -> 1-3
4 < 5
5 >= 5

Procédure PL/SQL terminée avec succès.
```

Cette fois l'opérateur est précisé sur chaque ligne **WHEN**

Il ne s'agit alors plus d'un simple test d'égalité, mais de n'importe quelle expression booléenne restituant un résultat TRUE ou FALSE.

On observe également que le débranchement dans une clause **WHEN** est exclusif. En effet, dans chaque itération de boucle, la variable LN$Num est inférieure à 5, mais n'est prise en compte dans la deuxième clause **WHEN** que lorsque la première n'est plus vérifiée

Pour le CASE de recherche, l'omission de la clause ELSE provoque une erreur

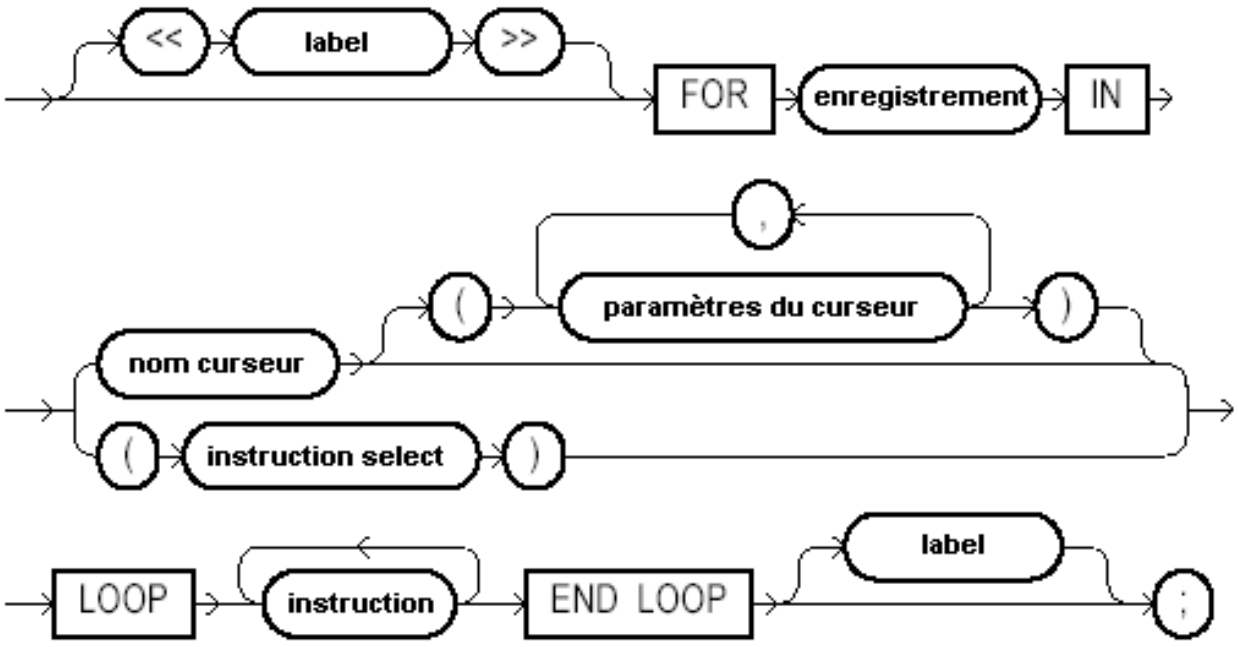
```
SQL> Declare
  2    LN$Num pls_integer := 0 ;
  3  Begin
  4     Loop
  5       LN$Num := LN$Num + 1 ;
```

```
  6        CASE
  7          WHEN LN$Num between 1 and 3  Then dbms_output.put_line( To_char( LN$Num ) || ' -> 1-3'
) ;
  8          WHEN LN$Num < 5  Then dbms_output.put_line( To_char( LN$Num ) || ' -> < 5' ) ;
  9        END CASE ;
 10        exit when LN$Num = 5 ;
 11      End loop ;
 12  End ;
 13  /
1 -> 1-3
2 -> 1-3
3 -> 1-3
4 -> < 5
Declare
*
ERREUR à la ligne 1 :
ORA-06592: CASE not found while executing CASE statement
ORA-06512: at line 6
```

1.2.13 - FOR (curseur)

Cette instruction permet de gérer un curseur sans utiliser les ordres OPEN, FETCH et CLOSE

Boucle sur curseur



enregistrement représente un nom de variable de type curseur implicite.

nom curseur représente le nom d'un curseur préalablement défini dans la section déclarative

```
SQL> Declare
  2     -- Déclaration du curseur
  3     CURSOR C_EMP IS
  4     Select
  5        *
  6     From
  7        EMP
```

```
  8     Where
  9        job = 'CLERK'
 10     ;
 11
 12  Begin
 13     For Cur IN C_EMP Loop
 14       dbms_output.put_line( To_char( Cur.empno ) || ' - ' || Cur.ename ) ;
 15     End loop ;
 16  End ;
 17  /
7369 - SMITH
7876 - ADAMS
7900 - JAMES
7934 - MILLER

Procédure PL/SQL terminée avec succès.
```

La variable de curseur implicite **Cur**, non définie dans la section déclarative, doit être utilisée pour manipuler dans la boucle,

les objets du curseur (To_char(Cur.empno),Cur.ename)

Après l'instruction **END LOOP;** l'utilisation de cette variable génère une erreur

Avec cette syntaxe, l'utilisation des instructions OPEN, FETCH et CLOSE est inutile

Instruction FOR et curseur paramétré

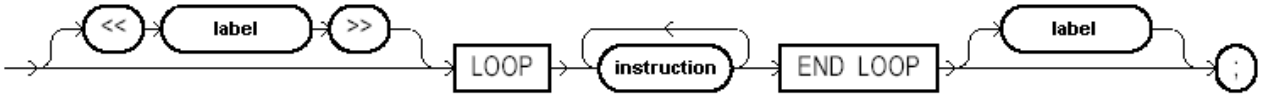
```
SQL> Declare
  2     -- Déclaration du curseur
  3     CURSOR C_EMP ( PC$Job IN EMP.job%Type ) IS
  4     Select
  5        *
  6     From
  7        EMP
  8     Where
  9        job = PC$Job
 10     ;
 11
 12  Begin
 13     For Cur IN C_EMP( 'SALESMAN' ) Loop
 14       dbms_output.put_line( To_char( Cur.empno ) || ' - ' || Cur.ename ) ;
 15     End loop ;
 16  End ;
 17  /
7499 - ALLEN
7521 - WARD
7654 - MARTIN
7844 - TURNER

Procédure PL/SQL terminée avec succès.
```

Le passage des paramètres s'effectue sur le curseur déclaré (C_EMP) et non sur la variable curseur (Cur)

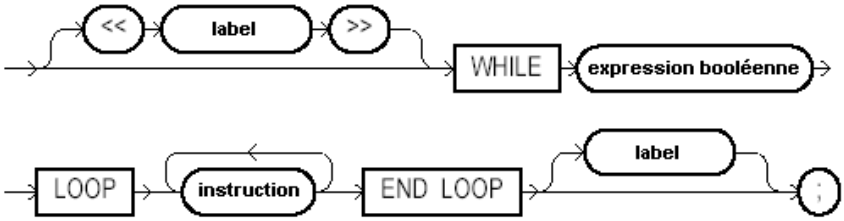
1.2.14 - FOR, LOOP, WHILE

Ces instructions déclarent une structure de type itérative (boucle)

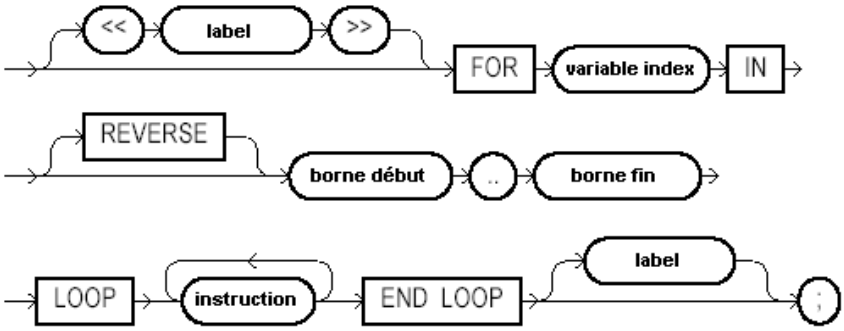
Boucle LOOP



Boucle WHILE



Boucle FOR



Trois syntaxes sont possibles

- **LOOP instruction;[instruction;[...]] END LOOP;**

Cette syntaxe met en place une boucle simple ou aucune condition de sortie n'est indiquée

Il faut donc une instruction **EXIT** pour sortir de ce type de boucle

```
SQL> Declare
  2    LN$I pls_integer := 0 ;
  3  Begin
  4    Loop
  5      LN$I := LN$I + 1 ;
  6      dbms_output.put_line( to_char( LN$I) ) ;
  7      exit when LN$I > 2 ;
  8    End loop ;
  9  End ;
 10  /
1
2
3
```

- **WHILE expression booléenne LOOP instruction;[instruction;[...]] END LOOP;**

Cette syntaxe permet de mettre en place une boucle dont la condition de test est évaluée au début.

Si expression booléenne donne le résultat FALSE, les instructions suivantes jusqu'au mot clé **END LOOP;** ne seront pas exécutées

```
SQL> Declare
  2    LN$I pls_integer := 0 ;
  3  Begin
  4    While LN$I < 3
  5    Loop
  6      LN$I := LN$I + 1 ;
  7      dbms_output.put_line( to_char( LN$I) ) ;
  8    End loop ;
  9  End ;
 10  /
1
2
3
```

- **FOR variable index IN [REVERSE] borne_début..borne_fin LOOP instruction;[instruction;[...]] END LOOP;**

Cette syntaxe permet de mettre en place une boucle dont le nombre d'itérations est fixé dès l'entrée

Variable index représente le nom de la variable qui servira d'indice. Cette variable ne nécessite pas de définition préalable dans la section déclarative

Reverse permet de faire varier l'indice dans le sens contraire (décrémentation)

borne début représente l'indice de départ

borne fin représente l'indice de fin

```
SQL> Declare
  2    LN$I pls_integer := 0 ;
  3  Begin
  4    For i in 1..3
  5    Loop
  6      dbms_output.put_line( to_char( i ) ) ;
  7    End loop ;
  8  End ;
  9  /
1
2
3
```

```
SQL> Declare
  2    LN$I pls_integer := 0 ;
  3  Begin
  4    For i in reverse 1..3
  5    Loop
  6      dbms_output.put_line( to_char( i ) ) ;
  7    End loop ;
  8  End ;
  9  /
```

```
3
2
1
```

1.2.15 - NULL

Cette instruction n'exécute rien et n'a aucun effet

1.2.16 - RAISE

Cette instruction permet de générer une exception

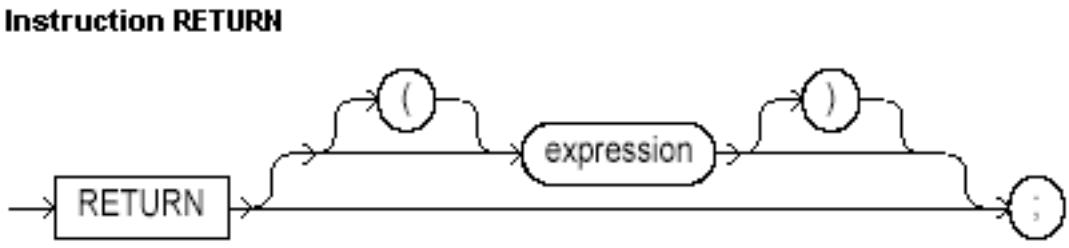
nom exception représente soit le nom d'une exception prédéfinie, soit une exception utilisateur définie dans la section déclarative

```
SQL> Declare
  2    LN$I pls_integer := 0 ;
  3    LE$Fin exception ;
  4  Begin
  5    Loop
  6      LN$I := LN$I + 1 ;
  7      dbms_output.put_line( to_char( LN$I) ) ;
  8      If LN$I > 2 Then
  9         RAISE LE$Fin ;
 10      End if ;
 11    End loop ;
 12  Exception
 13     When LE$Fin Then
 14        Null ;
 15  End ;
 16  /
1
2
3
```

Si la variable LN$I est > 2, alors on provoque le saut dans la section EXCEPTION avec l'erreur utilisateur LE$Fin

1.2.17 - RETURN

Cette instruction permet de sortir d'une procédure ou d'une fonction



expression représente la valeur de retour d'une fonction. Cette valeur doit être compatible avec le type défini dans la clause RETURN de la déclaration de fonction

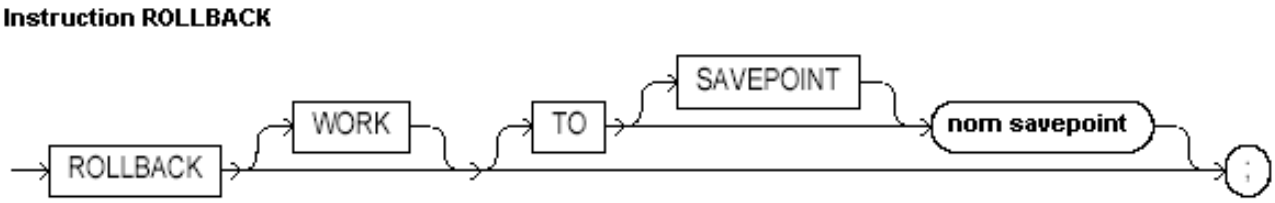
1.2.18 - SAVEPOINT

Cette instruction permet de placer une étiquette savepoint dans le corps du code.

Elle permet au traitement d'annuler, avec l'instruction **ROLLBACK**, les modifications effectuées à partir de cette étiquette

1.2.19 - ROLLBACK

Cette instruction permet d'annuler en base toutes les modifications effectuées au cours de la transaction



nom savepoint représente le nom d'une étiquette savepoint préalablement définie dans le corps du code avec l'instruction **SAVEPOINT**

Avec **TO SAVEPOINT** nom savepoint, l'annulation porte sur toutes les modifications effectuées à partir de l'étiquette nom savepoint

```
SQL> Begin
  2     Insert Into EMP( empno, ename, job )
  3       values( 9991, 'Dupontont', 'CLERK' ) ;
  4     Insert Into EMP( empno, ename, job )
  5       values( 9992, 'Duboudin', 'CLERK' ) ;
  6
  7     SAVEPOINT mise_a_jour ;
  8
  9     Update EMP Set sal = 2500 Where empno > 9990 ;
 10     ROLLBACK TO SAVEPOINT mise_a_jour ;
 11
 12     Commit ;
 13  End ;
 14  /

Procédure PL/SQL terminée avec succès.

SQL> Select * From EMP Where empno > 9990 ;

     EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM     DEPTNO
---------- ---------- --------- ---------- --------- ---------- ---------- ----------
      9991 Dupontont  CLERK
      9992 Duboudin   CLERK
```

Dans cet exemple, une étiquette **SAVEPOINT** est placée après les instructions d'insertion

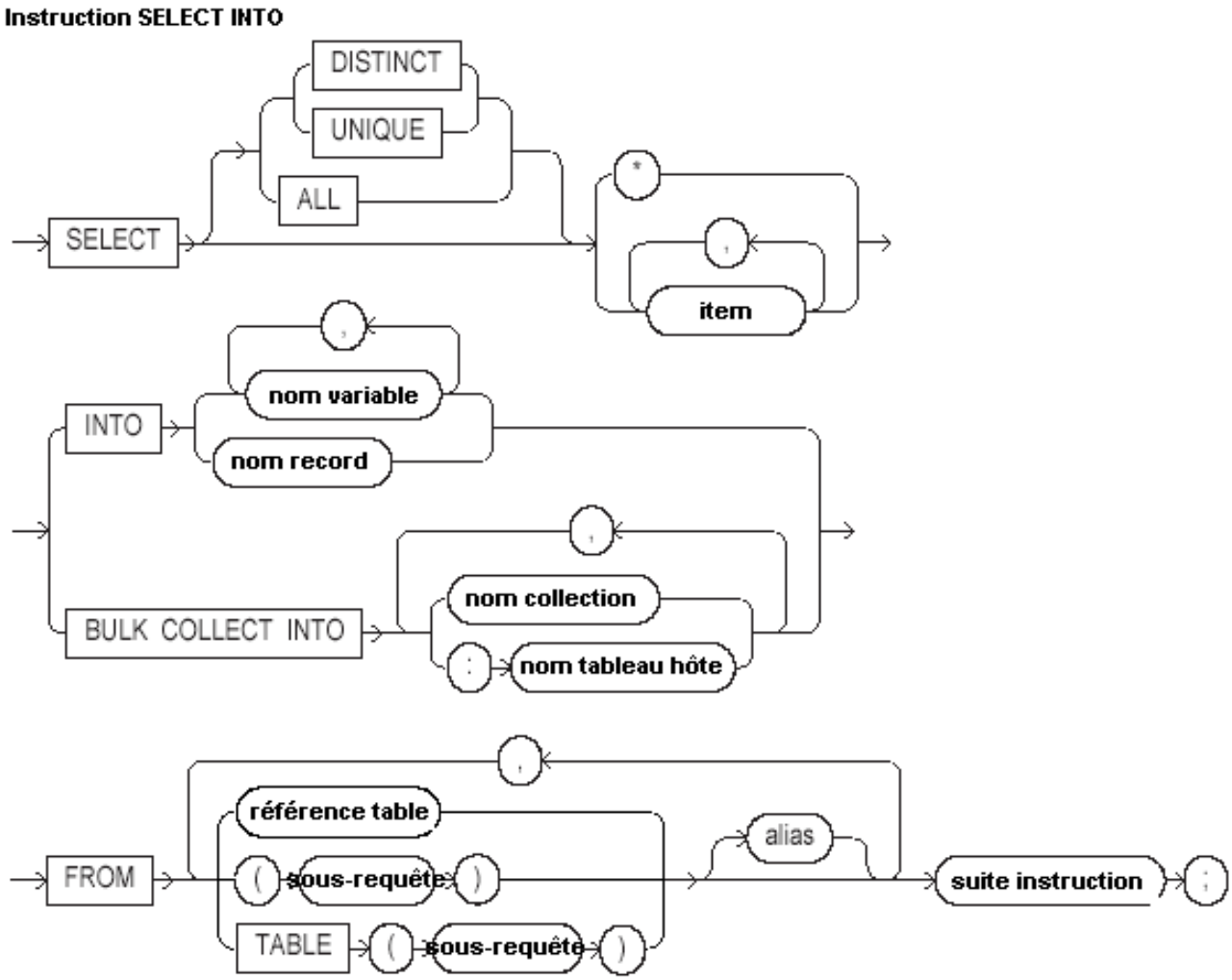
Un **ROLLBACK TO SAVEPOINT** est ajouté après l'instruction de mise à jour

puis un **COMMIT** est effectué

Les insertions sont bien enregistrées en base mais pas la mise à jour

1.2.20 - SELECT INTO

Séléction d'une ou de plusieurs lignes



Cette instruction permet d'exécuter un ordre Select implicite.

Cet ordre ne doit ramener qu'une ligne sous peine de générer l'exception NO_DATA_FOUND si aucune ligne n'est ramenée

ou TOO_MANY_ROWS si plus d'une ligne sont ramenées

Utilisée avec la clause **BULK COLLECT**, elle permet de charger une collection avec les lignes ramenées

item représente un littérale ou un nom de colonne

nom variable représente le nom d'une variable d'accueil. Son type doit être identique à celui de item

nom record représente le nom d'un enregistrement composé de champs de même type que les items ramenés

nom collection représente le nom d'une collection d'accueil

nom tableau hôte représente le nom de la variable tableau passée par un programme tiers

référence table représente la liste des tables et/ou vues de l'ordre SQL

sous-requête représente le texte d'une sous-requête SQL conforme

suite instruction représente la suite de l'ordre Select (clauses Where, Group by, Order by, etc.)

```
SQL> Declare
  2    LN$Num EMP.empno%Type ;
  3    LC$Nom EMP.ename%Type ;
  4    LC$Job EMP.job%Type ;
  5  Begin
  6    Select
  7       empno
  8      ,ename
  9      ,job
 10    Into
 11       LN$Num
 12      ,LC$Nom
 13      ,LC$Job
 14    From
 15      EMP
 16    Where
 17      empno = 7369
 18    ;
 19  End ;
 20  /

Procédure PL/SQL terminée avec succès.
```

Dans l'exemple suivant, aucun employé ne porte le numéro 1

la requête ne ramène donc aucune ligne et génère l'exception NO_DATA_FOUND

```
SQL> Declare
  2    LN$Num EMP.empno%Type ;
  3    LC$Nom EMP.ename%Type ;
  4    LC$Job EMP.job%Type ;
  5  Begin
  6    Select
  7       empno
  8      ,ename
  9      ,job
 10    Into
 11       LN$Num
 12      ,LC$Nom
 13      ,LC$Job
 14    From
 15      EMP
 16    Where
 17      empno = 1
 18    ;
 19  End ;
 20  /
Declare
*
```

```
ERREUR à la ligne 1 :
ORA-01403: Aucune donnée trouvée
ORA-06512: à ligne 7
```

Dans l'exemple suivant, la clause WHERE a été retirée

la requête ramène donc plusieurs lignes et génère l'exception TOO_MANY_ROWS

```
SQL> Declare
  2    LN$Num EMP.empno%Type ;
  3    LC$Nom EMP.ename%Type ;
  4    LC$Job EMP.job%Type ;
  5  Begin
  6    Select
  7       empno
  8      ,ename
  9      ,job
 10    Into
 11       LN$Num
 12      ,LC$Nom
 13      ,LC$Job
 14    From
 15      EMP
 16    ;
 17  End ;
 18  /
Declare
*
ERREUR à la ligne 1 :
ORA-01422: l'extraction exacte ramène plus que le nombre de lignes demandé
ORA-06512: à ligne 7
```

Dans l'exemple suivant toute une ligne de la table EMP est chargée dans un enregistrement

```
SQL> Declare
  2    LR$Emp EMP%Rowtype ;
  3  Begin
  4    Select
  5       *
  6    Into
  7       LR$Emp
  8    From
  9      EMP
 10    Where
 11      empno = 7369
 12    ;
 13  End ;
 14  /

Procédure PL/SQL terminée avec succès.
```

Dans l'exemple suivant toutes les lignes de la table EMP sont chargées dans une collection

```
SQL> Declare
  2    TYPE TYP_TAB_EMP IS TABLE OF EMP%Rowtype ;
  3    Tabemp TYP_TAB_EMP ;
  4  Begin
  5    Select
  6       *
  7    BULK COLLECT
  8    Into
  9       Tabemp
 10    From
 11      EMP
 12    ;
 13    For i IN Tabemp.first..Tabemp.last Loop
 14      dbms_output.put_line( To_char( Tabemp(i).empno ) || ' - ' || Tabemp(i).ename ) ;
 15    End loop ;
 16  End ;
 17  /
```

```
7369 - SMITH
7499 - ALLEN
7521 - WARD
7566 - JONES
7654 - MARTIN
7698 - BLAKE
7782 - CLARK
7788 - SCOTT
7839 - KING
7844 - TURNER
7876 - ADAMS
7900 - JAMES
7902 - FORD
7934 - MILLER

Procédure PL/SQL terminée avec succès.
```

1.2.21 - Instruction SQL

Représente toute instruction SQL valide

- **INSERT**
- **UPDATE**
- **DELETE**

1.2.22 - Les curseurs explicites

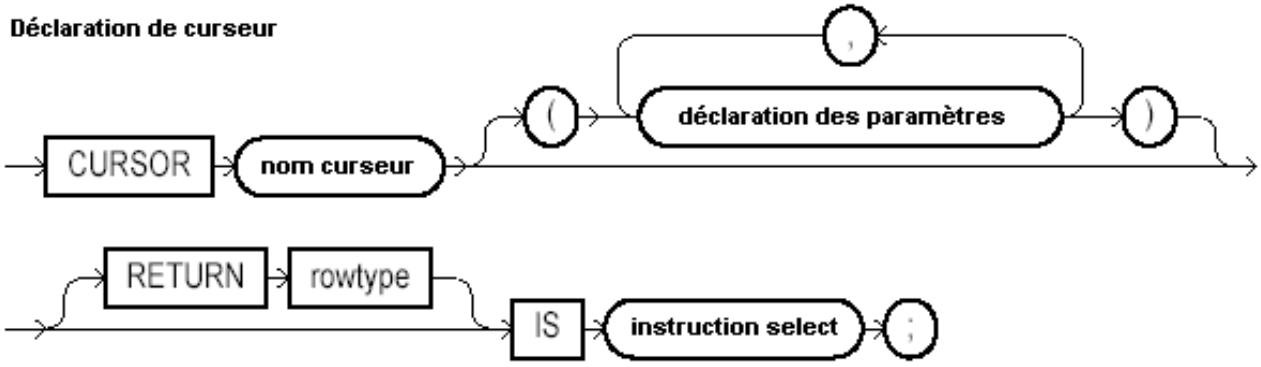
Un curseur est une zone mémoire de taille fixe, utilisée par le moteur SQL pour analyser et interpréter un ordre SQL

Un curseur explicite, contrairement au curseur implicite (SELECT INTO) est géré par l'utilisateur

pour traiter un ordre Select qui ramène plusieurs lignes

Tout curseur explicite géré dans la section exécution doit avoir été déclaré dans la section déclarative

- **Déclaration**



nom curseur représente le nom du curseur que l'on déclare

déclaration des paramètres(facultatif) représente la liste des paramètres transmis au curseur

instruction select représente l'ordre SQL Select d'alimentation du curseur

```
SQL> Declare
  2    -- déclaration du curseur
  3    CURSOR C_EMP IS
  4    Select
  5       empno
  6      ,ename
  7      ,job
  8    From
  9      EMP
 10    ;
 11    -- variables d'accueil
 12    LN$Num EMP.empno%Type ;
 13    LC$Nom EMP.ename%Type ;
 14    LC$Job EMP.job%Type ;
 15  Begin
 16    Open C_EMP ; -- ouverture du curseur
 17    Loop -- boucle sur les lignes
 18      Fetch C_EMP Into LN$Num, LC$Nom, LC$Job ; -- Lecture d'une ligne
 19      Exit When C_EMP%NOTFOUND ; -- sortie lorsque le curseur ne ramène plus de ligne
 20    End loop ;
 21    Close C_EMP ; -- fermeture du curseur
 22  End ;
 23  /

Procédure PL/SQL terminée avec succès.
```

Un curseur nommé C_EMP est déclaré avec l'ordre Select correspondant (CURSOR C_EMP IS...)

Il est ouvert avec l'instruction **OPEN**

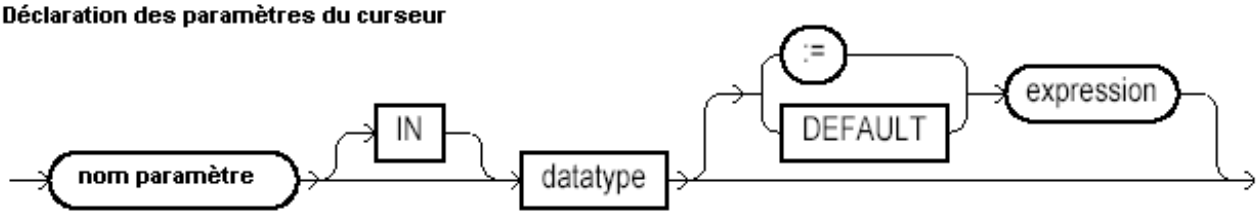
lu avec l'instruction **FETCH**

et fermé avec l'instruction **CLOSE**

- **Paramètres du curseur**

Un curseur est paramétrable. On peut donc utiliser le même curseur pour obtenir différents résultats

Déclaration des paramètres du curseur



nom paramètre représente le nom de la variable paramètre

datatype représente le type SQL de la variable paramètre (doit correspondre en type avec la colonne visée)

expression représente la valeur par défaut du paramètre (doit correspondre en type avec celui du paramètre)

```
SQL> Declare
  2    -- déclaration du curseur
  3    CURSOR C_EMP ( PN$Num IN EMP.empno%Type )IS
  4    Select
  5       empno
  6      ,ename
  7      ,job
  8    From
  9      EMP
 10    Where
 11      empno = PN$Num
 12    ;
 13    -- variables d'accueil
 14    LN$Num EMP.empno%Type ;
 15    LC$Nom EMP.ename%Type ;
 16    LC$Job EMP.job%Type ;
 17  Begin
 18    Open C_EMP( 7369 ) ; -- ouverture du curseur avec passage du paramètre 7369
 19    Loop
 20      Fetch C_EMP Into LN$Num, LC$Nom, LC$Job ; -- Lecture d'une ligne
 21      Exit When C_EMP%NOTFOUND ; -- sortie lorsque le curseur ne ramène plus de ligne
 22      dbms_output.put_line( 'Employé ' || To_char(LN$Num) || ' ' || LC$Nom ) ;
 23    End loop ;
 24    Close C_EMP ; -- fermeture du curseur
 25    Open C_EMP( 7521 ) ; -- ouverture du curseur avec passage du paramètre 7521
 26    Loop
 27      Fetch C_EMP Into LN$Num, LC$Nom, LC$Job ; -- Lecture d'une ligne
 28      Exit When C_EMP%NOTFOUND ; -- sortie lorsque le curseur ne ramène plus de ligne
 29      dbms_output.put_line( 'Employé ' || To_char(LN$Num) || ' ' || LC$Nom ) ;
 30    End loop ;
 31    Close C_EMP ; -- fermeture du curseur
 32  End ;
 33  /
Employé 7369 SMITH
Employé 7521 WARD

Procédure PL/SQL terminée avec succès.
```

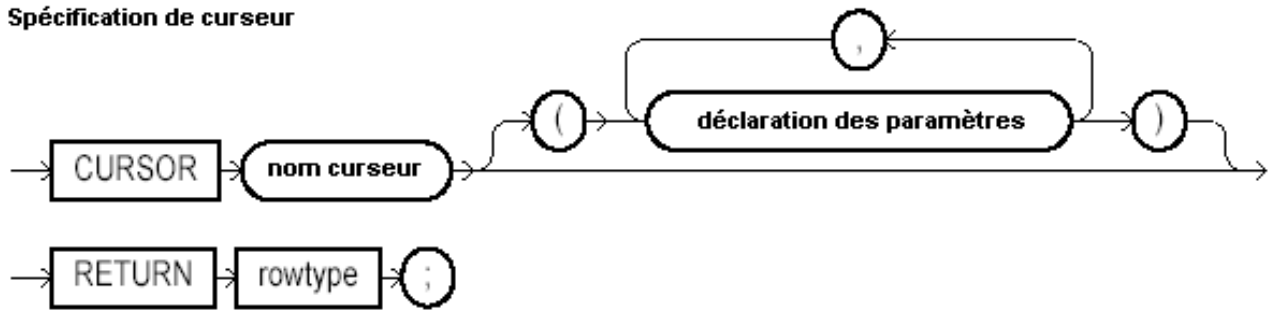
- **Déclaration d'une variable curseur**

nom variable curseur représente le nom de la variable curseur déclarée

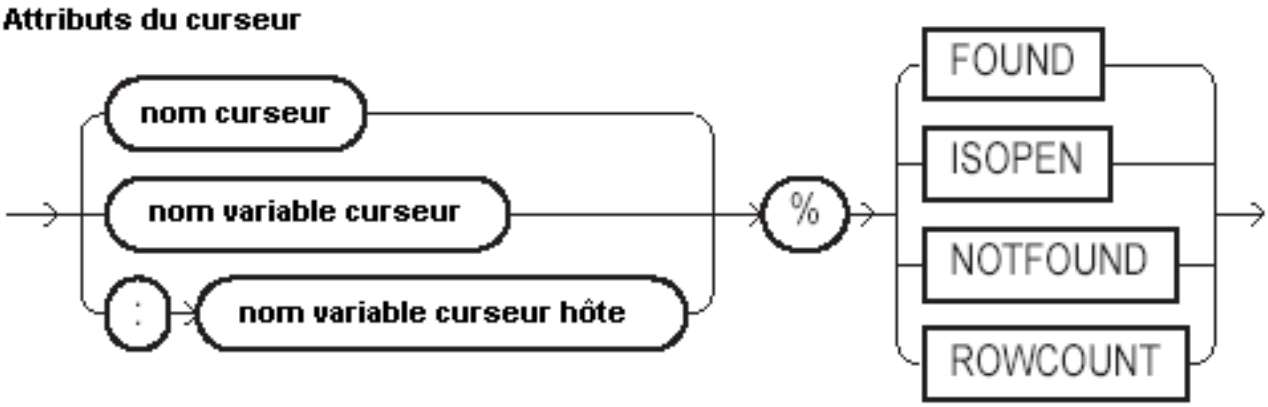
nom type représente le nom d'un type curseur

```
SQL> Declare
  2     TYPE TYP_REF_CUR IS REF CURSOR ;
  3     -- variable curseur
  4     CEMP TYP_REF_CUR ;
  5     -- variables d'accueil
  6     LN$Num EMP.empno%Type ;
  7     LC$Nom EMP.ename%Type ;
  8     LC$Job EMP.job%Type ;
  9  Begin
 10     Open CEMP For 'Select empno, ename, job From EMP'; -- ouverture du curseur
 11     Loop
 12       Fetch CEMP Into LN$Num, LC$Nom, LC$Job ; -- Lecture d'une ligne
 13       Exit When CEMP%NOTFOUND ; -- sortie lorsque le curseur ne ramène plus de ligne
 14     End loop ;
 15     Close CEMP ; -- fermeture du curseur
 16  End ;
 17  /

Procédure PL/SQL terminée avec succès.
```

- **Spécification d'un curseur**



Les attributs de curseur



Chaque curseur dispose de 4 attributs

- **%FOUND**

Cet attribut prend la valeur TRUE lorsque une ligne est ramenée, sinon il prend la valeur FALSE

- **%NOTFOUND**

Cet attribut prend la valeur FALSE lorsque une ligne est ramenée, sinon il prend la valeur TRUE

- **%ISOPEN**

Cet attribut prend la valeur TRUE lorsque le curseur indiqué est ouvert, sinon il prend la valeur FALSE

- **%ROWCOUNT**

Cet attribut retourne le nombre de lignes impactées par la dernière instruction SQL

1.2.23 - Portée des variables

La portée ou visibilité d'une variable est limitée au bloc PL/SQL dans laquelle elle est déclarée. Elle est donc locale au bloc PL/SQL

```
SQL>  Begin
  2
  3     Declare
  4       LC$Ch1    varchar2(10) := 'Phrase 2';
  5     Begin
  6        dbms_output.put_line( 'LC$Ch1 = ' || LC$Ch1 ) ;
  7     End ;
  8
  9     dbms_output.put_line( 'LC$Ch1 = ' || LC$Ch1 ) ;
```

```
 10
 11   End ;
 12  /
   dbms_output.put_line( 'LC$Ch1 = ' || LC$Ch1 ) ;
                                        *
ERREUR à la ligne 9 :
ORA-06550: line 9, column 41:
PLS-00201: identifier 'LC$CH1' must be declared
ORA-06550: line 9, column 4:
PL/SQL: Statement ignored
```

Dans cet exemple, la variable LC$Ch1 déclarée dans le sous-bloc, n'existe plus dans le bloc principal

Dans le cas de blocs imbriqués ou une même variable est déclarée dans chaque bloc, la visibilité d'une variable se rapporte toujours à la plus proche déclaration

```
SQL>  Declare
  2     LC$Ch1    varchar2(10) := 'Phrase 1';
  3   Begin
  4
  5     Declare
  6       LC$Ch1    varchar2(10) := 'Phrase 2';
  7     Begin
  8       dbms_output.put_line( 'LC$Ch1 = ' || LC$Ch1 ) ;
  9     End ;
 10
 11     dbms_output.put_line( 'LC$Ch1 = ' || LC$Ch1 ) ;
 12
 13   End ;
 14  /
LC$Ch1 = Phrase 2
LC$Ch1 = Phrase 1

Procédure PL/SQL terminée avec succès.
```

1.3 - La section de gestion des erreurs

Débutée par le mot clé **EXCEPTION**, elle contient le code mis en oeuvre pour la gestion des erreurs générées par la section d'exécution

Une erreur survenue lors de l'exécution du code déclenche ce que l'on nomme une exception. Le code erreur associé est transmis à la section **EXCEPTION**, pour vous laisser la possibilité de la gérer et donc de ne pas mettre fin prématurément à l'application.

Prenons l'exemple suivant :

Nous souhaitons retrouver la liste des employés dont la date d'entrée est inférieure au premier janvier 1970

```
SQL> Declare
  2    LC$Nom EMP.ename%Type ;
  3  Begin
  4   Select empno
  5   Into LC$Nom
  6   From EMP
  7   Where hiredate < to_date('01/01/1970','DD/MM/YYYY') ;
  8  End ;
```

```
  9  /
Declare
*
ERREUR à la ligne 1 :
ORA-01403: Aucune donnée trouvée
ORA-06512: à ligne 4
```

Comme la requête ne ramène aucune ligne, l'exception prédéfinie NO_DATA_FOUND est générée et transmise à la section **EXCEPTION** qui peut traiter le cas et poursuivre l'exécution de l'application.

L'exception NO_DATA_FOUND (ORA_01403) correspond au code erreur numérique +100.

Il existe des milliers de code erreur Oracle et il serait vain de tous leur donner un libellé.

Voici la liste des exceptions prédéfinies qui bénéficient de ce traitement de faveur :

Exception prédéfinie	Erreur Oracle	Valeur de SQLCODE
ACCESS_INTO_NULL	ORA-06530	-6530
CASE_NOT_FOUND	ORA-06592	-6592
COLLECTION_IS_NULL	ORA-06531	-6531
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
SELF_IS_NULL	ORA-30625	-30625
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

Toutes les autres exceptions doivent être interceptées via leur code erreur numérique.

En plus des erreurs Oracle, vous pouvez intercepter vos propres erreurs en déclarant des variables dont le type est **exception** et en provoquant vous-même le saut dans la section de gestion des erreurs à l'aide de l'instruction **RAISE**

```
DECLARE
   LE$Fin  Exception ;
   #
Begin
   #..
   Raise LE$Fin ;
```

```
   #
EXCEPTION
   WHEN LE$Fin Then
      ###.
END ;
```

Il n'est pas possible de déclarer la même exception deux fois dans le même bloc. Toutefois, dans le cas de blocs imbriqués, vous pouvez déclarer la même exception dans la section **EXCEPTION** de chaque bloc

```
DECLARE
   LE$Fin  Exception ;
BEGIN
   DECLARE
      LE$Fin  Exception ;
   BEGIN
      ...
   EXCEPTION
      WHEN LE$Fin Then
        ...
   END ;
EXCEPTION
   WHEN LE$Fin Then
      ...
END ;
```

Le peu d'exceptions prédéfinies vous oblige à traiter tous les autres cas dans la clause WHEN OTHERS en testant le code erreur SQL

```
EXCEPTION
   WHEN NO_DATA_FOUND Then
      ...
   WHEN OTHERS THEN
     If SQLCODE = # Then #
     Elsif SQLCODE = # Then #
        ...
     End if ;
END;
```

Vous pouvez associer un code erreur Oracle à vos propres variables exception à l'aide du mot clé **PRAGMA EXCEPTION_INIT**, dans le cadre de la section déclarative de la façon suivante :

Nom_exception EXCEPTION ;

PRAGMA EXCEPTION_INIT(nom_exception, -code_error_oracle);

Exemple :

Lorsque l'on tente d'insérer plus de caractères dans une variable que sa déclaration ne le permet, Oracle déclenche une erreur -6502. Nous allons "nommer" cette erreur en LE$trop_long et l'intercepter dans la section exception

```
SQL> Declare
  2    LC$Chaine varchar2(10) ;
  3    LE$trop_long exception ;
  4    pragma exception_init( LE$trop_long, -6502 ) ;
  5  Begin
  6    LC$Chaine := rpad( ' ', 30) ;
  7  Exception
  8    when LE$trop_long then
```

```
  9        dbms_output.put_line( 'Chaîne de caractères trop longue') ;
 10  End ;
 11  /
Chaîne de caractères trop longue

Procédure PL/SQL terminée avec succès.

SQL>
```

Le code erreur numérique Oracle ayant généré la plus récente erreur est récupérable en interrogeant la fonction **SQLCODE**.

Le libellé erreur associé est récupérable en interrogeant la fonction **SQLERRM**

```
SQL> Declare
  2    LC$Chaine varchar2(10) ;
  3  Begin
  4    LC$Chaine := rpad( ' ', 30) ;
  5  Exception
  6    when others then
  7       dbms_output.put_line( 'Code    erreur : ' || to_char( SQLCODE )) ;
  8       dbms_output.put_line( 'libellé erreur : ' || to_char( SQLERRM )) ;
  9  End ;
 10  /
Code    erreur : -6502
libellé erreur : ORA-06502: PL/SQL: numeric or value error: character string buffer too small

Procédure PL/SQL terminée avec succès.
```

Poursuite de l'exécution après l'interception d'une exception

Une fois dans la section **EXCEPTION**, il n'est pas possible de retourner dans la section exécution juste après l'instruction qui a généré l'erreur.

Par contre il est tout à fait possible d'encadrer chaque groupe d'instructions voire même chaque instruction avec les mots clé

BEGIN # EXCEPTION # END;

Cela permet de traiter l'erreur et de continuer l'exécution

```
  1  Declare
  2    LC$Ch1    varchar2(20) := 'Phrase longue';
  3    LC$Chaine varchar2(10) ;
  4    LE$trop_long exception ;
  5    pragma exception_init( LE$trop_long, -6502 ) ;
  6  Begin
  7    Begin
  8       LC$Chaine := LC$Ch1;
  9    Exception
 10      when LE$trop_long then
 11          LC$Chaine := Substr( LC$Ch1, 1, 10 ) ;
 12    End ;
 13    -- poursuite du traitement --
 14    dbms_output.put_line(LC$Chaine ) ;
 15*  End ;
 16  /
Phrase lon
```

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

Vous pouvez également définir vos propres messages d'erreur avec la procédure RAISE_APPLICATION_ERROR

DBMS_STANDARD.raise_application_error(numero_erreur, message[, {TRUE | FALSE}])

numero_erreur représente un entier négatif compris entre -20000 et -20999

message représente le texte du message d'une longueur maximum de 2048 octets

TRUE indique que l'erreur est ajoutée à la pile des erreurs précedentes

FALSE indique que l'erreur remplace toutes les erreurs précédentes

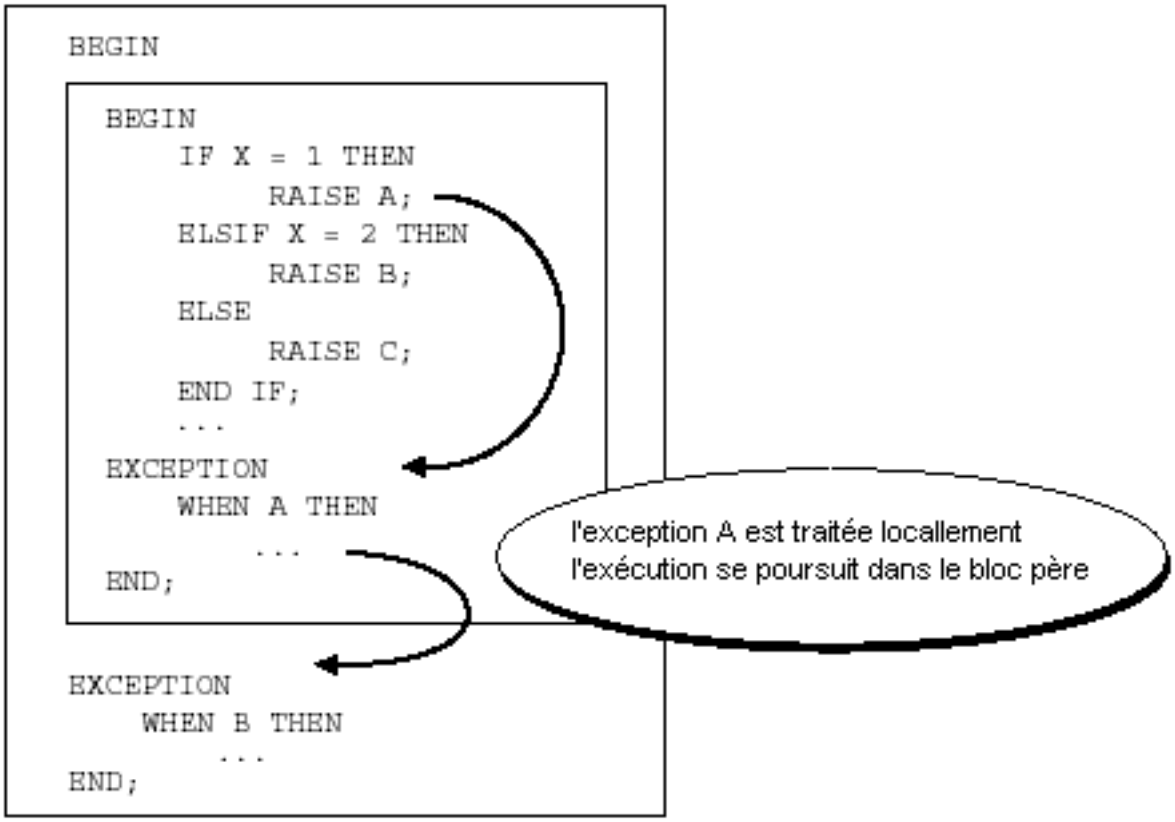
Du fait que cette procédure appartienne a un paquetage standard, il n'est pas nécessaire de préfixer cette procédure

L'appel de la procédure raise_application_error ne peut être exécuté que depuis une procédure stockée, et déclenche un retour immédiat au programme appelant en lui transmettant le code et le libellé de l'erreur

Propagation des exceptions

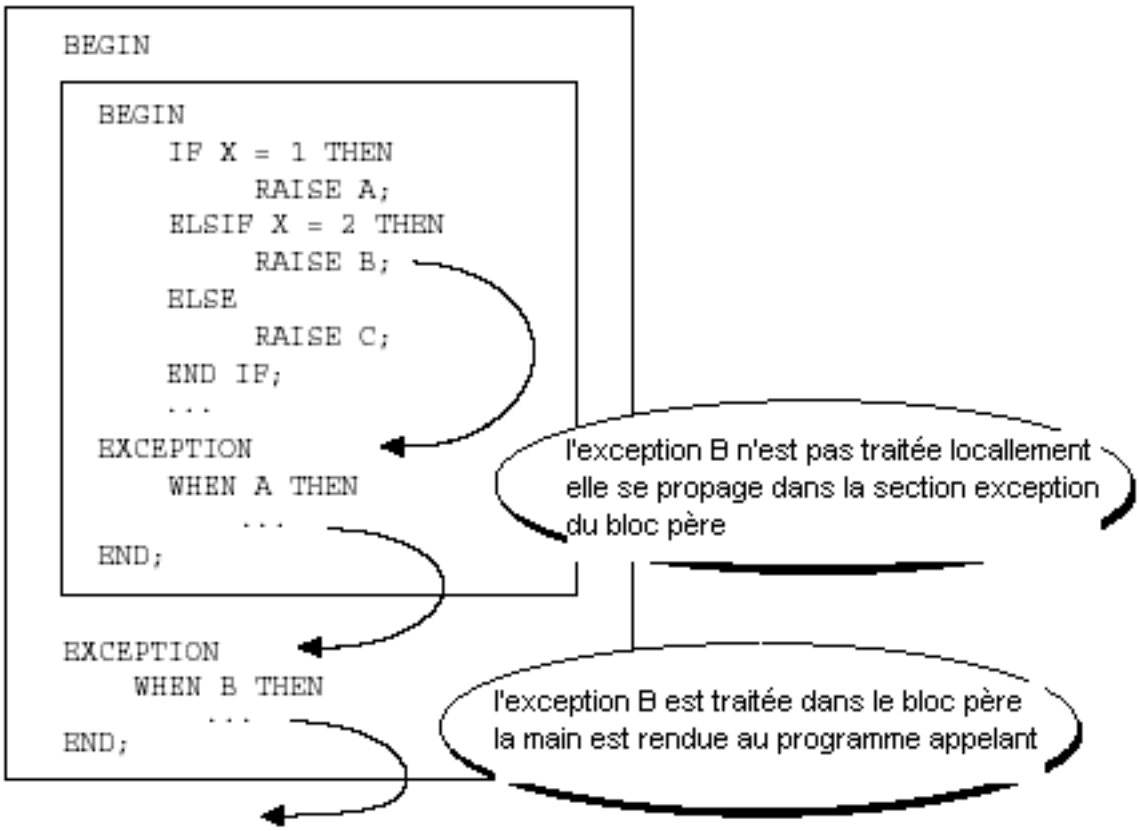
Si une exception n'est pas traitée au sein du bloc **BEGIN # END;** dans lequel elle est générée,

elle remonte de bloc en bloc jusqu'à ce qu'elle soit traitée ou rende la main au programme appelant.



Dans cet exemple, l'exception A est traitée dans le bloc local.

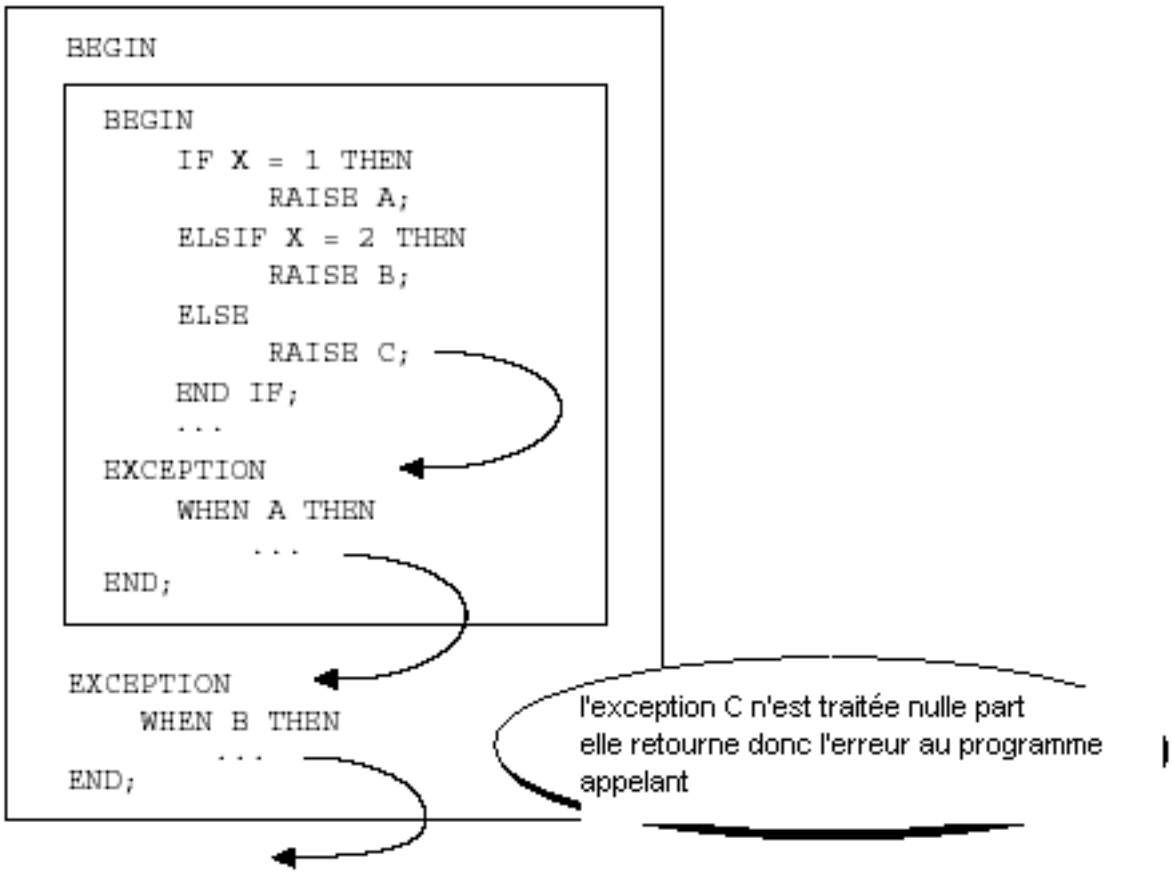
Le traitement se poursuit dans le bloc parent



Dans cet exemple, l'exception B n'est pas traitée dans le bloc local.

Elle se propage dans le bloc parent dans lequel elle est traitée

Puis la main est rendue au programme appelant



Dans cet exemple, l'exception C n'est traitée ni dans le bloc local ni dans les blocs parents

la main est rendue au programme appelant avec le code erreur

Commentaires dans les blocs PL/SQL

Pour mettre une ligne unique en commentaire, il faut la faire précéder d'un double tiret **--**

-- cette ligne seule est un commentaire

Pour mettre plusieurs lignes en commentaire, il faut les encadrer avec les symboles **/*** et ***/**

/* toutes les lignes suivantes

sont en commentaire

elles ne seront ni compilées

ni exécutées

*/

2 - Les variables, types et littéraux

2.1 - Les variables

nom variable [CONSTANT] type [[NOT NULL] := expression] ;

nom variable représente le nom de la variable composé de lettres, chiffres, $, _ ou #

Le nom de la variable ne peut pas excéder 30 caractères

CONSTANT indique que la valeur ne pourra pas être modifiée dans le code du bloc PL/SQL

NOT NULL indique que la variable ne peut pas être NULL, et dans ce cas **expression** doit être indiqué.

type représente de type de la variable correspondant à l'un des types suivants :

(dans le premier tableau, **les types Oracle sont en gras**, les sous-types compatible ANSI/ISO en normal)

Types scalaires				
BINARY_INTEGER	DEC	DECIMAL	DOUBLE PRECISION	FLOAT
INT	INTEGER	NATURAL	NATURALN	**NUMBER**
NUMERIC	**PLS_INTEGER**	POSITIVE	POSITIVEN	REAL
SIGNTYPE	SMALLINT	**CHAR**	CHARACTER	**LONG**
LONG RAW	**NCHAR**	**NVARCHAR2**	**RAW**	**ROWID**
STRING	**UROWID**	**VARCHAR**	**VARCHAR2**	
BOOLEAN	**DATE**			
INTERVAL DAY TO SECOND(9i)	**INTERVAL YEAR TO MONTH(9i)**	**TIMESTAMP(9i)**	**TIMESTAMP WITH LOCAL TIME ZONE(9i)**	**TIMESTAMP WITH TIME ZONE(9i)**

Types composés		
RECORD	TABLE	VARRAY

Types références	
REF CURSOR	REF type_objet

Types grands objets			
BFILE	BLOB	CLOB	NCLOB

Types suplémentaires				
SYS.ANYDATA	SYS.ANYTYPE	SYS.ANYDATASET		
XMLTYPE	URITYPE			
MDSYS.SDO_GEOMETRY				
ORDSYS.ORDAUDIO	ORDSYS.ORDIMAGE	ORDSYS.ORDVIDEO	ORDSYS.ORDDOC	ORDSYS.ORDIMAGESIGNATU

Vous pouvez également créer des sous-types :

SUBTYPE nom_sous-type IS type ;

SUBTYPE entier_court IS SMALLINT ;

i entier_court ;

Et utiliser les types dérivés

%TYPE

référence à un type existant qui est soit une colonne d'une table soit un type défini précédemment

nom_variable nom_table.nom_colonne%TYPE ;

nom_variable nom_variable_ref%TYPE ;

%ROWTYPE

référence à une ligne d'une table ou d'un curseur

nom_variable nom_table%ROWTYPE ;

nom_variable nom_curseur%ROWTYPE ;

```
Declare
  -- variable de même type que le colonne ENAME de la table EMP
  LC$Nom EMP.ENAME%TYPE ;
  -- variable de même type qu'une ligne de la table EMP
  LR$EMP    EMP%ROWTYPE ;
  LC$Dat1 DATE ;
```

```
    -- variable de même type que LC$Dat1 (DATE)
    LC$Dat2   LC$Dat1%TYPE ;
    -- Curseur --
    Cursor C_EMP is
    Select empno, ename, job From EMP ;
    -- variable de type ligne du curseur C_EMP
    LR$C_emp C_EMP%ROWTYPE ;
```

2.2 - Types prédéfinis

2.2.1 - Types caractères

CHAR[(n)]

Chaîne de caractères de longueur fixe avec n compris entre 1 et 32767 (par défaut 1)

VARCHAR2(n)

Chaîne de caractères de longueur variable avec n compris entre 1 et 32767

Ces types PL/SQL ont une capacité supérieure à celle des colonnes de tables de même type.

(une colonne CHAR ne peut excéder 2000 caractères et une colonne de type VARCHAR2 4000 caractères)

LONG

Chaîne de caractères de longueur variable avec au maximum 32760 octets

RAW[(n)]

Chaîne de caractères ou données binaires de longueur variable avec n compris entre 1 et 32767. Le contenu d'une variable de ce type n'est pas interprété par PL/SQL (pas de gestion des caractères nationaux)

LONG RAW

Identique au type LONG qui peut contenir des données binaires

Jeux de caractères multi-octets

NCHAR[(n)]

Chaîne de caractères de longueur fixe avec n compris entre 1 et 32767 (par défaut 1)

NVARCHAR2[(n)]

Chaîne de caractères de longueur variable avec n compris entre 1 et 32767

Le nombre de caractères réellement stockés dépend du nombre d'octets utilisés pour coder chaque caractère

UROWID, ROWID

Permet de stocker l'adresse absolue d'une ligne dans une table sous la forme d'une chaîne de caractères

Le format d'une telle variable est le suivant :

000000FFFBBBBBBRRR

000000 représente le numéro de l'objet qui possède cette ligne (dans le cas de cluster, plusieurs objets peuvent partager le même segment)

FFF représente le numéro du fichier qui contient la ligne

BBBBBB représente le numéro du bloc dans le fichier

RRR représente le numéro de ligne dans le bloc

2.2.2 - Types numériques

NUMBER[(e,d)]

Nombre réel avec e chiffres significatifs stockés et d décimales

BINARY_INTEGER

Nombre entier compris entre -2 147 483 647 et +2 147 483 647

(Utilise les fonctions de la librairie arithmétique)

(10g)BINARY_FLOAT

Nombre à virgule flottante simple précision au format IEEE 754

un littéral de ce type est écrit avec un f terminateur (ex. 3.125f)

(10g)BINARY_DOUBLE

Nombre à virgule flottante double précision au format IEEE 754

un littéral de ce type est écrit avec un d terminateur (ex. 3.12548d)

PLS_INTEGER

Nombre entier compris entre -2 147 483 647 et +2 147 483 647

(Plus rapide que BINARY_INTEGER car il utilise les registres du processeur)

2.2.3 - Types pour les grands objets

BFILE

Stocke la référence vers un fichier du système d'exploitation

BLOB

Permet de stocker un objet binaire jusqu'à 4 Go

CLOB

Permet de stocker un ensemble de caractères, jusqu'à 4 Go

NCLOB

Permet de stocker un ensemble de caractères, codés sur un ou plusieurs octets, jusqu'à 4 Go

2.2.4 - Types supplémentaires

SYS.ANYTYPE, SYS.ANYDATA

Une variable de ce type peut contenir un objet de n'importe quel type scalaire ou objet

Définie comme colonne d'une table, elle pourrait contenir une variable de type NUMBER dans une ligne, une variable de type VARCHAR2 dans une autre, une variable de type objet dans une troisième, etc.

Il faut utiliser les méthodes associées pour insérer la valeur correspondant au type désiré sur chaque ligne

SYS.ANYDATA.CONVERT...

SYS.ANYDATA.ConvertNumber(1500) pour insérer une variable numérique

SYS.ANYDATA.ConvertVarchar2('Hello') pour insérer une variable caractère

Liste des fonctions de conversion

- ConvertNumber(num IN NUMBER) RETURN AnyData
- ConvertDate(dat IN DATE) RETURN AnyData
- ConvertChar(c IN CHAR) RETURN AnyData
- ConvertVarchar(c IN VARCHAR) RETURN AnyData
- ConvertVarchar2(c IN VARCHAR2) RETURN AnyData
- ConvertRaw(r IN RAW) RETURN AnyData
- ConvertBlob(b IN BLOB) RETURN AnyData
- ConvertClob(c IN CLOB) RETURN AnyData
- ConvertBfile(b IN BFILE) RETURN AnyData
- ConvertObject(obj IN "(object_type)") RETURN AnyData
- ConvertRef(rf IN REF "(object_type)") RETURN AnyData
- ConvertCollection(col IN "(COLLECTION_1)") RETURN AnyData

Et les méthodes suivantes pour retrouver les valeurs insérées

nom_variable.GET...

- GetNumber(self IN AnyData, num OUT NOCOPY NUMBER) RETURN PLS_INTEGER
- GetDate(self IN AnyData, dat OUT NOCOPY DATE) RETURN PLS_INTEGER
- GetChar(self IN AnyData, c OUT NOCOPY CHAR) RETURN PLS_INTEGER
- GetVarchar(self IN AnyData, c OUT NOCOPY VARCHAR) RETURN PLS_INTEGER
- GetVarchar2(self IN AnyData, c OUT NOCOPY VARCHAR2) RETURN PLS_INTEGER
- GetRaw(self IN AnyData, r OUT NOCOPY RAW) RETURN PLS_INTEGER
- GetBlob(self IN AnyData, b OUT NOCOPY BLOB) RETURN PLS_INTEGER
- GetClob(self IN AnyData, c OUT NOCOPY CLOB) RETURN PLS_INTEGER
- GetBfile(self IN AnyData, b OUT NOCOPY BFILE) RETURN PLS_INTEGER
- GetObject(self IN AnyData, obj OUT NOCOPY "(object_type)") RETURN PLS_INTEGER
- GetRef(self IN AnyData, rf OUT NOCOPY REF "(object_type)") RETURN PLS_INTEGER
- GetCollection(self IN AnyData, col OUT NOCOPY "(collection_type)") RETURN PLS_INTEGER

Le type ANYDATA supporte également les méthodes suivantes:

- Procédure BEGINCREATE pour la création d'un nouveau type
- Procédure membre PIECEWISE pour définir le mode d'accès à la valeur courante
- Procédure membre SET... Pour positionner les valeurs
- Procédure membre ENDCREATE Pour terminer la création d'un nouveau type
- Fonction membre GETTYPENAME Pour retrouver la définition complète du type
- Fonction membre GETTYPE Pour retrouver le type de l'objet

SYS.ANYDATASET

Ce type contient à la fois la description et un ensemble de données de même type.

Liste des fonctions attachées à ce type

- Procédure membre ADDINSTANCE Pour l'ajout d'une nouvelle instance de données
- Procédure BEGINCREATE pour la création d'un nouveau type
- Procédure membre PIECEWISE pour définir le mode d'accès à la valeur courante
- Procédure membre SET... Pour positionner les valeurs
- Procédure membre ENDCREATE Pour terminer la création d'un nouveau type
- Fonction membre GETTYPENAME Pour retrouver la définition complète du type
- Fonction membre GETTYPE Pour retrouver le type de l'objet
- Fonction membre GETINSTANCE Pour retrouver l'instance suivante
- Fonctions membre GET... Pour retrouver les valeurs
- Fonction membre GETCOUNT Pour retrouver le nombre d'instances du type

Types XML

Ces types sont utilisés pour stocker des objets XML

Le type XMLTYPE posséde des fonctions membres pour insérer, extraire et interroger les données XML via les expressions de type XPATH

Pour manipuler les données XML, Oracle met à disposition les fonctions

- XMLAGG
- XMLCOLATTVAL
- XMLCONCAT
- XMLDATA
- XMLELEMENT
- XMLFOREST
- XMLSEQUENCE
- XMLTRANSFORM

Ainsi que les paquetages

- DBMS_XMLDOM
- DBMS_XMLGEN
- DBMS_XMLPARSER
- DBMS_XMLQUERY
- DBMS_XMLSAVE
- DBMS_XMLSCHEMA

Les types URI (URITYPE, DBURITYPE, XDBURITYPE et HTTPURITYPE) permettent de gérer les données sous forme d'URL.

Pour manipuler les données XML, Oracle met à disposition le paquetage URIFACTORY

Type Spatial

MDSYS.SDO_GEOMETRY

Pour la manipulation d'objets Oracle Spatial

Types MEDIA

Ces types sont utilisés pour stocker des objets multi-média avec Oracle interMedia

ORDSYS.ORDAUDIO

Pour le stockage de données audio

ORDSYS.ORDIMAGE

Pour le stockage des images

ORDSYS.ORDIMAGESIGNATURE

Pour le stockage des propriétés des images

ORDSYS.ORDVIDEO

Pour le stockage des données vidéo

ORDSYS.ORDDOC

Pour le stockage de tout type de données multi-média

2.3 - Les Types et Sous-types définis par l'utilisateur

En plus des types prédéfinis, l'utilisateur peut définir ses propres types avec le mot clé **TYPE** ou **SUBTYPE**

- **SUBTYPE nom_sous-type IS type_base[(précision)] [NOT NULL]**

nom_sous-type représente le nom du sous-type déclaré

type_base représente le nom du type prédéfini

précision représente une longueur pour les caractères et longueur + décimales pour les numériques

SUBTYPE chaine_courte IS VARCHAR2(10);

le sous-type utilisateur chaine_courte définit un VARCHAR2(10)

à la suite de cette définition, toute déclaration de variable de type chaine_courte sera égale à VARCHAR2(10)

SUBTYPE chaine IS VARCHAR2(100);

le sous-type utilisateur chaine définit un VARCHAR2(100)

SUBTYPE chaine_longue IS VARCHAR2(1000);

le sous-type utilisateur chaine_longue définit un VARCHAR2(1000)

SUBTYPE NOM_EMP IS EMP.ename%Type;

le sous-type NOM_EMP définit un type identique à la colonne ename de la table EMP

SUBTYPE REC_EMP IS EMP%ROWTYPE;

le sous-type REC_EMP définit un type identique à une ligne de la table EMP

TYPE tableau_numerique IS TABLE OF NUMBER;

le type tableau_numerique définit un tableau de NUMBER

TYPE TAB_REC_EMP IS TABLE OF REC_EMP;

le type TAB_REC_EMP définit un tableau d'éléments de type REC_EMP

Lorsque vos nouveaux types et sous-types sont déclarés, vous pouvez les utiliser pour typer de nouvelles variables

```
SQL> Declare
  2    -- définition des types et sous-types
  3    SUBTYPE chaine_courte IS VARCHAR2(10);
  4    SUBTYPE chaine IS VARCHAR2(100);
  5    SUBTYPE chaine_longue IS VARCHAR2(1000);
  6    TYPE tableau_numerique IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
  7    SUBTYPE NOM_EMP IS EMP.ename%Type;
  8    SUBTYPE REC_EMP IS EMP%ROWTYPE;
  9    TYPE TAB_REC_EMP IS TABLE OF REC_EMP;
 10    -- définition des variables
 11    LC$cc chaine_courte;
 12    LC$cl chaine_longue;
 13    Tab   tableau_numerique;
 14  Begin
 15    LC$cc := 'Court' ;
 16    LC$cl := 'Chaîne beaucoup plus longue' ;
 17    dbms_output.put_line( 'Chaîne longue = ' || LC$cl ) ;
 18    For i in 1..5 Loop
 19       Tab(i) := i + (.1 * i ) ;
 20    dbms_output.put_line( 'Tab(' || Ltrim( To_char( i ) ) || ') = ' || To_char( Tab(i) ) ) ;
 21    End loop ;
```

```
 22  End ;
 23  /
Chaîne longue = Chaîne beaucoup plus longue
Tab(1) = 1,1
Tab(2) = 2,2
Tab(3) = 3,3
Tab(4) = 4,4
Tab(5) = 5,5

Procédure PL/SQL terminée avec succès.
```

2.4 - Les littéraux

Un littéral ou valeur constante désigne une valeur fixe.

Par exemple 'LUNDI', 'Montpellier', '2012' représentent des valeurs littérales de type caractère.

12.3, 25 représentent des valeurs littérales de type numérique

Ces valeurs peuvent apparaître dans des initialisations de variables, des calculs ou transmises à des procédures ou fonctions.

2.4.1 - Littéral de type caractère

Désigne une valeur fixe comme étant de type caractère

La valeur peut contenir n'importe quel caractère à l'exception d'une simple apostrophe

(pour saisir une apostrophe dans un littéral, il faut la doubler (''))

Il doit être encadré d'une paire d'apostrophes

Il peut être précédé du caractère N pour indiquer qu'il doit être transformé dans le jeu de caractères national

Il a les mêmes propriétés que les types CHAR et VARCHAR2

Sa longueur ne peut pas dépasser 4000 octets

'J''aime le PL/SQL'

'Cordialement'

'Select * From EMP'

2.4.2 - Littéral de type entier

Désigne une valeur fixe comme étant de type entier

Ne peut contenir que les chiffres de 0 à 9

Il peut être précédé des signes + ou -

Il peut contenir jusqu'à 38 chiffres de précision

-12

267589

+3

2.4.3 - Littéral de type décimal

Désigne une valeur fixe comme étant de type numérique

Ne peut contenir que les chiffres de 0 à 9

Il peut être précédé des signes + ou -

Il peut contenir jusqu'à 38 chiffres de précision

Il peut contenir le caractère e ou E qui indique que la valeur est spécifiée en notation scientifique. Les chiffres après le E indiquent l'exposant. Ce dernier est valide dans un intervalle de -130 à 125

-54

+3.1415

0.006

75E-12

2.4.4 - Littéral de type intervalle (9i)

Nouveauté 9i Spécifie une période de temps, déclinée en années et mois ou en jours, heures, minutes et secondes.

Les deux types de littéraux de type intervalle sont YEAR TO MONTH et DAY TO SECOND

Chaque type contient un préfixe et peut contenir un suffixe. Le préfixe désigne l'unité de base de date ou d'heure. Le suffixe définit les parties d'incrément associées à l'unité de base.

Si vos données sont sous forme numérique, vous pouvez utiliser les fonctions de conversion NUMTOYMINTERVAL ou NUMTODSINTERVAL pour les convertir en littéraux de type intervalle.

INTERVAL 'nombre_entier [-nombre_entier]' YEAR ou MONTH (précision) TO YEAR ou MONTH

Nombre_entier [-nombre_entier] spécifie une valeur entière pour le préfixe et éventuellement le suffixe du littéral. Si le préfixe est YEAR et le suffixe est MONTH, nombre_entier pour le mois doit être entre 0 et 11

Précision représente le nombre maximum de chiffres pour le préfixe compris entre 0 et 9. Par défaut sa valeur est 2

INTERVAL '12-3' YEAR TO MONTH : intervalle de 12 ans et 3 mois

INTERVAL '115' YEAR(3) : intervalle de 115 ans (la précision du suffixe doit être spécifiée YEAR(3) si elle est supérieure à la valeur par défaut)

INTERVAL '24' MONTH : intervalle de 24 mois

Il est possible d'additionner ou soustraire un littéral de type intervalle à un autre

INTERVAL '6-4' YEAR TO MONTH - INTERVAL '6' MONTH

- **Intervalle de type DAY TO SECOND**

INTERVAL 'nombre_entier' DAY ou HOUR ou MINUTE ou SECOND (précision) TO DAY ou HOUR ou MINUTE ou SECOND (fractions de secondes)

Nombre_entier peut représenter soit :

Un nombre de jours

Une heure au format HH[:MI[:SS[.fractions_de_secondes]]]

Précision représente le nombre de chiffres du préfixe, compris entre 0 et 9. Par défaut sa valeur est 2

Fractions_de_secondes représente le nombre de chiffres des fractions de secondes, compris entre 1 et 9. Par défaut sa valeur est 6

Les valeurs correctes pour les champs sont :

HOUR 0 à 23

MINUTE 0 à 59

SECOND 0 à 59.999999999

INTERVAL '6 4 :10 :22.356' DAY TO SECOND(3) : intervalle de 6 jours, 4 heures, 10 minutes, 22 secondes et 356 millièmes de secondes

INTERVAL '6 4 :10' DAY TO MINUTE : intervalle de 6 jours, 4 heures et 10 minutes

INTERVAL '365 12' DAY(3) TO HOUR : intervalle de 365 jours et 12 heures

INTERVAL '8 :10 :20.3333333' HOUR TO SECOND(7) : Intervalle de 8 heures, 10 minutes, 20.3333333 secondes

INTERVAL '18 :30' HOUR TO MINUTE : intervalle de 18 heures et 30 minutes

INTERVAL '20' MINUTE : intervalle de 20 minutes

INTERVAL '4.12345' SECOND(2,4) : intervalle arrondi à 4.1235 secondes car la précision demandée sur les fractions de secondes est de 4 chiffres

Il est possible d'additionner ou soustraire un littéral de type intervalle à un autre

INTERVAL '30' DAY - INTERVAL '18' HOUR

3 - Les fonctions natives

Ces fonctions SQL ne sont pas propres au PL/SQL et ne sont donc pas développées outre mesure dans ce chapitre,

notamment au niveau des formats et littéraux utilisables.

Reportez-vous à la documentation SQL pour davantage de précisions

3.1 - Les fonctions chaînes de caractères

2 syntaxes :

chaîne || chaîne

CONCAT(chaîne, chaîne)

Ces fonctions formatent une chaîne sur une longueur donnée par ajout de caractères avant (LPAD) ou après (RPAD) la chaîne passée en argument

LPAD(chaîne, longueur [, 'caractères'])

RPAD(chaîne, longueur [, 'caractères'])

chaîne représente le nom d'une colonne, d'une variable ou un littéral

longueur représente le nombre total de caractères du résultat

caractères représente le ou les caractères de remplissage

```
RPAD( 'Total', 20, '.' )
Total...............

LPAD( 'Hello', 20, '.' )
...............Hello

RPAD( LPAD( '2320 euros', 20, '*' ), 30, '*' )
**********2320 euros**********

LPAD( 'Hello', 20, '-*-' )
```

```
-*--*--*--*--*-Hello
```

Si caractères n'est pas spécifié, le caractère par défaut est l'espace (CHR(32))

LTRIM(chaîne [, 'caractères'])

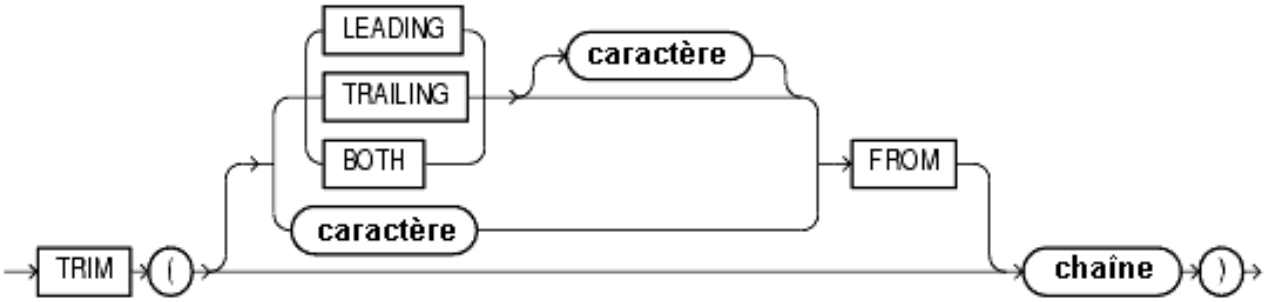
RTRIM(chaîne [, 'caractères'])

```
LTRIM( '"Libellé', '"' )
Libellé

RTRIM( 'Libellé----', '-' )
Libellé

LTRIM( RTRIM( '"Libellé"', '"' ), '"' )
Libellé
```

si caractères n'est pas spécifié, le caractère par défaut est l'espace (CHR(32))

```
RTRIM( '(Libellé     ') || ')'
(Libellé)
```

TRIM



Cette fonction permet de cumuler les possibilités de LTRIM et RTRIM

chaîne représente la chaîne de caractères en entrée de la fonction

si **LEADING** est spécifié, tous les premiers caractères identiques à **caractère** sont supprimés

si **TRAILING** est spécifié, tous les derniers caractères identiques à **caractère** sont supprimés

si **BOTH** ou rien est spécifié, tous les premiers et derniers caractères identiques à **caractère** sont supprimés

si caractère n'est pas spécifié la valeur par défaut est l'espace

si seule **chaîne** est spécifiée tous les espaces en début et fin de chaîne sont supprimés

si **caractère** ou **chaîne** est NULL, la fonction retourne NULL

```
SQL> Declare
  2     LC$Ch1 Varchar(20) := '   libellé   ' ;
  3     LC$Ch2 Varchar(20) := '***libellé***' ;
  4  Begin
  5     dbms_output.put_line( TRIM( LC$Ch1 ) ) ;
  6     dbms_output.put_line( TRIM( '*' FROM LC$Ch2 ) ) ;
  7  End ;
  8  /
libellé
libellé

Procédure PL/SQL terminée avec succès.
```

LOWER(chaîne)

NLS_LOWER(chaîne [, nls_paramètre])

UPPER(chaîne)

NLS_UPPER(chaîne [, nls_paramètre])

INITCAP(chaîne)

NLS_INITCAP(chaîne [, nls_paramètre])

```
-- Conversion d'une chaîne en minuscules
LOWER( 'ORACLE' )
oracle

-- Conversion d'une chaîne en majuscules
UPPER( 'oracle' )
ORACLE

-- Conversion d'une chaîne avec la première lettre de chaque mot en majuscule
INITCAP( 'le sgbd oracle' )
Le Sgbd Oracle
```

LENGTH(chaîne)

```
LENGTH( 'le sgbd oracle' )
14
```

SUBSTR(chaîne, début [, 'nombre'])

chaîne représente le nom d'une colonne, d'une variable ou un littéral

début représente la position de départ de recherche dans la chaîne

nombre représente le nombre de caractères à extraire

```
-- extraction de 4 caractères à partir du 4ème caractère
SUBSTR( 'le sgbd oracle', 4, 4 )
sgbd
```

si **nombre** est omis, la fonction ramène tous les caractères à partir de la position début

```
SUBSTR( 'le sgbd oracle', 4 )
sgbd oracle
```

si **nombre** est négatif l'extraction débute à partir de la fin de la chaîne

```
SUBSTR( 'le sgbd oracle', -6 )
oracle
```

INSTR(chaîne, sous-chaîne [, début [,nombre occurrences]])

INSTRB(chaîne, sous-chaîne [, début [,nombre occurrences]])

chaîne représente le nom d'une colonne, d'une variable ou un littéral passé en argument

sous-chaîne représente le nom d'une variable ou un littéral dont on cherche la position

début (optionnel) représente la position de départ de la recherche dans chaîne

nombre occurrences (si début renseigné) représente le nombre d'occurrences trouvées à ignorer

Lorsque **sous-chaîne** représente plusieurs caractères, la fonction retourne la position du premier caractère de la sous-chaîne

recherche de la première position de la sous-chaîne oracle

```
INSTR( 'le sgbd oracle', 'oracle' )
9
```

recherche de la deuxième position de la sous-chaîne oracle

```
INSTR( 'le sgbd oracle d''oracle corporation', 'oracle', 1, 2 )
18
```

si la sous-chaîne n'est pas trouvée, la fonction retourne 0

```
INSTR( 'le sgbd oracle d''oracle corporation', 'texte', 1, 2 )
0
```

REPLACE(chaîne, chaîne source, chaîne cible)

chaîne représente le nom d'une colonne, d'une variable ou un littéral passé en argument

chaîne source représente le nom d'une variable ou un littéral de recherche

chaîne cible représente le nom d'une variable ou un littéral de remplacement

```
REPLACE( 'banjaur', 'a', 'o' )
bonjour
```

Elimination de caractères

```
REPLACE( '"champs1","champs2"', '"', '' )
champs1,champs2
```

TRANSLATE(chaîne, chaîne source, chaîne cible)

chaîne représente le nom d'une colonne, d'une variable ou un littéral passé en argument

chaîne source représente le nom d'une variable ou un littéral de recherche

chaîne cible représente le nom d'une variable ou un littéral de remplacement

Chaque caractère présent dans **chaîne** qui est également présent dans **chaîne source** est remplacé

par le caractère qui occupe la même position dans **chaîne cible**

```
TRANSLATE( 'Pas d''accents : éèàùôö', 'éèàùôö', 'eeauoo' )
Pas d'accents : eeauoo
```

Dans l'exemple suivant, le caractère _ est remplacé par _ et tous les autres (interdit sous Unix)

sont également remplacés par _

```
TRANSLATE( 'Nom*de[fichier<unix>', '_ /\<>|(){}[]*&"'''$;','_____________________' )
Nom_de_fichier_unix_
```

Elimination de caractères indésirables

```
TRANSLATE( 'Nom+de|fichier!unix', 'A+|!', 'A' )
Nomdefichierunix
```

Le premier caractère A de la chaîne source est un leurre qui indique à la fonction de remplacer

toutes les occurrences de A par A et de remplacer les autres caractères (+|!) par rien.

3.2 - Les fonctions arithmétiques

ABS(valeur)

Valeur absolue d'un nombre

ABS(200) = 200

ABS(-200) = 200

CEIL(valeur)

Entier supérieur ou égal à **valeur**

CEIL(5) = 5

CEIL(5.1) = 6

CEIL(-5) = -5

CEIL(-5.1) = -5

CEIL(-5.9) = -5

FLOOR(valeur)

Entier inférieur ou égal à **valeur**

FLOOR(5) = 5

FLOOR(5.1) = 5

FLOOR(-5) = -5

FLOOR(-5.1) = -6

FLOOR(-5.9) = -6

MOD(valeur, diviseur)

Reste d'une division

MOD(10, 2) = 0

MOD(10, 3) = 1

MOD(10, .6) = 0.4

MOD(-10, 2) = 0

MOD(-10, .6) = -0.4

Lorsque **diviseur** est supérieur à **valeur**, la fonction retourne la valeur

MOD(5, 12) = 5

Si **valeur** est un entier alors MOD(valeur, 1) = 0 (idéal pour tester que valeur est entier)

Si **valeur** est un entier pair alors MOD(valeur, 2) = 0

POWER(valeur, exposant)

Elévation d'un nombre à une puissance

POWER(8, 2) = 64

POWER(8, -2) = 0.015625

SQRT(valeur)

Racine carrée d'un nombre

SQRT(64) = 8

EXP(valeur)

e (2,71828182845905) élevé à une puissance

EXP(5) = 148.413159102577

LN(valeur)

Logarithme naturel, ou base e, d'une valeur

LN(148.413159102577) = 5

LOG(base, valeur)

Logarithme d'une valeur

LOG(10, 2) = 0.301029995663981

LN(valeur) est identique à LOG(2,71828182845905 (EXP(1)), valeur)

ROUND(valeur, précision)

Arrondi d'une valeur à un certain nombre de chiffres de précision

ROUND(9.254, 1)

9,3

ROUND(9.259, 2)

9,26

ROUND(9.258, 0)

9

ROUND(9.9, 0)

10

ROUND(99.259, -1)

100

TRUNC(valeur, précision)

Suppression d'une partie des chiffres de précision décimale

TRUNC(9.259, 2)

9,25

TRUNC(9.9, 0)

9

TRUNC(99.259, -1)

90

une précision négative est intéressante pour arrondir des nombres sur des tranches de milliers, millions, milliards, etc.

SIGN(valeur)

Signe d'une valeur

SIGN(12)

1

SIGN(-12)

-1

SIN(valeur), COS(valeur), TAN(valeur)

Renvoie la valeur trigonométrique d'un angle exprimé en radians

SINH(valeur), COSH(valeur), TANH(valeur)

Renvoie la valeur trigonométrique hyperbolique d'un angle exprimée en radians

ASIN(valeur), ACOS(valeur), ATAN(valeur)

Renvoie respectivement l'arc sinus, cosinus et tangente en radians

Ces fonctions portent sur des groupes de lignes et sont utilisées dans les ordres select

(Bien sûr rien n'empêche de coder : AVG(40), mais cela n'a pas beaucoup de sens)

AVG(colonne)

Valeur moyenne des valeurs de **colonne**

(les valeurs NULL ne sont pas prises en compte)

select AVG(SAL) from EMP

AVG(SAL)

2073,21429

COUNT(colonne)

Nombre de valeurs de **colonne** (les valeurs NULL ne sont pas prises en compte)

select COUNT(ENAME) from EMP;

COUNT(ENAME)

14

MAX(colonne)

Valeur maximum des valeurs de **colonne**

select ENAME, SAL from emp where SAL = (select MAX(SAL) From EMP);

ENAME SAL

KING 5000

MIN(colonne)

Valeur minimum des valeurs de **colonne**

Select MIN(SAL) From EMP

SUM(colonne)

Somme des valeurs de colonne

Select SUM(SAL) From EMP

Ces fonctions portent sur un ensemble de colonnes d'une seule ligne

Elles peuvent être utilisées avec de nombreuses valeurs qui peuvent apparaître sous forme de colonnes, de littéraux, de calculs ou de combinaison d'autres colonnes.

GREATEST(valeur1, valeur2, valeur3,#)

Valeur la plus grande de la liste

GREATEST(1, 5, 15, 8)

15

GREATEST('Elmer', 'Achille','Richard', 'Josianne')

Richard

GREATEST(Length('Elmer'), Length('Achille'),Length('Richard'), Length('Josianne'))

8

LEAST(valeur1, valeur2, valeur3,#)

Valeur la plus petite de la liste

LEAST(1, 5, 15, 8)

1

LEAST(3, 5+5, 8, 12-6)

3

3.3 - Les fonctions de conversion et de transformation

BIN_TO_NUM(bit [,bit[...]])

Conversion d'une suite de bits en nombre

```
SQL> SELECT BIN_TO_NUM(1,0,0,0,0,0,0,0,0,0,0,0) FROM DUAL ;

BIN_TO_NUM(1,0,0,0,0,0,0,0,0,0,0,0)
-----------------------------------
                               4096

SQL>
```

BITAND(arg1, arg2)

Applique un ET logique sur les deux arguments

```
SELECT order_id, customer_id,
  DECODE(BITAND(order_status, 1), 1, 'Warehouse', 'PostOffice') Location,
  DECODE(BITAND(order_status, 2), 2, 'Ground', 'Air') Method,
  DECODE(BITAND(order_status, 4), 4, 'Insured', 'Certified') Receipt
  FROM orders
  WHERE order_status < 8;

  ORDER_ID CUSTOMER_ID LOCATION   MET RECEIPT
---------- ----------- ---------- --- ---------
      2458         101 Postoffice Air Certified
      2397         102 Warehouse  Air Certified
      2454         103 Warehouse  Air Certified
      2354         104 Postoffice Air Certified
      2358         105 Postoffice G   Certified
      2440         107 Warehouse  G   Certified
      2357         108 Warehouse  Air Insured
      2435         144 Postoffice G   Insured
...
```

CHARTOROWID(char)

Conversion d'une chaîne en ROWID

CONVERT(chaîne, jeu caract source, jeu caract cible)

Conversion d'une chaîne d'un jeu de caractères à un autre

```
select CONVERT('Ä Ê Í Õ Ø A B C D E ', 'US7ASCII', 'WE8ISO8859P1') from dual ;

CONVERT('ÄÊÍÕØABCDE'
--------------------
A E I ? ? A B C D E
```

DECODE(valeur, si1, alors1, si2, alors2,#.., sinon)

Substitution valeur par valeur

```
select SAL, DECODE( TRUNC( SAL, -3 ), 0, 'Bof', 1000, 'Mieux', 2000, 'Pas mal', 'Super' ) from emp ;

       SAL DECODE(
---------- -------
       800 Bof
      1600 Mieux
      1250 Mieux
      2975 Pas mal
      1250 Mieux
      2850 Pas mal
      2450 Pas mal
      3000 Super
      5000 Super
      1500 Mieux
      1100 Mieux
       950 Bof
      3000 Super
      1300 Mieux
```

BIN_TO_NUM(bit [,bit[...]])

Conversion d'une suite de bits en nombre

```
SQL> SELECT BIN_TO_NUM(1,0,0,0,0,0,0,0,0,0,0,0,0) FROM DUAL ;

BIN_TO_NUM(1,0,0,0,0,0,0,0,0,0,0,0,0)
-------------------------------------
                                 4096

SQL>
```

DUMP(expr [,format_retour [,début [,longueur]]])

Retourne le type interne, longueur en octets de l'expression passée en argument

format_retour peut prendre l'une des quatre valeurs suivantes :

- **8** octal

- **10** décimal
- **16** héxadécimal
- **17** caractères

On peut ajouter 1000 à l'argument format_retour pour forcer DUMP à afficher le jeu de caractères en cours

```
SQL> -- DUMP --
SQL> SELECT DUMP('Hello', 8) "Octal",
  2  DUMP('Hello', 10) "Décimal",
  3  DUMP('Hello', 16) "Hexadécimal",
  4  DUMP('Hello', 17) "Caractère"
  5  FROM DUAL ;

Octal                            Décimal                          Hexadécimal
Caractère
-------------------------------- -------------------------------- ----------------------------
------------------------
Typ=96 Len=5: 110,145,154,154,157 Typ=96 Len=5: 72,101,108,108,111 Typ=96 Len=5: 48,65,6c,6c,6f
Typ=96 Len=5: H,e,l,l,o

SQL>
```

```
SELECT DUMP('abc', 1016)
   FROM DUAL;

DUMP('ABC',1016)
----------------------------------------
Typ=96 Len=3 CharacterSet=WE8DEC: 61,62,63

SQL>
```

HEXTORAW(char)

Conversion d'un nombre hexadécimal en un nombre binaire

RAWTOHEX(raw)

Conversion d'un nombre binaire en nombre hexadécimal

ROWIDTOCHAR(rowid)

Conversion d'un ROWID en chaîne

```
select ROWIDTOCHAR( ROWID ) from emp ;

ROWIDTOCHAR(ROWID)
------------------
AAAHW7AABAAAMUiAAA
AAAHW7AABAAAMUiAAB
AAAHW7AABAAAMUiAAC
AAAHW7AABAAAMUiAAD
AAAHW7AABAAAMUiAAE
AAAHW7AABAAAMUiAAF
```

TO_CHAR(date [, 'format'])

TO_CHAR(nombre [, 'format'])

Transformation d'un type DATE ou NUMBER en chaîne

```
select TO_CHAR( SYSDATE, 'DD/MM/YYYY HH24:MI:SS' ) from dual ;

TO_CHAR(SYSDATE,'DD
-------------------
31/01/2004 19:30:56
```

```
select TO_CHAR( 1256.35, '999G999D000' ) from dual ;

TO_CHAR(1256
------------
   1.256,350
```

TO_DATE(nombre [, 'format'])

TO_DATE(chaîne [, 'format'])

Transformation d'un type NUMBER ou CHAR ou VARCHAR2 en date

```
select TO_DATE( '01/01/04', 'DD/MM/RR' ) from dual ;

TO_DATE('01/01/04',
-------------------
01/01/2004 00:00:00
```

```
Select TO_DATE( 2453036, 'J' ) from dual ;

TO_DATE(2453036,'J'
-------------------
31/01/2004 00:00:00
```

TO_DSINTERVAL(chaîne)(9i)

Transformation d'un type CHAR ou VARCHAR2 en INTERVAL DAY TO SECOND

TO_LOB(LONG)

TO_LOB(LONG RAW)

Conversion d'une valeur de type LONG ou LONG RAW en valeur de type LOB

TO_CLOB(char)

TO_CLOB(colonne lob)

Conversion d'une valeur de type CHAR ou LOB en valeur de type CLOB

TO_MULTI_BYTE

Conversion d'une chaîne de caractères mono-octet en chaîne de caractères multi-octets

Cette fonction n'est nécessaire que si votre jeu de caractères contient à la fois des caractères mono-octets et des caractères multi-octets

TO_NUMBER(chaîne [, 'format'])

Transformation d'un type CHAR ou VARCHAR2 en nombre

```
select TO_NUMBER( '25,8', '9G999D00' ) from dual ;

TO_NUMBER('25,8','9G999D00')
----------------------------
                        25,8
```

TO_SINGLE_BYTE

Conversion d'une chaîne de caractères multi-octets en chaîne de caractères mono-octet

Cette fonction n'est nécessaire que si votre jeu de caractères contient à la fois des caractères mono-octets et des caractères multi-octets

TO_TIMESTAMP(chaîne)(9i)

Transformation d'un type CHAR ou VARCHAR2 en TIMESTAMP

```
select to_timestamp( '01/01/2004 01:00:00' ,'DD/MM/YYYY HH24:MI:SS') from dual ;

TO_TIMESTAMP('01/01/200401:00:00','DD/MM/YYYYHH24:MI:SS')
---------------------------------------------------------------------------
01/01/04 01:00:00,000000000
```

TO_TIMESTAMP_TZ(chaîne)(9i)

Transformation d'un type CHAR ou VARCHAR2 en TIMESTAMP + TIME ZONE

TO_YMINTERVAL(chaîne)(9i)

Transformation d'un type CHAR ou VARCHAR2 en INTERVAL YEAR TO MONTH

Chaîne représente un couple année-mois 'AA-MM'

```
select ename "Salarié", hiredate "Date entrée" , hiredate + TO_YMINTERVAL( '05-01' ) "Ancienneté"
from emp ;

Salarié    Date entrée         Ancienneté
---------- ------------------- -------------------
SMITH      17/12/1980 00:00:00 17/01/1986 00:00:00
ALLEN      20/02/1981 00:00:00 20/03/1986 00:00:00
WARD       22/02/1981 00:00:00 22/03/1986 00:00:00
```

VSIZE(argument)

Retourne le nombre d'octets nécessaires au stockage de l'argument

```
SQL> -- Nombre d'octets de stockage --
SQL> SELECT VSIZE(10),VSIZE(123.9658), VSIZE(123.9658123456789), VSIZE(123456789456) FROM DUAL ;

 VSIZE(10) VSIZE(123.9658) VSIZE(123.9658123456789) VSIZE(123456789456)
---------- --------------- ------------------------ -------------------
         2               5                       10                   7

SQL>
SQL> SELECT VSIZE(SYSDATE), VSIZE(CURRENT_TIMESTAMP) FROM DUAL ;

VSIZE(SYSDATE) VSIZE(CURRENT_TIMESTAMP)
-------------- ------------------------
             7                       13

SQL>
```

3.4 - Les fonctions sur les dates

ADD_MONTHS(date, nombre de mois)

Ajoute ou soustrait un nombre de mois à une date

```
select SYSDATE, ADD_MONTHS( SYSDATE, 1 ) from dual ;

SYSDATE    ADD_MONTHS
---------- ----------
```

```
31/01/2004 29/02/2004
```

```
select SYSDATE, ADD_MONTHS( SYSDATE, -12 ) from dual ;

SYSDATE    ADD_MONTHS
---------- ----------
31/01/2004 31/01/2003
```

EXTRACT (datetemps FROM valeur)(9i)

Extraction d'un segment d'une valeur de type date ou d'un littéral de type intervalle

(Pour une raison inconnue, cette fonction ne permet pas d'extraire l'heure d'une date...)

datetemps peut être l'un des arguments suivants

valeur est l'un des arguments suivants

Extraction de la partie année d'une date :

```
SQL> SELECT EXTRACT(YEAR FROM TO_DATE('10/01/2004','DD/MM/YYYY')) FROM DUAL;

EXTRACT(YEARFROMTO_DATE('10/01/2004','DD/MM/YYYY'))
---------------------------------------------------
                                               2004
```

Affichage des employés dont la date d'entée est supérieure à 1981

```
SQL>  SELECT empno "Numéro", ename "Nom", to_char(hiredate,'DD/MM/YYYY') "Entré le"
  2      FROM EMP
  3      WHERE EXTRACT(YEAR FROM
  4      TO_DATE(hiredate, 'DD/MM/RR')) > 1981
  5     ORDER BY hiredate
  6  /

    Numéro Nom        Entré le
---------- ---------- ----------
      7934 MILLER     24/01/1982
      7788 SCOTT      20/04/1987
      7876 ADAMS      24/05/1987
```

LAST_DAY(date)

Dernier jour du mois de la date passée en argument

```
select LAST_DAY( '01/01/2004' ) from dual ;

LAST_DAY('
```

```
----------
31/01/2004
```

MONTHS_BETWEEN(date2, date1)

Nombre de mois qui séparent **date2** de **date1**

```
select MONTHS_BETWEEN( '01/10/2004', '01/01/2004' ) from dual ;

MONTHS_BETWEEN('01/10/2004','01/01/2004')
-----------------------------------------
                                        9
```

NEXT_DAY(date, 'jour')

Date du prochain jour après **date** ou **jour** est un jour de la semaine

```
select NEXT_DAY( '31/01/2004', 'Lundi' ) from dual ;

NEXT_DAY('
----------
02/02/2004
```

NEW_TIME(date, fuseau1, fuseau2)

Date et heure de **date** en time zone **fuseau2** lorsque **date** est en time zone **fuseau1**

```
select sysdate, NEW_TIME( sysdate, 'AST', 'PST' ) from dual ;

SYSDATE             NEW_TIME(SYSDATE,'A
------------------- -------------------
31/01/2004 18:15:20 31/01/2004 14:15:20
```

NUMTODSINTERVAL(nombre, 'type')(9i)

Conversion d'un nombre en intervalle de type DAY TO SECOND

'type' peut prendre l'une des quatre valeurs suivantes :

- **'DAY'** précise que nombre indique un nombre de jours
- **'HOUR'** précise que nombre indique un nombre d'heures
- **'MINUTE'** précise que nombre indique un nombre de minutes
- **'SECOND'** précise que nombre indique un nombre de secondes

```
SQL> -- Ajout de 8 minutes à une date --
SQL> SELECT SYSDATE, SYSDATE + NUMTODSINTERVAL(8,'MINUTE') FROM DUAL ;
```

```
SYSDATE             SYSDATE+NUMTODSINTE
------------------- -------------------
25/11/2004 12:18:23 25/11/2004 12:26:23

SQL>
SQL> -- Ajout de 8 heures à une date --
SQL> SELECT SYSDATE, SYSDATE + NUMTODSINTERVAL(8,'HOUR') FROM DUAL ;

SYSDATE             SYSDATE+NUMTODSINTE
------------------- -------------------
25/11/2004 12:18:23 25/11/2004 20:18:23

SQL>
SQL> -- ajout de 30 secondes à une date --
SQL> SELECT SYSDATE, SYSDATE + NUMTODSINTERVAL(30,'SECOND') FROM DUAL ;

SYSDATE             SYSDATE+NUMTODSINTE
------------------- -------------------
25/11/2004 12:18:23 25/11/2004 12:18:53

SQL>
```

NUMTOYMINTERVAL(nombre, 'type')(9i)

Conversion d'un nombre en intervalle de type YEAR TO MONTH

'type' peut prendre l'une des deux valeurs suivantes :

- **'YEAR'** précise que nombre indique un nombre d'années
- **'MONTH'** précise que nombre indique un nombre de mois

```
SQL> -- Ajout de 6 mois à une date --
SQL> SELECT SYSDATE + NUMTOYMINTERVAL(6,'MONTH') FROM DUAL ;

SYSDATE+NUMTOYMINTE
-------------------
25/05/2005 12:18:23

SQL>
SQL> -- Ajout de 8 ans à une date --
SQL> SELECT SYSDATE + NUMTOYMINTERVAL(8,'YEAR') FROM DUAL ;

SYSDATE+NUMTOYMINTE
-------------------
25/11/2012 12:18:23

SQL>
```

ROUND(date [, format])

```
select ROUND( to_date('01/01/2004 10:25','DD/MM/YYYY HH24:MI') ) from dual
SQL> /

ROUND(TO_DATE('01/0
-------------------
01/01/2004 00:00:00
```

```
select ROUND( to_date('01/01/2004 18:25','DD/MM/YYYY HH24:MI') ) from dual
SQL> /
```

```
ROUND(TO_DATE('01/0
------------------
02/01/2004 00:00:00
```

sans l'argument **format**, la date est arrondie au matin 00:00:00 si l'heure transmise dans date est située avant 12:00, sinon elle est arrondie au lendemain 00:00:00

TRUNC(date [, format])

```
select TRUNC( to_date('01/01/2004 18:25','DD/MM/YYYY HH24:MI') ) from dual ;

TRUNC(TO_DATE('01/0
------------------
01/01/2004 00:00:00
```

```
select TRUNC( to_date('01/01/2004 10:25','DD/MM/YYYY HH24:MI') ) from dual ;

TRUNC(TO_DATE('01/0
------------------
01/01/2004 00:00:00
```

sans l'argument **format**, la date est arrondie au matin 00:00:00

SYS_EXTRACT_UTC(timestamp)(9i)

Conversion d'une date au fuseau Greenwich

```
SQL> -- Ramener une date/heure au format Greenwich --
SQL> SELECT SYSDATE, SYS_EXTRACT_UTC(SYSTIMESTAMP) FROM DUAL ;

SYSDATE             SYS_EXTRACT_UTC(SYSTIMESTAMP)
------------------- ---------------------------------------------------------
25/11/2004 12:18:26 25/11/04 11:18:26,399272

SQL>
```

4 - Procédures, Fonctions et paquetages

Une procédure est un ensemble de code PL/SQL nommé, défini par l'utilisateur et généralement stocké dans la BDD

Une fonction est identique à une procédure à la différence qu'elle retourne une valeur

Un paquetage est le regroupement de plusieurs procédures et fonctions dans un objet distinct

Ces ensembles nommés sont stockés dans la base de données, offrant les avantages suivants :

- Le code relatif aux règles de gestion est centralisé. Cela permet de dissocier les fonctions au sein d'une équipe

  La partie traitement des règles de gestion est confiée à une partie de l'équipe et la conception des interfaces

  est confiée à l'autre partie
- Ces traitements stockés sont donc déportés des interfaces clientes, permettant le partage du code entre plusieurs applications

  ainsi qu'une amélioration des performances, car le code stocké est pré-compilé
- Ces traitements sont accessibles par toute application tierce supportant l'appel des procédures stockées (Sql*Plus, Forms, Reports, Pro*C, Pro*Cobol, etc.)
- Cela permet également de tirer parti de la réutilisation des requêtes dans la base qui se trouvent dans le pool partagé de la zone SGA(System Global Area)

Pour créer un objet procédural, vous devez disposer du privilège système CREATE PROCEDURE pour votre schéma ou du privilège système CREATE ANY PROCEDURE pour la création dans un autre schéma

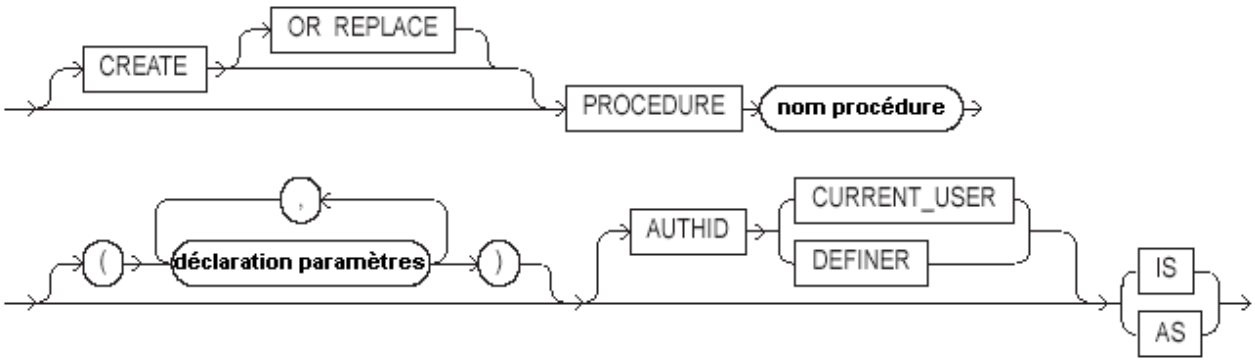
Pour autoriser un autre schéma à exécuter une procédure de votre schéma, vous devez lui octroyer le privilège EXECUTE

GRANT EXECUTE ON ma_procedure TO autre_schéma

4.1 - Les Procédures

Une procédure est un ensemble de code PL/SQL nommé, défini par l'utilisateur et généralement stocké dans la BDD

Une procédure est paramétrable afin d'en faciliter la réutilisation



CREATE indique que l'on veut créer une procédure stockée dans la base

La clause facultative **OR REPLACE** permet d'écraser une procédure existante portant le même nom

nom procédure est le nom donné par l'utilisateur à la procédure

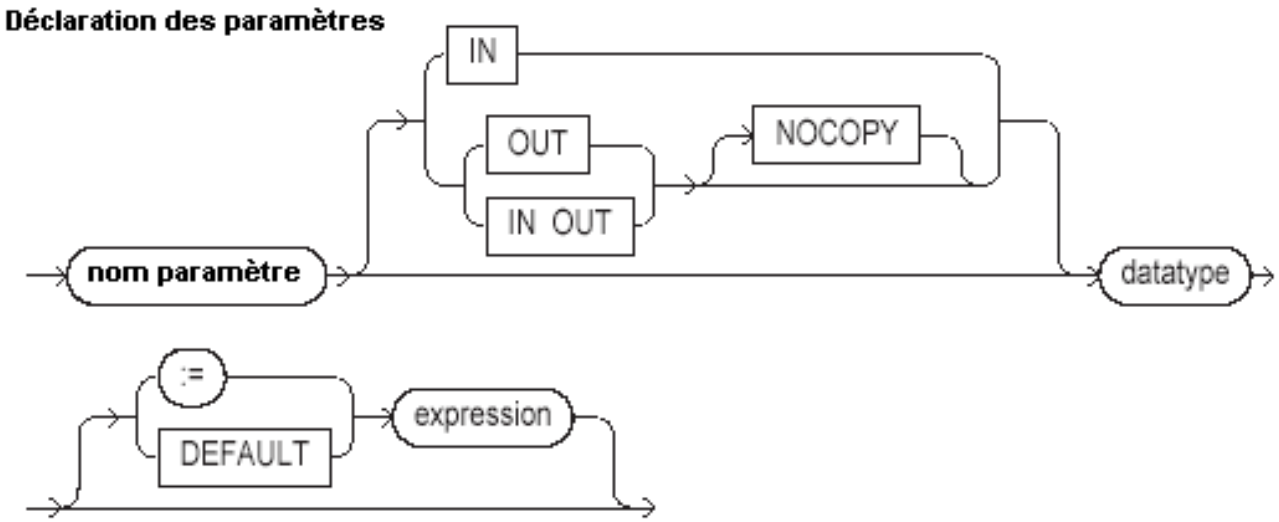
AUTHID indique sur quel schéma la procédure s'applique :

- **CURRENT_USER**

Indique que la procédure utilise les objets du schéma de l'utilisateur qui appelle la procédure

- **DEFINER**(défaut)

Indique que la procédure utilise les objets du schéma de création de la procédure



nom paramètre est le nom donné par l'utilisateur au paramètre transmis

IN(valeur par défaut) indique que le paramètre transmis par le programme appelant n'est pas modifiable par la procédure

OUT indique que le paramètre est modifiable par la procédure

IN OUT indique que le paramètre est transmis par le programme appelant et renseigné par la procédure

NOCOPY indique que le paramètre est transmis par référence (pointeur) et non par copie de la valeur

Par défaut, les paramètres sont transmis par copie, c'est à dire qu'un espace mémoire est créé pour recevoir une copie de la valeur

avec la clause NOCOPY, aucun espace mémoire supplémentaire n'est créé, c'est donc l'adresse de l'espace mémoire initial qui est transmise, permettant d'une part de ne pas gaspiller la mémoire disponible (surtout lorsqu'il s'agit de grands objets (LOB) et également d'éviter le temps nécessaire à la gestion de ces nouveaux espace mémoire (empilement, dépilement, etc.)

datatype représente le type SQL ou PL/SQL du paramètre

:= représente le symbole d'assignation d'une valeur par défaut

DEFAULT identique à :=

expression représente la valeur par défaut du paramètre (doit être conforme au type du paramètre)

Créons une procédure permettant d'augmenter le salaire d'un employé

```
SQL> CREATE OR REPLACE PROCEDURE Augmentation
  2   (
  3     PN$Numemp IN EMP.empno%Type -- numéro de l'employé
  4    ,PN$Pourcent IN NUMBER       -- pourcentage d'augmentation
  5   ) IS
  6  BEGIN
  7    -- augmentation de l'employé
  8    Update EMP Set sal = sal * PN$Pourcent
  9    Where empno = PN$Numemp ;
 10  END;
 11  /

Procédure créée.
```

La procédure Augmentation reçoit deux paramètres

PN$Numemp en entrée (IN) de même type que la colonne empno de la table EMP qui reçoit le numéro d'employé

PN$Pourcent en entrée (IN) de type NUMBER qui reçoit le pourcentage d'augmentation

Faisons maintenant appel à cette procédure dans un bloc PL/SQL anonyme

```
SQL> Declare
  2    LR$Emp EMP%Rowtype ;
  3  Begin
  4    Select * Into LR$Emp From EMP Where empno = 7369 ; -- lecture ligne avant mise à jour
  5    dbms_output.put_line( 'Avant augmentation ' || To_char( LR$Emp.empno )
  6      || ' ' || LR$EMp.ename || ' --> ' || To_char( LR$Emp.sal ) ) ;
  7    Augmentation( 7369, 1.1 ) ; -- appel de la procédure
  8    Select * Into LR$Emp From EMP Where empno = 7369 ; -- lecture ligne après mise à jour
  9    dbms_output.put_line( 'Après augmentation ' || To_char( LR$Emp.empno )
 10      || ' ' || LR$EMp.ename || ' --> ' || To_char( LR$Emp.sal ) ) ;
 11  End ;
 12  /
Avant augmentation 7369 SMITH --> 880
Après augmentation 7369 SMITH --> 968

Procédure PL/SQL terminée avec succès.
```

Les paramètres sont passés lors de l'appel de la fonction

D'une façon générale, les procédures ne devraient pas exécuter d'instruction de fin de transaction (COMMIT, ROLLBACK, Ordre DDL)

La décision d'enregistrer ou annuler la transaction en cours relève du programme appelant

Si toutefois, le traitement impose un enregistrement en base, la procédure peut être déclarée Autonome,

via la directive de compilation **PRAGMA AUTONOMOUS_TRANSACTION**

Imaginons que vous ayez besoin d'une procédure de trace qui utilise l'instruction INSERT pour enregistrer vos messages dans une table d'erreurs

Afin de dépister correctement la trace désirée, cette procédure doit enregistrer chaque insertion avec l'instruction **COMMIT**

Cependant, nous voulons que cet enregistrement ne valide que les instructions de notre procédure de trace

Pour atteindre cet objectif, nous allons donc créer une procédure autonome

Nous avons besoin d'une table de trace

```
SQL> CREATE TABLE TRACE(
  2      UTI Varchar2(30) DEFAULT USER
  3     ,DDATE Date DEFAULT SYSDATE
  4     ,LIGNE Varchar2(4000) ) ;

Table créée.
```

Cette table permettra de stocker l'utilisateur, la date et la ligne de trace

Nous allons créer maintenant notre procédure

Celle-ci pourra utiliser au choix la sortie du message sur écran avec la fonction DBMS_OUTPUT.PUT_LINE,

ou bien l'insertion dans la table de trace

Au passage nous allons augmenter les possibilité natives

En effet la fonction DBMS_OUTPUT.PUT_LINE est limitée à 255 caractères, et une colonne VARCHAR2 à 4000

Qu'à cela ne tienne, nous allons contourner le problème en découpant le message en tranches, permettant d'afficher quelque soit la méthode jusqu'à 32767 caractères

```
SQL> CREATE OR REPLACE procedure DEBUG ( PC$Message in VARCHAR2, PC$Output in VARCHAR2 DEFAULT 'E' )
  2  Is
  3  PRAGMA AUTONOMOUS_TRANSACTION ;
  4    LC$Chaine Varchar2(4000) ;
  5    LN$Tranches PLS_INTEGER ;
  6    LN$Reste  PLS_INTEGER ;
  7    LN$Pos PLS_INTEGER := 1 ;
  8    LN$Inc PLS_INTEGER ;
  9  Begin
 10
 11    If PC$Output = 'E' Then
 12       -- Sortie sur ecran (DBMS_OUTPUT) --
 13       LN$Inc := 255 ;
 14       LN$Tranches := Length( PC$Message ) / LN$Inc ;
 15       LN$Reste    := MOD( Length( PC$Message ), LN$Inc ) ;
 16       If LN$Reste > 0 Then LN$Tranches := LN$Tranches + 1 ; End if ;
 17
 18       -- Sortie --
 19       For i in 1..LN$Tranches Loop
 20          LC$Chaine := Substr( PC$Message, LN$Pos, LN$Inc ) ;
 21          DBMS_OUTPUT.PUT_LINE( LC$Chaine ) ;
 22          LN$Pos := LN$Pos + LN$Inc ;
 23       End loop ;
 24
 25    Else
 26       -- Sortie sur table (INSERT) --
 27       LN$Inc := 4000 ;
 28       LN$Tranches := Length( PC$Message ) / LN$Inc ;
 29       LN$Reste    := MOD( Length( PC$Message ), LN$Inc ) ;
 30       If LN$Reste > 0 Then LN$Tranches := LN$Tranches + 1 ; End if ;
 31
 32       -- Sortie --
 33       For i in 1..LN$Tranches Loop
 34          LC$Chaine := Substr( PC$Message, LN$Pos, LN$Inc ) ;
 35          Insert into TRACE (LIGNE) Values ( LC$Chaine ) ;
 36          Commit ; -- enregistrement de la ligne
 37          LN$Pos := LN$Pos + LN$Inc ;
 38       End loop ;
 39    End if ;
 40
 41  End;
 42  /

Procédure créée.
```

Cette procédure accepte en premier paramètre la chaîne de caractères de trace (max 32767 caractères)

et en deuxième paramètre (facultatif) le type de sortie désiré **E** pour écran (défaut), différent de **E** pour table

La directive de compilation **PRAGMA AUTONOMOUS_TRANSACTION** en entête de la section déclarative indique que cette procédure s'exécute dans sa propre transaction.

Utilisons maintenant cette procédure pour notre système de trace sur écran

```
SQL> set linesize 100
SQL> Declare
  2    LC$Ch Varchar2(32767) ;
  3  Begin
  4    LC$Ch := Rpad( '1', 300, '1' ) || Rpad( '2', 300, '2' ) ; -- trace de 600 caractères
  5    Debug( LC$Ch ) ;
  6  End ;
  7  /
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111122222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222

Procédure PL/SQL terminée avec succès.
```

Puis sur table

```
SQL> Declare
  2    LC$Ch Varchar2(32767) ;
  3  Begin
  4    LC$Ch := Rpad( '1', 2100, '1' ) || Rpad( '2', 2100, '2' ) ; -- trace de 4200 caractères
  5    Debug( LC$Ch, 'T' ) ;
  6  End ;
  7  /

Procédure PL/SQL terminée avec succès.
SQL> select * from trace ;

UTI                            DDATE
------------------------------ --------
LIGNE
--------------------------------------------------------------------------------------------------------------------------------
SCOTT                          13/03/04
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111112222222

SCOTT                          13/03/04
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
etc...
```

Lorsqu'un paramètre est passé en mode **OUT**, la procédure peut le modifier.

Le programme appelant doit avoir défini une variable correspondante dans sa section déclarative

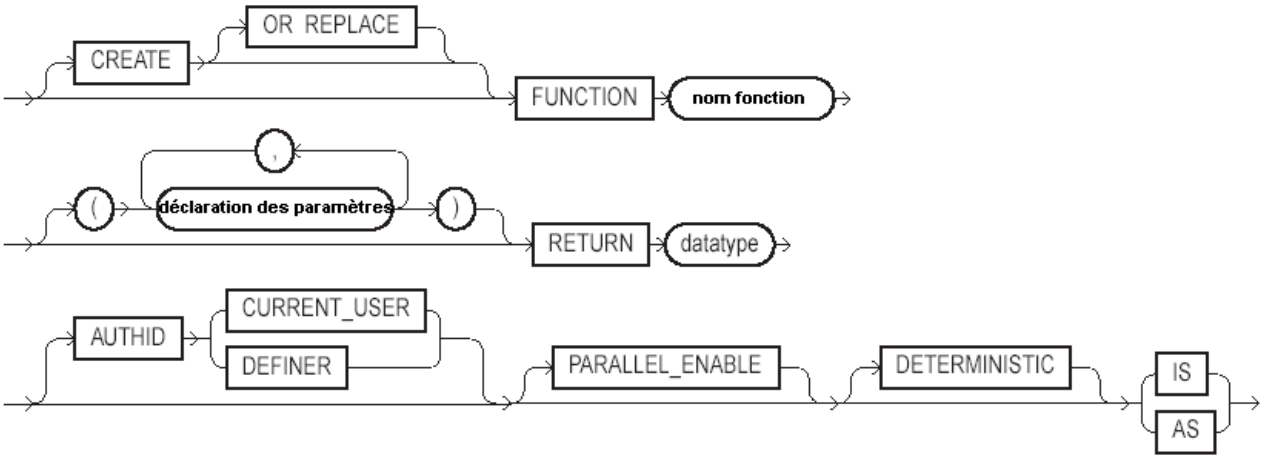
```
SQL> CREATE OR REPLACE PROCEDURE Test_Augmentation
  2     (
  3       PN$Numemp IN EMP.empno%Type
  4      ,PN$Pourcent IN OUT NUMBER
  5     ) IS
  6     LN$Salaire EMP.sal%Type ;
  7  BEGIN
```

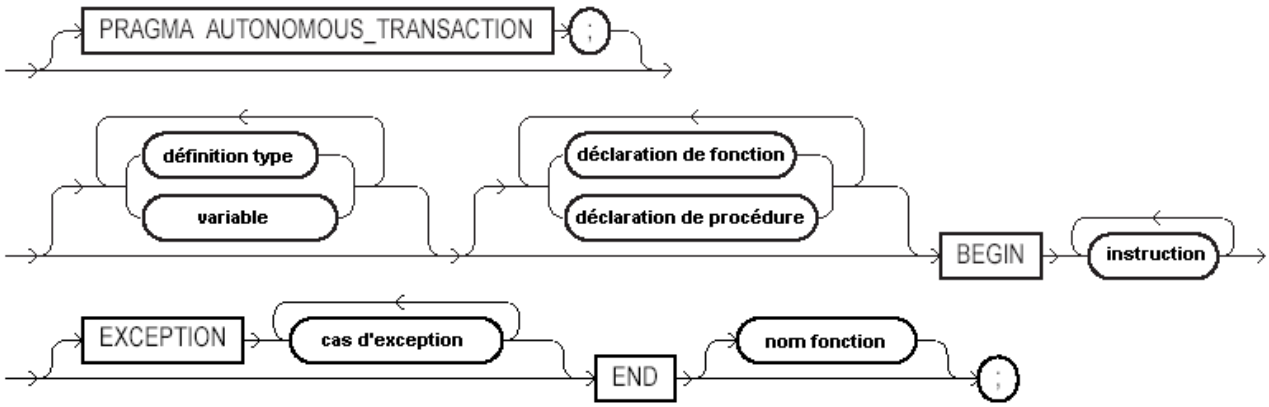
```
  8      Select sal Into LN$Salaire From EMP Where empno = PN$Numemp ;
  9      -- augmentation virtuelle de l'employé
 10      PN$Pourcent := LN$Salaire * PN$Pourcent ;
 11  END;
 12  /

Procédure créée.
SQL> select empno, sal from emp where empno = 7369 ;

     EMPNO        SAL
---------- ----------
      7369        880

SQL> Declare
  2    LN$Pourcent NUMBER := 1.1 ;
  3  Begin
  4    Test_Augmentation( 7369, LN$Pourcent ) ;
  5    dbms_output.put_line( 'Employé 7369 après augmentation : ' || To_char( LN$Pourcent ) ) ;
  6  End ;
  7  /
Employé 7369 après augmentation : 968
```

4.2 - Les Fonctions

Une fonction est identique à une procédure à la différence qu'elle retourne obligatoirement une valeur

d'où le mot clé obligatoire **RETURN**





CREATE indique que l'on veut créer une fonction stockée dans la base

La clause facultative **OR REPLACE** permet d'écraser une fonction existante portant le même nom

nom fonction est le nom donné par l'utilisateur à la fonction

AUTHID indique sur quel schéma la fonction s'applique :

- **CURRENT_USER**

Indique que la fonction utilise les objets du schéma de l'utilisateur qui appelle la fonction

- **DEFINER**(défaut)

Indique que la fonction utilise les objets du schéma de création de la fonction

nom paramètre est le nom donné par l'utilisateur au paramètre transmis

IN(valeur par défaut) indique que le paramètre transmis par le programme appelant n'est pas modifiable par la fonction

OUT indique que le paramètre est modifiable par la procédure

IN OUT indique que le paramètre est transmis par le programme appelant et renseigné par la fonction

NOCOPY indique que le paramètre est transmis par référence (pointeur) et non par copie de la valeur

datatype représente le type SQL ou PL/SQL du paramètre

:= représente le symbole d'assignation d'une valeur par défaut

DEFAULT identique à :=

expression représente la valeur par défaut du paramètre (doit être conforme au type du paramètre)

PRAGMA AUTONOMOUS_TRANSACTION indique que la fonction sera exécutée dans une transaction autonome

Au chapitre sur les procédures, nous avons vu la procédure permettant de simuler une augmentation

en retournant le nouveau salaire théorique dans une variable IN OUT

Transformons cette procédure en fonction

```
SQL> CREATE OR REPLACE FUNCTION F_Test_Augmentation
  2      (
  3        PN$Numemp IN EMP.empno%Type
  4       ,PN$Pourcent IN NUMBER
  5      ) Return NUMBER
  6  IS
  7      LN$Salaire EMP.sal%Type ;
  8  BEGIN
  9       Select sal Into LN$Salaire From EMP Where empno = PN$Numemp ;
 10       -- augmentation virtuelle de l'employé
 11       LN$Salaire := LN$Salaire * PN$Pourcent ;
 12    Return( LN$Salaire ) ; -- retour de la valeur
 13  END;
 14  /

Fonction créée.
```

La valeur de retour de la fonction est directement utilisable, même sans déclaration d'une variable d'accueil

```
SQL> Declare
  2    LN$Salaire emp.sal%Type ;
  3  Begin
  4    Select sal Into LN$Salaire From EMP Where empno = 7369 ;
  5    dbms_output.put_line( 'Salaire de 7369 avant augmentation ' || To_char( LN$Salaire ) ) ;
  6    dbms_output.put_line( 'Salaire de 7369 après augmentation ' || To_char( F_Test_Augmentation(
7369, 1.1 ) ) ) ;
  7  End ;
  8  /
Salaire de 7369 avant augmentation 880
Salaire de 7369 après augmentation 968

Procédure PL/SQL terminée avec succès.
```

4.3 - Les Paquetages

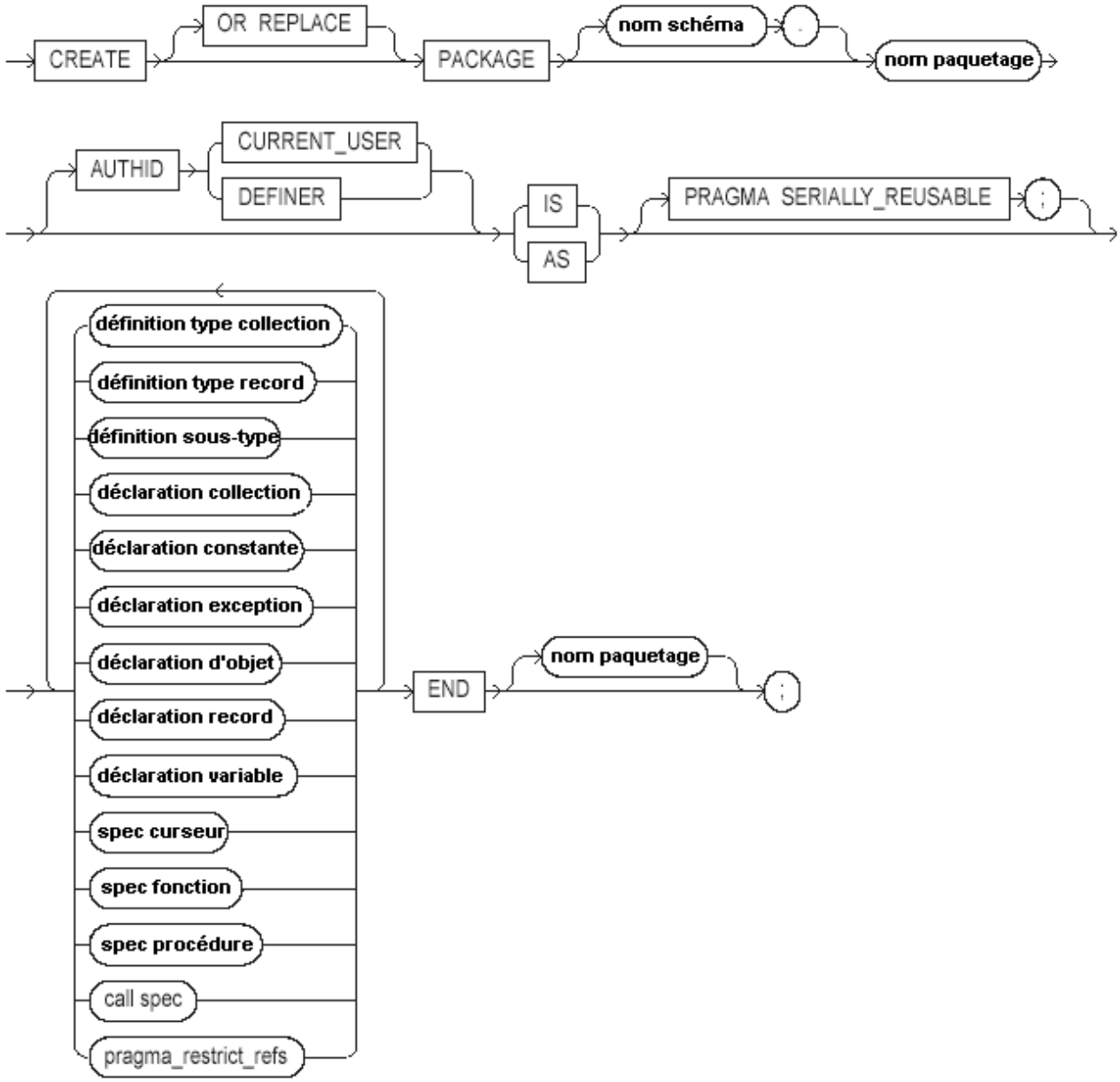
Un paquetage est un ensemble de procédures et fonctions regroupées dans un objet nommé

Par exemple le paquetage Oracle DBMS_LOB regroupe toutes les fonctions et procédures manipulant les grands objets (LOBs)

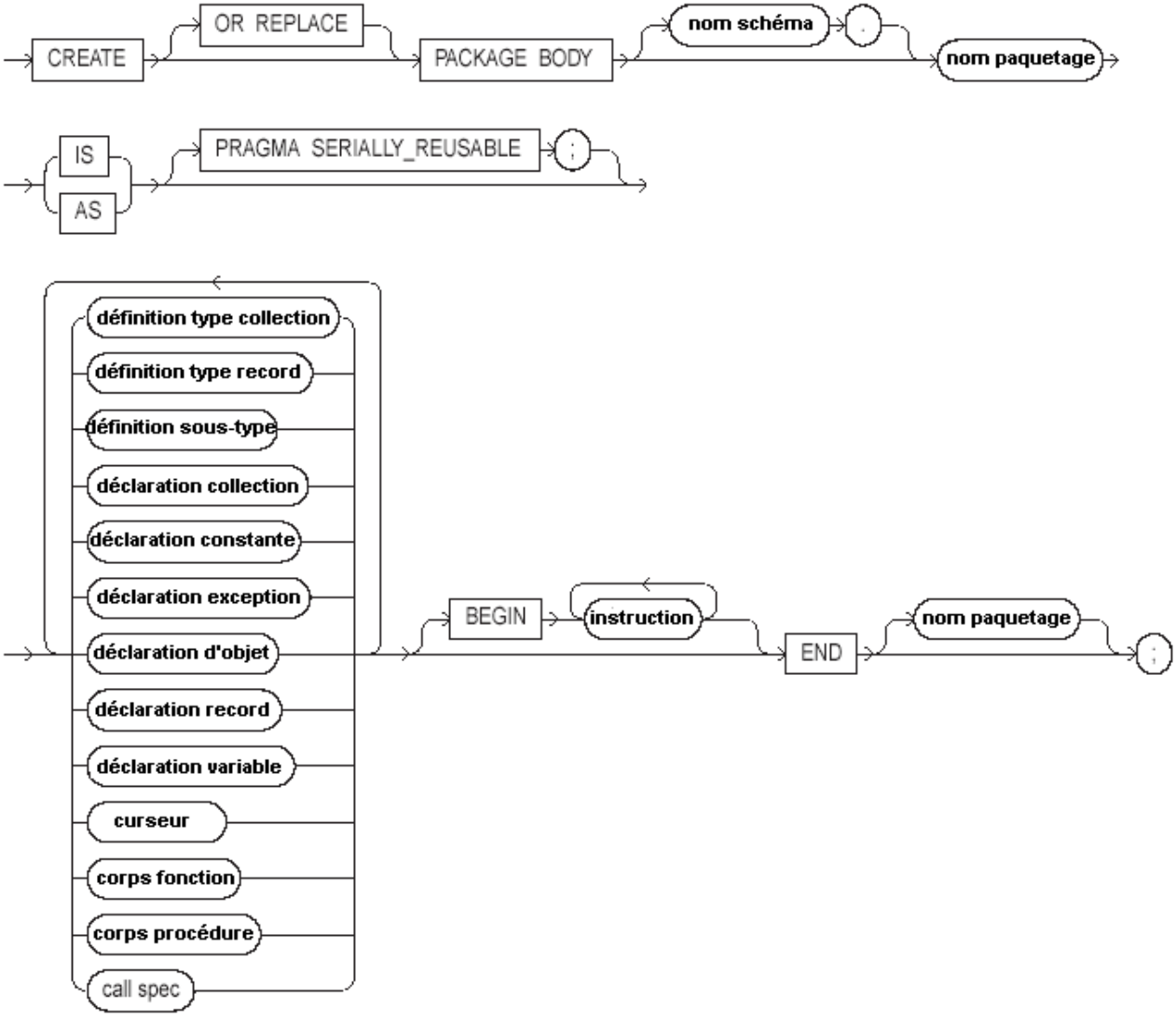
Le paquetage UTL_FILE regroupe les procédures et fonctions permettant de lire et écrire des fichiers du système d'exploitation

Un paquetage est organisé en deux parties distinctes

Déclaration, spécification de paquetage



Corps de paquetage



La déclaration de la partie spécification d'un paquetage s'effectue avec l'instruction **CREATE [OR REPLACE] PACKAGE**

Celle de la partie corps avec l'instruction **CREATE [OR REPLACE] PACKAGE BODY**

```
SQL> CREATE OR REPLACE PACKAGE Pkg_Finance
  2  IS
  3    -- Variables globales et publiques
  4    GN$Salaire EMP.sal%Type ;
  5
  6    -- Fonctions publiques
  7    FUNCTION F_Test_Augmentation
  8          (
  9            PN$Numemp IN EMP.empno%Type
 10           ,PN$Pourcent IN NUMBER
```

```
 11          ) Return NUMBER ;
 12
 13    -- Procédures publiques
 14    PROCEDURE Test_Augmentation
 15       (
 16         PN$Numemp IN EMP.empno%Type -- numéro de l'employé
 17        ,PN$Pourcent IN OUT NUMBER       -- pourcentage d'augmentation
 18       ) ;
 19
 20  End Pkg_Finance ;
 21  /

Package créé.
```

```
SQL> CREATE OR REPLACE PACKAGE BODY Pkg_Finance IS
  2
  3     -- Variables globales privées
  4     GR$Emp EMP%Rowtype ;
  5
  6     -- Procédure privées
  7     PROCEDURE Affiche_Salaires
  8     IS
  9       CURSOR C_EMP IS select * from EMP ;
 10     BEGIN
 11       OPEN C_EMP ;
 12       Loop
 13         FETCH C_EMP Into GR$Emp ;
 14         Exit when C_EMP%NOTFOUND ;
 15         dbms_output.put_line( 'Employé ' || GR$Emp.ename || ' --> ' || Lpad( To_char( GR$Emp.sal
), 10 ) ) ;
 16       End loop ;
 17       CLOSE C_EMP ;
 18     END Affiche_Salaires ;
 19
 20     -- Fonctions publiques
 21     FUNCTION F_Test_Augmentation
 22         (
 23           PN$Numemp IN EMP.empno%Type
 24          ,PN$Pourcent IN NUMBER
 25         ) Return NUMBER
 26     IS
 27         LN$Salaire EMP.sal%Type ;
 28     BEGIN
 29          Select sal Into LN$Salaire From EMP Where empno = PN$Numemp ;
 30          -- augmentation virtuelle de l'employé
 31          LN$Salaire := LN$Salaire * PN$Pourcent ;
 32
 33          -- Affectation de la variable globale publique
 34          GN$Salaire := LN$Salaire ;
 35
 36       Return( LN$Salaire ) ; -- retour de la valeur
 37     END F_Test_Augmentation;
 38
 39     -- Procédures publiques
 40     PROCEDURE Test_Augmentation
 41        (
 42          PN$Numemp IN EMP.empno%Type
 43         ,PN$Pourcent IN OUT NUMBER
 44        ) IS
 45        LN$Salaire EMP.sal%Type ;
 46     BEGIN
 47         Select sal Into LN$Salaire From EMP Where empno = PN$Numemp ;
 48         -- augmentation virtuelle de l'employé
 49        PN$Pourcent := LN$Salaire * PN$Pourcent ;
 50
 51        -- appel procédure privée
 52        Affiche_Salaires ;
 53
 54     END Test_Augmentation;
 55
 56
 57  END Pkg_Finance;
 58  /

Corps de package créé.
```

La spécification du paquetage est créée avec une variable globale et publique : GN$Salaire

une procédure publique : PROCEDURE Test_Augmentation

une fonction publique : FUNCTION F_Test_Augmentation

qui sont visibles depuis l'extérieur (le programme appelant)

Le corps du paquetage est créé avec une procédure privée : PROCEDURE Afiche_Salaires qui n'est visible que dans le corps du paquetage

Le corps définit également une variable globale au corps du paquetage : GR$Emp utilisée par la procédure privée

L'accès à un objet d'un paquetage est réalisé avec la syntaxe suivante :

nom_paquetage.nom_objet[(liste paramètres)]

Appel de la fonction F_Test_Augmentation du paquetage

```
SQL> Declare
  2      LN$Salaire emp.sal%Type ;
  3  Begin
  4      Select sal Into LN$Salaire From EMP Where empno = 7369 ;
  5      dbms_output.put_line( 'Salaire de 7369 avant augmentation '
  6          || To_char( LN$Salaire ) ) ;
  7      dbms_output.put_line( 'Salaire de 7369 après augmentation '
  8          || To_char( Pkg_Finance.F_Test_Augmentation( 7369, 1.1 ) ) ) ;
  9  End ;
 10  /
Salaire de 7369 avant augmentation 880
Salaire de 7369 après augmentation 968

Procédure PL/SQL terminée avec succès.
```

Appel de la procédure Test_Augmentation du paquetage

```
SQL> Declare
  2       LN$Pourcent NUMBER := 1.1 ;
  3  Begin
  4       Pkg_Finance.Test_Augmentation( 7369, LN$Pourcent ) ;
  5       dbms_output.put_line( 'Employé 7369 après augmentation : ' || To_char( LN$Pourcent ) ) ;
  6  End ;
  7  /
Employé SMITH -->        880
Employé ALLEN -->       1936
Employé WARD -->       1375
Employé JONES -->       3273
Employé MARTIN -->       1375
Employé BLAKE -->       3135
Employé CLARK -->       2695
Employé SCOTT -->       3300
Employé KING -->       5500
Employé TURNER -->       1650
Employé ADAMS -->       1210
```

```
Employé JAMES -->       1045
Employé FORD -->       3300
Employé MILLER -->       1430
Employé Dupontont -->
Employé Duboudin -->
Employé 7369 après augmentation : 968

Procédure PL/SQL terminée avec succès.
```

Interrogation de la variable globale publique : GN$Salaire

```
SQL> Begin
  2       dbms_output.put_line( 'Valeur salaire du package : ' || To_char( Pkg_Finance.GN$Salaire )
) ;
  3  End ;
  4  /
Valeur salaire du package : 968

Procédure PL/SQL terminée avec succès.
```

4.4 - Fonctions sur des ensembles de lignes (PIPELINED) (9i)

Oracle 9i

Depuis la version 9i apparaît une nouvelle possibilité de PL/SQL de pouvoir définir des fonctions qui acceptent en argument des collections ou des références à un curseur, et qui retournent les données au fur et à mesure de l'exécution de la fonction.

Lors de la déclaration de la fonction, le mot clé PIPELINED est ajouté dans l'entête et les informations sont retournées à l'aide de la commande PIPE ROW.

Soit l'exemple suivant

```
SQL> create type MtSal as object (numemp number(4), salaire number(7,2) ) ;
  2  /

Type créé.

SQL> create type MtSalTab as table of MtSal ;
  2  /

Type créé.
```

```
SQL> CREATE OR REPLACE function LigneSalaire (cur_lig in SYS_REFCURSOR)
  2    return MtSalTab PIPELINED
  3  IS
  4   LSal  MtSal := MtSal (NULL,NULL) ;
  5   Remp  emp%rowtype ;
  6  Begin
  7   Loop
  8     Fetch cur_lig into Remp ;
  9     Exit When cur_lig%NOTFOUND ;
 10       -- Manipulation des données --
 11       LSal.numemp := Remp.empno ;
 12       LSal.salaire := Remp.sal * 1.1 ;
 13       -- Retour des valeurs --
 14       PIPE ROW( LSal ) ;
 15   End loop ;
 16   Return ;
 17  End ;
```

```
 18  /

Fonction créée.
```

```
SQL> Select numemp, salaire from
  2  table(LigneSalaire(CURSOR(Select * from EMP)))
  3  /

    NUMEMP    SALAIRE
---------- ----------
      7369        968
      7499       1936
      7521     1512,5
      7566     3600,3
      7654     1512,5
      7698     3448,5
      7782     2964,5
      7788       3630
      7839       6050
      7844       1815
      7876       1331
      7900     1149,5
      7902       3630
      7934       1573

14 ligne(s) sélectionnée(s).
```

L'instruction **table(LigneSalaire(CURSOR(Select * from EMP)))** indique que la fonction LigneSalaire retourne un ensemble de données.

Cet exemple est bien évidemment simpliste, mais on peut imaginer toutes sortes de traitements sur les lignes du curseur avant de renvoyer le résultat.

Pour faire une estimation de la masse salariale totale après une augmentation de salaire de 10% pour l'ensemble du personnel, il suffit d'interroger la fonction de la façon suivante :

```
SQL> Select sum(salaire) from
  2  table(LigneSalaire(CURSOR(Select * from EMP)))
  3  /

SUM(SALAIRE)
------------
     35120,8
```

Bien entendu un résultat équivalent pourrait être obtenu avec la requête :

select sum(sal * 1.1) from emp;

mais, répétons-le le traitement effectué dans la fonction peut être beaucoup plus élaboré.

4.5 - Maintenance des objets procéduraux

Visualisation du code source

Le code source des objets procéduraux est visible par l'intermédiare des vues du dictionnaire de données

USER_SOURCE pour les objets appartenant au schéma

ALL_SOURCE pour les objets appartenant aux schémas accessibles

DBA_SOURCE pour les objets appartenant à tous les schémas

```
SQL> desc user_source
 Nom
 ----------------------
 NAME
 TYPE
 LINE
 TEXT
```

Pour dissocier le code des procédures, fonctions et paquetages, la restriction sur la colonne TYPE doit être la suivante :

- **Procédure** : TYPE = 'PROCEDURE'
- **Fonction** : TYPE = 'FUNCTION'
- **Spécification de paquetage** : TYPE = 'PACKAGE'
- **Corps de paquetage** : TYPE = 'PACKAGE BODY'

```
SQL> Select text from user_source
  2  Where name = 'AUGMENTATION'
  3  And   type = 'PROCEDURE'
  4  /

TEXT
----------------------------------------
PROCEDURE Augmentation
 (
   PN$Numemp IN EMP.empno%Type
  ,PN$Pourcent IN NUMBER
 ) IS
BEGIN
  -- augmentation de l'employé
  Update EMP Set sal = sal * PN$Pourcent
  Where empno = PN$Numemp ;
END;

10 ligne(s) sélectionnée(s).
```

Compilation des objets

- **Procédure** : ALTER PROCEDURE nom_procédure COMPILE
- **Fonction** : ALTER FUNCTION nom_fonction COMPILE
- **Spécification de paquetage** : ALTER PACKAGE nom_package COMPILE PACKAGE
- **Corps de paquetage** : ALTER PACKAGE nom_package COMPILE BODY
- **Spécification + Corps de paquetage** : ALTER PACKAGE nom_package COMPILE

Suppression des objets

- **Procédure** : DROP PROCEDURE nom_procédure
- **Fonction** : DROP FUNCTION nom_fonction
- **Paquetage entier** : DROP PACKAGE nom_package
- **Corps de paquetage** : DROP PACKAGE BODY nom_package

5 - Collections et enregistrements

Ces types de données n'existent qu'en PL/SQL et n'ont pas d'équivalent dans la base Oracle

Il n'est pas possible de stocker un enregistrement directement dans la base

Les collections

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)

On peut distinguer trois types différents de collections :

Les collections de type NESTED TABLE et VARRAY doivent-être 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

Un enregistrement ressemble à une structure d'un L3G

Il est composé de champs qui peuvent être de type différent

5.1 - Déclarations et initialisation

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 ;

index_by_type représente l'un des types suivants :

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

```
SQL> declare
  2    -- collection de type nested table
  3    TYPE TYP_NES_TAB is table of varchar2(100) ;
  4    -- collection de type index by
  5    TYPE TYP_IND_TAB is table of number index by binary_integer ;
  6    tab1 TYP_NES_TAB ;
  7    tab2 TYP_IND_TAB ;
  8  Begin
  9    tab1 := TYP_NES_TAB('Lundi','Mardi','Mercredi','Jeudi' ) ;
 10    for i in 1..10 loop
 11       tab2(i):= i ;
 12    end loop ;
 13  End;
 14  /

Procédure PL/SQL terminée avec succès.
```

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] ;

```
SQL> declare
  2    -- collection de type VARRAY
  3    TYPE TYP_VAR_TAB is VARRAY(30) of varchar2(100) ;
  4    tab1 TYP_VAR_TAB := TYP_VAR_TAB('','','','','','','','','','');
  5  Begin
  6    for i in 1..10 loop
  7       tab1(i):= to_char(i) ;
  8    end loop ;
  9  End;
 10  /

Procédure PL/SQL terminée avec succès.
```

Déclaration d'un tableau VARRAY de 30 éléments de type varchar2(100)

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

```
SQL> declare
  2    -- Record --
  3    TYPE T_REC_EMP IS RECORD (
  4       Num emp.empno%TYPE,
  5       Nom emp.ename%TYPE,
  6       Job emp.job%TYPE );
  7
  8    R_EMP T_REC_EMP ; -- variable enregistrement de type T_REC_EMP
  9  Begin
 10    R_EMP.Num := 1 ;
 11    R_EMP.Nom := 'Scott' ;
 12    R_EMP.job := 'GASMAN' ;
 13  End;
 14  /

Procédure PL/SQL terminée avec succès.
```

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

```
SQL> declare
  2    -- Record --
  3    TYPE T_REC_EMP IS RECORD (
  4       Num emp.empno%TYPE,
  5       Nom emp.ename%TYPE,
  6       Job emp.job%TYPE );
  7    -- Table de records --
  8    TYPE TAB_T_REC_EMP IS TABLE OF T_REC_EMP index by binary_integer ;
  9    t_rec TAB_T_REC_EMP ; -- variable tableau d'enregistrements
 10  Begin
 11    t_rec(1).Num := 1 ;
 12    t_rec(1).Nom := 'Scott' ;
 13    t_rec(1).job := 'GASMAN' ;
 14    t_rec(2).Num := 2 ;
 15    t_rec(2).Nom := 'Smith' ;
 16    t_rec(2).job := 'CLERK' ;
 17  End;
 18  /

Procédure PL/SQL terminée avec succès.
```

Les éléments d'un enregistrement peuvent être des objets, des collections ou d'autres enregistrements.

```
Declare
  TYPE Temps IS RECORD
  (
    heures   SMALLINT,
    minutes  SMALLINT,
    secondes SMALLINT
```

```
    );
    TYPE Vol IS RECORD
    (
      numvol     PLS_INTEGER,
      Numavion   VARCHAR2(15),
      Commandant Employe,      -- type objet
      Passagers  ListClients, -- type nested table
      depart     Temps,        -- type record
      arrivee    Temps         -- type record
    );
  Begin
    ...
  End  ;
```

A la différence des types VARRAY et (NESTED)TABLES, les types RECORD ne peuvent pas être créés et stockés dans la base.

Initialisation des collections

Les collections de type NESTED TABLE et VARRAY doivent être initialisées avant toute utilisation (à l'exception des collections de type INDEX-BY TABLE).

Pour initialiser une collection, il faut se référer à son constructeur. Celui-ci, créé automatiquement par Oracle porte le même nom que la collection.

```
  Declare
  -- Déclaration d'un type tableau VARRAY de 30 éléments de type Varchar2(100)
  TYPE TYP_VAR_TAB is VARRAY(30) of varchar2(100) ;
  -- Déclaration et initialisation d'une variable de type TYP_VAR_TAB
  tab1 TYP_VAR_TAB := TYP_VAR_TAB('','','','','','','','','','');
```

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.

```
  Declare
  TYPE TYP_VAR_TAB is VARRAY(30) of varchar2(100) ;
  tab1 TYP_VAR_TAB; -- collection NULL
```

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 ;
```

5.2 - Accès aux éléments d'une collection

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**31 et 2**31

Pour une collection de type NESTED TABLE, l'indice doit être un nombre valide compris entre 1 et 2**31

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.

Indice peut être un littéral, une variable ou une expression

```
  1   Declare
  2     Type TYPE_TAB_EMP IS TABLE OF Varchar2(60) INDEX BY BINARY_INTEGER ;
  3     emp_tab TYPE_TAB_EMP ;
  4     i       pls_integer ;
  5   Begin
  6     For i in 0..10 Loop
  7       emp_tab( i+1 ) := 'Emp ' || ltrim( to_char( i ) ) ;
  8     End loop ;
  9* End ;
SQL> /

Procédure PL/SQL terminée avec succès.
```

```
  1  Declare
  2    Type TYPE_TAB_JOURS IS TABLE OF PLS_INTEGER INDEX BY VARCHAR2(20) ;
  3    jour_tab TYPE_TAB_JOURS ;
  4  Begin
```

```
    5      jour_tab( 'LUNDI' ) := 10 ;
    6      jour_tab( 'MARDI' ) := 20 ;
    7      jour_tab( 'MERCREDI' ) := 30 ;
    8* End ;
  SQL> /

  Procédure PL/SQL terminée avec succès.
```

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 ;
```

Les collections ne peuvent pas être comparées entre elles.

Les opérateurs d'égalité ou de comparaison ne peuvent pas être utilisés entre 2 collections

```
  SQL> Declare
    2    Type TYPE_TAB_STRING  IS TABLE OF Varchar2(10) ;
    3    tab1 TYPE_TAB_STRING  := TYPE_TAB_STRING( '1','2','3' );
    4    tab2 tab1%TYPE        := TYPE_TAB_STRING( '1','2','3' );
    5  Begin
    6     If tab1 = tab2 Then
    7        null ;
    8     End if ;
    9  End ;
   10  /
     If tab1 = tab2 Then
        *
  ERREUR à la ligne 6 :
  ORA-06550: Ligne 6, colonne 12 :
  PLS-00306: numéro ou types d'arguments erronés dans appel à '='
  ORA-06550: Ligne 6, colonne 4 :
  PL/SQL: Statement ignored
```

(10g)

Les collections de même type peuvent être comparées en égalité ou différence

```
  DECLARE
     TYPE Colors IS TABLE OF VARCHAR2(64);
     primaries Colors := Colors('Blue','Green','Red');
     rgb Colors := Colors('Red','Green','Blue');
     traffic_light Colors := Colors('Red','Green','Amber');
  BEGIN
  -- On peut utiliser = ou !=, mais pas < ou >.
  -- Notez que ces 2 collections sont égales même si leurs membres sont dans un ordre différent.
     IF primaries = rgb THEN
        dbms_output.put_line('OK, PRIMARIES et RGB ont les mêmes membres.');
     END IF;
     IF rgb != traffic_light THEN
```

```
        dbms_output.put_line('RGB et TRAFFIC_LIGHT ont des membres différents.');
    END IF;
END;
```

(10g)

Il est possible d'appliquer certains opérateurs sur des tables imbriquées

```
DECLARE
  TYPE nested_typ IS TABLE OF NUMBER;
  nt1 nested_typ := nested_typ(1,2,3);
  nt2 nested_typ := nested_typ(3,2,1);
  nt3 nested_typ := nested_typ(2,3,1,3);
  nt4 nested_typ := nested_typ(1,2,4);
  reponse BOOLEAN;
  combien NUMBER;
  PROCEDURE verif(test BOOLEAN DEFAULT NULL, quantite NUMBER DEFAULT NULL) IS
  BEGIN
     IF truth IS NOT NULL THEN
        dbms_output.put_line(CASE test WHEN TRUE THEN 'True' WHEN FALSE THEN 'False' END);
     END IF;
     IF quantity IS NOT NULL THEN
        dbms_output.put_line(quantite);
     END IF;
  END;
BEGIN
  reponse := nt1 IN (nt2,nt3,nt4); -- true, nt1 correspond à nt2
  verif(test => reponse);
  reponse := nt1 SUBMULTISET OF nt3; -- true, tous les éléments correspondent
  verif(test => reponse);
  reponse := nt1 NOT SUBMULTISET OF nt4; -- true
  verif(test => reponse);

  combien := CARDINALITY(nt3); -- nombre d'éléments dans nt3
  verif(quantite => combien);
  combien := CARDINALITY(SET(nt3)); -- nombre d'éléments distincts
  verif(quantite => combien);

  reponse := 4 MEMBER OF nt1; -- false, aucun élément ne correspond
  verif(test => reponse);
  reponse := nt3 IS A SET; -- false, nt3 a des éléments dupliqués
  verif(test => reponse);
  reponse := nt3 IS NOT A SET; -- true, nt3 a des éléments dupliqués
  verif(test => reponse);
  reponse := nt1 IS EMPTY; -- false, nt1 a des éléments
  verif(test => reponse);
END;
```

5.3 - Méthodes associées aux collections

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

Seule la méthode EXISTS peut être utilisée sur une collection NULL.

L'utilisation de toute autre méthode sur une collection NULL provoque l'exception COLLECTION_IS_NULL

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

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

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$Nbre pls_integer ;
Begin
  LN$Nbre := ma_collection.COUNT ;
End ;
```

LIMIT

Cette méthode retourne le nombre maximum d'éléments permis d'une collection

Elle n'est utile que pour les collections de type VARRAY et retourne NULL pour les collections des autre types

```
Declare
    -- collection de type VARRAY
    TYPE TYP_VAR_TAB is VARRAY(30) of varchar2(100) ;
    tab1 TYP_VAR_TAB := TYP_VAR_TAB('','','','','','','','','','');
Begin
    for i in 1..tab1.LIMIT loop
       ###.
    end loop ;
End;
```

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

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

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 ;
```

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 ;
```

EXTEND[(n[,i])]

Cette méthode permet d'étendre une collection par ajout de nouveaux éléments

Elle dispose de 3 syntaxes différentes

- **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[(n)]

Cette méthode permet de supprimer un ou plusieurs éléments situés à la fin d'une collection

Elle dispose de 2 formes de syntaxe différentes

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[(n[,m])]

Cette méthode permet de supprimer un, plusieurs, ou la totalité des éléments d'une collection

Elle dispose de 3 formes de syntaxe différentes

- **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)**

Suppression de l'élément d'indice n de 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(4,1) ;
  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    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.

```
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    dbms_output.put_line( 'Réassignation des élément d''indice 4, 5 et 6' ) ;
 14    tab(4) := 'Jeudi' ;
 15    tab(5) := 'Vendredi' ;
 16    tab(6) := 'Samedi' ;
 17    dbms_output.put_line( 'tab.COUNT = ' || tab.COUNT) ;
 18    For i in tab.first..tab.last Loop
 19      If tab.EXISTS(i) Then
 20         dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') = ' || tab(i) ) ;
 21      Else
 22         dbms_output.put_line( 'tab(' || ltrim( to_char( i ) ) || ') inexistant ') ;
 23      End if ;
 24    End loop ;
 25  End;
 26  /
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
Réassignation des élément d'indice 4, 5 et 6
tab.COUNT = 7
tab(1) = lundi
tab(2) = mardi
tab(3) = mercredi
tab(4) = Jeudi
tab(5) = Vendredi
tab(6) = Samedi
tab(7) = lundi

Procédure PL/SQL terminée avec succès.
```

Principales exceptions

```
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 ;
```

5.4 - Utilisation des collections avec les données issues de la base

Prenons l'exemple d'une table des entêtes de factures qui stocke également les lignes des factures

Sous Sql*Plus définissons le type ligne de facture (TYP_LIG_FAC)

```
SQL>  CREATE TYPE TYP_LIG_FAC AS OBJECT (
  2   numlig Integer,
  3   code   Varchar2(20),
  4   Pht    Number(6,2),
  5   Tva    Number(3,2),
  6   Qte    Integer
  7   );
  8  /

Type créé.
```

Définissons le type TYP_TAB_LIG_FAC comme étant une collection d'éléments du type TYP_LIG_FAC

```
SQL>  CREATE TYPE TYP_TAB_LIG_FAC AS TABLE OF TYP_LIG_FAC ;
  2  /

Type créé.
```

Création de la table des factures

```
SQL> CREATE TABLE FACTURE (
  2   numero   Number(9),
  3   numcli   Number(6),
  4   date_fac Date,
  5   ligne    TYP_TAB_LIG_FAC )
  6   NESTED TABLE ligne STORE AS ligne_table ;

Table créée.
```

Chaque élément présent dans la colonne ligne est une collection de type NESTED TABLE qui va permettre de stocker les différentes lignes de la facture

Insertion de données dans la table FACTURE

```
SQL> Begin
  2     Insert into FACTURE
  3       values( 1, 1214, sysdate,
  4       Typ_tab_lig_fac( Typ_lig_fac( 1, 'Oracle 9i', 999.99, 5.5, 3 ),
  5                        Typ_lig_fac( 1, 'Forms 9i', 899.99, 5.5, 3 ),
  6                        Typ_lig_fac( 1, 'Reports 9i', 699.99, 5.5, 3 )
  7           )
  8     );
  9
 10     Insert into FACTURE
 11       values( 2, 1215, sysdate,
 12       Typ_tab_lig_fac( Typ_lig_fac( 1, 'Oracle 9i', 999.99, 5.5, 1 ),
 13                        Typ_lig_fac( 1, 'Forms 9i', 899.99, 5.5, 1 ),
 14                        Typ_lig_fac( 1, 'Reports 9i', 699.99, 5.5, 1 )
 15           )
 16     );
 17  End ;
 18  /

Procédure PL/SQL terminée avec succès.

SQL> commit;

Validation effectuée.
```

Modification d'une facture

```
SQL> Declare
  2    Tab_lig Typ_tab_lig_fac := Typ_tab_lig_fac(
  3              Typ_lig_fac( 1, 'Forms 9i', 899.99, 5.5, 2 ),
  4              Typ_lig_fac( 1, 'Reports 9i', 699.99, 5.5, 2 )
  5              );
  6  Begin
  7    Update FACTURE
  8    Set ligne = Tab_lig Where numero = 2 ;
  9  End;
 10  /

Procédure PL/SQL terminée avec succès.

SQL> commit;

Validation effectuée.
```

Utilisation d'un type enregistrement (RECORD) pour sélectionner une ligne de la table FACTURE ainsi que toutes les lignes rattachées via la NESTED TABLE

```
SQL> Declare
  2    TYPE Fact_rec IS RECORD
  3    (
  4      numero   facture.NUMERO%type,
  5      numcli   facture.NUMCLI%type,
  6      date_fac facture.DATE_FAC%type,
  7      lignes   facture.LIGNE%type
  8    ) ;
  9    rec_fact Fact_rec ;
 10    Cursor C_Fact is select * from facture ;
 11  Begin
 12    Open C_Fact ;
 13    Loop
 14      Fetch C_Fact into rec_fact ;
```

```
 15      Exit when C_Fact%NOTFOUND ;
 16      For i IN 1..rec_fact.lignes.last Loop
 17        dbms_output.put_line( 'Numcli/Numfac ' || rec_fact.numcli || '/' || rec_fact.numero
 18          || ' Ligne ' || rec_fact.lignes(i).numlig
 19          || ' code ' || rec_fact.lignes(i).code || ' Qté '
 20          || To_char(rec_fact.lignes(i).qte) ) ;
 21      End loop ;
 22    End loop ;
 23  End ;
 24  /
Numcli/Numfac 1214/1 Ligne 1 code Oracle 9i Qté 3
Numcli/Numfac 1214/1 Ligne 1 code Forms 9i Qté 3
Numcli/Numfac 1214/1 Ligne 1 code Reports 9i Qté 3
Numcli/Numfac 1215/2 Ligne 1 code Forms 9i Qté 2
Numcli/Numfac 1215/2 Ligne 1 code Reports 9i Qté 2

Procédure PL/SQL terminée avec succès.
```

Le champ lignes de l'enregistrement est déclaré de type LIGNE%type donc de type TYP_LIG_FAC.

On récupère dans un enregistrement l'entête de la facture ainsi que toutes les colonnes des lignes attachées.

Ou l'on s'aperçoit que le type RECORD permet de stocker et manipuler des objets complexes.

Une variable de type RECORD peut être utilisée dans une clause RETURNING INTO

```
SQL> Declare
  2    TYPE Emp_rec IS RECORD
  3    (
  4      empno    emp.empno%type,
  5      empname  emp.ename%type,
  6      salaire  emp.sal%type
  7    );
  8    emp_info Emp_rec ;
  9  Begin
 10      Select empno, ename, sal Into emp_info From EMP where empno = 7499 ;
 11      dbms_output.put_line( 'Ancien  salaire pour ' || emp_info.empno || ' : ' ||
To_char(emp_info.salaire) ) ;
 12
 13      Update EMP set sal = sal * 1.1 Where empno = 7499
 14         RETURNING empno, ename, sal INTO emp_info ;
 15
 16      dbms_output.put_line( 'Nouveau salaire pour ' || emp_info.empno || ' : ' ||
To_char(emp_info.salaire) ) ;
 17  End  ;
 18  /
Ancien  salaire pour 7499 : 1760
Nouveau salaire pour 7499 : 1936

Procédure PL/SQL terminée avec succès.
```

5.5 - Traitements en masse des collections

Les collections permettent le traitement des données en " masse "

Elles permettent de charger les données d'une table, de les traiter puis de les enregistrer dans la base

Afin de limiter les interactions coûteuses entre le moteur PL/SQL et le moteur SQL,

les collections peuvent être traitées intégralement grâce à la copie des données par blocs

Cette copie des données par blocs autorise un ordre SQL à traiter toute la collection grâce aux instructions BULK COLLECT et FORALL.

Pour s'en convaincre, analysons le code suivant :

```
SQL> Declare
  2  TYPE   TYP_TAB_NUM IS TABLE OF TEST.B%TYPE INDEX BY PLS_INTEGER ;
  3  TYPE   TYP_TAB_CAR IS TABLE OF TEST.A%TYPE INDEX BY PLS_INTEGER ;
  4  tab1   TYP_TAB_NUM ;
  5  tab2   TYP_TAB_CAR ;
  6  LN$T1  number ;
  7  LN$T2  number ;
  8  LN$T3  number ;
  9  begin
 10     For i in 1..50000 Loop
 11       tab1(i) := i ;
 12       tab2(i) := ltrim( to_char( i ) ) ;
 13     End loop ;
 14
 15     Select to_char( sysdate, 'SSSSS' ) into LN$T1 from dual ;
 16
 17     For i in 1..50000 Loop
 18         Insert into TEST( A, B ) Values( tab2(i), tab1(i) ) ;
 19     End loop ;
 20
 21     Select to_char( sysdate, 'SSSSS' ) into LN$T2 from dual ;
 22
 23     Forall i in 1..50000
 24         Insert into TEST( A, B ) Values( tab2(i), tab1(i) ) ;
 25
 26     Select to_char( sysdate, 'SSSSS' ) into LN$T3 from dual ;
 27
 28     dbms_output.put_line( 'Temps d''exécution en secondes' ) ;
 29     dbms_output.put_line( 'For    ' || to_char(LN$T2 - LN$T1) ) ;
 30     dbms_output.put_line( 'Forall ' || to_char(LN$T3 - LN$T2) ) ;
 31
 32
 33  End ;
 34  /
Temps d'exécution en secondes
For    14
Forall 1
```

Les deux parties de code réalisent exactement la même opération soit l'insertion de 50000 lignes dans une table.

Cependant les temps d'exécutions respectifs sont sans commune mesure

La différence s'explique uniquement par la charge de travail générée par les passages entre le moteur PL/SQL et le moteur SQL

BULK COLLECT

(Select)(Fetch)(execute immediate) # BULK COLLECT Into nom_collection [,nom_collection, #] [LIMIT

nombre_lignes] ;

Ce mot clé demande au moteur SQL de retourner l'ensemble des lignes lues dans une ou plusieurs collections avant de rendre la main au moteur PL/SQL.

Cette fonctionnalité réduit donc considérablement les allers-retours entre les deux moteurs.

Dans le cas d'une instruction FETCH, la clause optionnelle LIMIT permet de restreindre le nombre de lignes ramenées.

Ce nombre de lignes doit être exprimé sous forme de littéral ou de variable

Dans l'exemple suivant, on alimente la collection par groupes de 3 lignes

```
SQL> Declare
  2   TYPE    TYP_TAB_EMP IS TABLE OF EMP.EMPNO%Type ;
  3   Temp_no TYP_TAB_EMP ;
  4   Cursor  C_EMP is Select empno From EMP ;
  5   Pass    Pls_integer := 1 ;
  6  Begin
  7   Open C_EMP ;
  8   Loop
  9     Fetch C_EMP BULK COLLECT into Temp_no LIMIT 3 ;
 10     For i In Temp_no.first..Temp_no.last Loop
 11       dbms_output.put_line( 'Pass ' || to_char(Pass) || ' Empno= ' || Temp_no(i) ) ;
 12     End loop ;
 13     Pass := Pass + 1 ;
 14     Exit When C_EMP%NOTFOUND ;
 15   End Loop ;
 16  End ;
 17  /
Pass 1 Empno= 7369
Pass 1 Empno= 7499
Pass 1 Empno= 7521
Pass 2 Empno= 7566
Pass 2 Empno= 7654
Pass 2 Empno= 7698
Pass 3 Empno= 7782
Pass 3 Empno= 7788
Pass 3 Empno= 7839
Pass 4 Empno= 7844
Pass 4 Empno= 7876
Pass 4 Empno= 7900
Pass 5 Empno= 7902
Pass 5 Empno= 7934

Procédure PL/SQL terminée avec succès.
```

On peut également utiliser le mot clé LIMIT pour effectuer par tranches, des opérations coûteuses pour le ROLLBACK SEGMENT

```
SQL> select empno,sal from emp;

     EMPNO        SAL
---------- ----------
      7369        800
      7499       1600
      7521       1250
      7566       2975
      7654       1250
      7698       2850
```

```
      7782       2450
      7788       3000
      7839       5000
      7844       1500
      7876       1100
      7900        950
      7902       3000
      7934       1300

14 ligne(s) sélectionnée(s).
```

```
SQL> Declare
  2   TYPE    TYP_TAB_EMP IS TABLE OF EMP.EMPNO%Type ;
  3   Temp_no TYP_TAB_EMP ;
  4   Cursor  C_EMP is Select empno From EMP ;
  5  Begin
  6   Open C_EMP ;
  7   Loop
  8     Fetch C_EMP BULK COLLECT into Temp_no LIMIT 3 ;
  9      Forall i In Temp_no.first..Temp_no.last
 10         Update EMP set SAL = Round(SAL * 1.1) Where empno = Temp_no(i) ;
 11      Commit ;
 12      Temp_no.DELETE ;
 13      Exit When C_EMP%NOTFOUND ;
 14   End Loop ;
 15  End ;
 16  /

Procédure PL/SQL terminée avec succès.
```

```
SQL> select empno,sal from emp;

     EMPNO        SAL
---------- ----------
      7369        880
      7499       1760
      7521       1375
      7566       3273
      7654       1375
      7698       3135
      7782       2695
      7788       3300
      7839       5500
      7844       1650
      7876       1210
      7900       1045
      7902       3300
      7934       1430

14 ligne(s) sélectionnée(s).
```

Le mot clé BULK COLLECT peut également être utilisé pour récupérer les résultats d'un ordre DML, lorsqu'il est associé à la clause RETURNING INTO.

```
SQL> Declare
  2    TYPE    TYP_TAB_EMPNO IS TABLE OF EMP.EMPNO%Type ;
  3    TYPE    TYP_TAB_NOM   IS TABLE OF EMP.ENAME%Type ;
  4    Temp_no TYP_TAB_EMPNO ;
  5    Tnoms   TYP_TAB_NOM ;
  6  Begin
  7     Delete From EMP where sal > 3000
  8        RETURNING empno, ename BULK COLLECT INTO Temp_no, Tnoms ;
  9     For i in Temp_no.first..Temp_no.last Loop
 10        dbms_output.put_line( 'Employé viré : ' || To_char( Temp_no(i) ) || ' ' || Tnoms(i) ) ;
 11     End  loop ;
 12  End ;
 13  /
Employé viré : 7566 JONES
Employé viré : 7698 BLAKE
Employé viré : 7788 SCOTT
Employé viré : 7839 KING
Employé viré : 7902 FORD
```

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

Une attention particulière doit être portée sur l'utilisation des méthodes appliquées aux collections, notamment FIRST.

Dans l'exemple suivant, la procédure tombe en erreur car aucune ligne n'est retournée.

Dans ce cas nom_collection.FIRST ne vaut pas zéro mais NULL

```
SQL> Declare
  2    TYPE    TYP_TAB_EMPNO IS TABLE OF EMP.EMPNO%Type ;
  3    TYPE    TYP_TAB_NOM   IS TABLE OF EMP.ENAME%Type ;
  4    Temp_no TYP_TAB_EMPNO ;
  5    Tnoms   TYP_TAB_NOM ;
  6  Begin
  7     Delete From EMP where sal < 100
  8        RETURNING empno, ename BULK COLLECT INTO Temp_no, Tnoms ;
  9     For i in Temp_no.first..Temp_no.last Loop
 10        dbms_output.put_line( 'Employé viré : ' || To_char( Temp_no(i) ) || ' ' || Tnoms(i) ) ;
 11     End  loop ;
 12  End ;
 13  /
Declare
*
ERREUR à la ligne 1 :
ORA-06502: PL/SQL : erreur numérique ou erreur sur une valeur
ORA-06512: à ligne 9
```

Il convient de tester la nullité d'une méthode avant de l'utiliser :

```
SQL> Declare
  2    TYPE    TYP_TAB_EMPNO IS TABLE OF EMP.EMPNO%Type ;
  3    TYPE    TYP_TAB_NOM   IS TABLE OF EMP.ENAME%Type ;
  4    Temp_no TYP_TAB_EMPNO ;
  5    Tnoms   TYP_TAB_NOM ;
  6  Begin
  7     Delete From EMP where sal < 100
  8        RETURNING empno, ename BULK COLLECT INTO Temp_no, Tnoms ;
  9     If Temp_no.first is not null Then
 10       For i in Temp_no.first..Temp_no.last Loop
 11          dbms_output.put_line( 'Employé viré : ' || To_char( Temp_no(i) ) || ' ' || Tnoms(i) ) ;
 12       End  loop ;
 13     End if ;
 14  End ;
 15  /

Procédure PL/SQL terminée avec succès.
```

FORALL

FORALL index IN borne_inférieure..borne_supérieure [SAVE EXCEPTION] ordre_sql

Bien que l'instruction FORALL précise une borne début et une borne fin, il ne peut pas être inclus dans une boucle FOR # LOOP

L'instruction FORALL ne peut pas être utilisée dans le code PL/SQL coté client

L'ordre SQL doit être INSERT, UPDATE ou DELETE en relation avec au moins une collection

Il doit exister des éléments dans la collection pour toutes les valeurs d'indice de l'instruction FORALL

Il n'est pas possible d'exprimer l'indice sous forme d'un calcul

FORALL ne peut traiter qu'un seul ordre SQL à la fois sous peine de générer un message d'erreur :

```
SQL> Declare
  2  TYPE TYP_TAB_NUM IS TABLE OF TEST.B%TYPE INDEX BY PLS_INTEGER ;
  3  TYPE TYP_TAB_CAR IS TABLE OF TEST.A%TYPE INDEX BY PLS_INTEGER ;
  4  tab1 TYP_TAB_NUM ;
  5  tab2 TYP_TAB_CAR ;
  6  begin
  7     For i in 1..1000 Loop
  8       tab1(i) := i ;
  9       tab2(i) := ltrim( to_char( i ) ) ;
 10     End loop ;
 11
 12     Forall i in 1..1000
 13         Insert into TEST( A, B ) Values( tab2(i), tab1(i) ) ;
 14         Delete from TEST where A = tab2(i) ;
 15  End ;
 16  /
       Delete from TEST where A = tab2(i) ;
                                  *
ERREUR à la ligne 14 :
ORA-06550: Ligne 14, colonne 40 :
PLS-00201: l'identificateur 'I' doit être déclaré
ORA-06550: Ligne 14, colonne 35 :
PL/SQL: ORA-00904:  : identificateur non valide
ORA-06550: Ligne 14, colonne 8 :
PL/SQL: SQL Statement ignored
```

La variable d'index I n'est plus connue en sortie de l'instruction FORALL

A partir d'oracle9i, les copies d'informations par blocs peuvent être effectuées directement dans les collections d'enregistrements

```
SQL> Declare
  2    TYPE   TYP_TAB_TEST IS TABLE OF TEST%ROWTYPE ;
  3    tabrec TYP_TAB_TEST ;
  4    CURSOR C_test is select A, B From TEST ;
  5  begin
  6     -- chargement de la collection depuis la table --
  7     Select A, B BULK COLLECT into tabrec From TEST ;
  8
  9     -- insertion de lignes à partir de la collection --
 10     Forall i in tabrec.first..tabrec.last
 11         insert into TEST values tabrec(i) ;
 12
 13     -- mise à jour des données de la collection --
 14     For i in tabrec.first..tabrec.last Loop
 15         tabrec(i).B := tabrec(i).B * 2 ;
 16     End loop ;
 17
 18     -- utilisation du curseur --
 19     Open C_test ;
 20     Fetch C_test BULK COLLECT Into tabrec ;
 21     Close C_test ;
 22
 23  End ;
 24  /

Procédure PL/SQL terminée avec succès.
```

Il n'est pas possible de mettre à jour une ligne complète avec une collection d'enregistrements en conjonction avec

l'instruction FORALL

```
Forall i in tabrec.first..tabrec.last
          update TEST set row =  tabrec(i) where A = tabrec(i).A ; -- INVALIDE
```

Pour cela il faut utiliser une boucle For..Loop

```
    For i in tabrec.first..tabrec.last loop
          update TEST set row =  tabrec(i) where A = tabrec(i).A ;
    End loop ;
```

Ou utiliser des collections simples

```
Declare
  TYPE   TYP_TAB_A IS TABLE OF TEST.A%TYPE ;
  TYPE   TYP_TAB_B IS TABLE OF TEST.B%TYPE ;
  taba   TYP_TAB_A ;
  tabb   TYP_TAB_B ;
  CURSOR C_test is select A, B From TEST ;
Begin
   -- utilisation du curseur --
   Open C_test ;
   Fetch C_test BULK COLLECT Into taba, tabb ;
   Close C_test ;

   -- mise à jour des données de la collection --
   For i in taba.first..taba.last Loop
       tabb(i) := tabb(i) * 2 ;
   End loop ;

   -- mise à jour des lignes de la table --
   Forall i in taba.first..taba.last
          update TEST set B =  tabb(i) where A = taba(i) ;
End ;
```

La gestion des erreurs avec la commande FORALL

Sans gestion particulière des exceptions dans le bloc PL/SQL, toute erreur d'exécution provoque l'annulation de toutes

les instructions réalisées par l'instruction FORALL (ROLLBACK)

En présence d'une section Exception dans le bloc PL/SQL, il sera possible de décider si l'on conserve les modifications valides jusqu'à l'erreur (COMMIT)

ou si l'on annule toute l'instruction FORALL (ROLLBACK)

Il est également possible, lors d'erreur sur une instruction, de sauvegarder l'information concernant l'exception et de poursuivre le processus

Cette fonctionnalité est implémentée par l'ajout du mot clé SAVE EXCEPTION dans l'instruction FORALL

Toutes les exceptions levées en cours d'exécution sont sauvées dans l'attribut %BULK_EXCEPTIONS, qui est une collection d'enregistrements.

Chaque enregistrement est composé de 2 champs :

%BULK_EXCEPTIONS(n).ERROR_INDEX qui contient l'indice de l'itération qui a provoqué l'erreur

%BULK_EXCEPTIONS(n).ERROR_CODE qui contient le code d'erreur

Le nombre total d'exceptions générées par l'instruction FORALL est contenu dans l'attribut SQL%BULK_EXCEPTIONS.COUNT

```
SQL> Declare
  2    TYPE    TYP_TAB IS TABLE OF Number ;
  3    tab     TYP_TAB := TYP_TAB( 2, 0, 1, 3, 0, 4, 5 ) ;
  4    nb_err  Pls_integer ;
  5  Begin
  6     Forall i in tab.first..tab.last SAVE EXCEPTIONS
  7         delete from TEST where B = 5 / tab(i) ;
  8  Exception
  9    When others then
 10      nb_err := SQL%BULK_EXCEPTIONS.COUNT ;
 11      dbms_output.put_line( to_char( nb_err ) || ' Erreurs ' ) ;
 12      For i in 1..nb_err Loop
 13         dbms_output.put_line( 'Indice ' || to_char( SQL%BULK_EXCEPTIONS(i).ERROR_INDEX ) || '
Erreur Oracle : '
 14         || to_char( SQL%BULK_EXCEPTIONS(i).ERROR_CODE ) ) ;
 15      End loop ;
 16  End ;
 17  /
2 Erreurs
Indice 2 Erreur Oracle :  1476
Indice 5 Erreur Oracle :  1476

Procédure PL/SQL terminée avec succès.
```

L'attribut %BULK_ROWCOUNT

En plus des attributs de curseur %FOUND, %NOTFOUND, %ISOPEN et %ROWCOUNT, le curseur implicite SQL dispose d'un attribut spécifique

à l'instruction FORALL : %BULK_ROWCOUNT

Il s'agit d'une collection de type INDEX BY TABLE pour laquelle l'élément d'indice n contient le nombre de lignes affectées par l'exécution de l'ordre SQL numéro n

Si aucune ligne n'est affectée par l'instruction numéro n alors l'attribut SQL%BULK_ROWCOUNT(n) vaut 0

```
SQL> Declare
  2    TYPE   TYP_TAB_TEST IS TABLE OF TEST%ROWTYPE ;
  3    TYPE   TYP_TAB_A IS TABLE OF TEST.A%TYPE ;
  4    TYPE   TYP_TAB_B IS TABLE OF TEST.B%TYPE ;
  5    tabrec TYP_TAB_TEST ;
  6    taba   TYP_TAB_A ;
  7    tabb   TYP_TAB_B ;
  8    total  Pls_integer := 0 ;
  9    CURSOR C_test is select A, B From TEST ;
 10  begin
 11     -- chargement de la collection depuis la table --
```

```
 12     Select A, B BULK COLLECT into tabrec From TEST ;
 13
 14     -- insertion de lignes à partir de la collection --
 15     Forall i in tabrec.first..tabrec.last
 16         insert into TEST values tabrec(i) ;
 17
 18     For i in tabrec.first..tabrec.last Loop
 19         total := total + SQL%BULK_ROWCOUNT(i) ;
 20     End loop ;
 21
 22     dbms_output.put_line('Total insertions : ' || to_char( total) ) ;
 23
 24     total := 0 ;
 25     -- mise à jour d'une ligne précise non permise avec les collections d'enregistrements --
 26     For i in tabrec.first..tabrec.last loop
 27       update TEST set row =  tabrec(i) where A = tabrec(i).A ;
 28     End loop ;
 29
 30     For i in tabrec.first..tabrec.last Loop
 31         total := total + SQL%BULK_ROWCOUNT(i) ;
 32     End loop ;
 33
 34     dbms_output.put_line('Total mises à jour : ' || to_char( total) ) ;
 35
 36   End ;
 37   /
Total insertions : 20
Total mises à jour : 20

Procédure PL/SQL terminée avec succès.
```

5.6 - Les collections et enregistrements en paramètres des procédures et fonctions

Le passage de ces types d'objets se fait de façon identique qu'avec les types scalaires.

Soit le package suivant :

```
SQL> CREATE OR REPLACE PACKAGE PKG_TEST IS
  2
  3    TYPE TYP_TAB_EMP  IS TABLE OF EMP%ROWTYPE ;
  4    TYPE TYP_TAB_EMP2 IS TABLE OF EMP%ROWTYPE INDEX BY PLS_INTEGER ;
  5
  6    PROCEDURE Affiche_lignes ( Temp IN TYP_TAB_EMP ) ;
  7
  8    PROCEDURE Affiche_ligne ( Temp IN OUT EMP%ROWTYPE ) ;
  9
 10    PROCEDURE Affiche_ligne_nocopy ( Temp IN OUT NOCOPY EMP%ROWTYPE ) ;
 11
 12    PROCEDURE Modifie_lignes ( Temp IN OUT TYP_TAB_EMP ) ;
 13
 14    PROCEDURE Modifie_lignes_nocopy ( Temp IN OUT NOCOPY TYP_TAB_EMP ) ;
 15
 16    PROCEDURE Modifie_lignes2 ( Temp IN OUT TYP_TAB_EMP2 ) ;
 17
 18    PROCEDURE Modifie_lignes_nocopy2 ( Temp IN OUT NOCOPY TYP_TAB_EMP2 ) ;
 19
 20    FUNCTION Affiche_lignes Return TYP_TAB_EMP ;
 21
 22  END;
 23  /

Package créé.

SQL> CREATE OR REPLACE PACKAGE BODY PKG_TEST IS
  2
  3    GN$Lig pls_integer := 0 ;
  4
  5    PROCEDURE Affiche_lignes ( Temp IN TYP_TAB_EMP )
  6    IS
  7    Begin
```

```
 8     GN$Lig := 0 ;
 9     For i IN Temp.first..Temp.last Loop
10       GN$Lig := GN$Lig + 1 ;
11       If GN$Lig  <= 10 Then
12          dbms_output.put_line( Rpad( Temp(i).ename, 25 ) || ' --> ' || To_char( Temp(i).sal ) )
   ;
13       End if ;
14     End loop ;
15   End Affiche_lignes ;
16
17   PROCEDURE Affiche_ligne ( Temp IN OUT EMP%ROWTYPE )
18   IS
19   Begin
20     GN$Lig := GN$Lig + 1 ;
21     Temp.sal := Temp.sal * 1.1 ;
22     If GN$Lig  <= 10 Then
23       dbms_output.put_line( Rpad( Temp.ename, 25 ) || ' --> ' || To_char( Temp.sal ) ) ;
24     End if ;
25   End Affiche_ligne ;
26
27   PROCEDURE Affiche_ligne_nocopy ( Temp IN OUT NOCOPY EMP%ROWTYPE )
28   IS
29   Begin
30     GN$Lig := GN$Lig + 1 ;
31     Temp.sal := Temp.sal * 1.1 ;
32     If GN$Lig  <= 10 Then
33       dbms_output.put_line( Rpad( Temp.ename, 25 ) || ' --> ' || To_char( Temp.sal ) ) ;
34     End if ;
35   End Affiche_ligne_nocopy ;
36
37   PROCEDURE Modifie_lignes ( Temp IN OUT TYP_TAB_EMP )
38   IS
39   Begin
40    GN$Lig := 0 ;
41    For i IN Temp.first..Temp.last Loop
42      Temp(i).sal := Temp(i).sal * 1.1 ;
43      Affiche_ligne( Temp(i) ) ;
44    End loop ;
45   End Modifie_lignes ;
46
47   PROCEDURE Modifie_lignes_nocopy ( Temp IN OUT NOCOPY TYP_TAB_EMP )
48   IS
49   Begin
50    GN$Lig := 0 ;
51    For i IN Temp.first..Temp.last Loop
52      Temp(i).sal := Temp(i).sal * 1.1 ;
53      Affiche_ligne_nocopy( Temp(i) ) ;
54    End loop ;
55   End Modifie_lignes_nocopy ;
56
57   PROCEDURE Modifie_lignes2 ( Temp IN OUT TYP_TAB_EMP2 )
58   IS
59   Begin
60    GN$Lig := 0 ;
61    For i IN Temp.first..Temp.last Loop
62      Temp(i).sal := Temp(i).sal * 1.1 ;
63      Affiche_ligne( Temp(i) ) ;
64      --dbms_output.put_line( Rpad( Temp(i).ename, 25 ) || ' --> ' || To_char( Temp(i).sal ) ) ;
65    End loop ;
66   End Modifie_lignes2 ;
67
68   PROCEDURE Modifie_lignes_nocopy2 ( Temp IN OUT NOCOPY TYP_TAB_EMP2 )
69   IS
70   Begin
71    GN$Lig := 0 ;
72    For i IN Temp.first..Temp.last Loop
73      Temp(i).sal := Temp(i).sal * 1.1 ;
74      Affiche_ligne_nocopy( Temp(i) ) ;
75      --dbms_output.put_line( Rpad( Temp(i).ename, 25 ) || ' --> ' || To_char( Temp(i).sal ) ) ;
76    End loop ;
77   End Modifie_lignes_nocopy2 ;
78
79
80   FUNCTION Affiche_lignes Return TYP_TAB_EMP
81   IS
82     Tlignes PKG_TEST.TYP_TAB_EMP ;
83     Cursor C_EMP is Select * From EMP ;
84   Begin
85     Open C_EMP ;
```

```
 86      Fetch C_EMP BULK COLLECT into Tlignes ;
 87      Close C_EMP ;
 88      Return( Tlignes ) ;
 89    End ;
 90
 91  END;
 92  /

Corps de package créé.
```

Collections en argument de procédure

Nous allons maintenant écrire un bloc PL/SQL anonyme qui utilise la fonction Affiche_lignes()

```
SQL> Declare
  2    Tlignes PKG_TEST.TYP_TAB_EMP ;
  3    Cursor  C_EMP is Select * From EMP ;
  4  Begin
  5
  6   Open  C_EMP ;
  7   Fetch C_EMP BULK COLLECT into Tlignes ;
  8   Close C_EMP ;
  9
 10   PKG_TEST.Affiche_lignes( Tlignes ) ;
 11
 12  End ;
 13  /
SMITH                     --> 880
ALLEN                     --> 1760
WARD                      --> 1375
JONES                     --> 3273
MARTIN                    --> 1375
BLAKE                     --> 3135
CLARK                     --> 2695
SCOTT                     --> 3300
KING                      --> 5500
TURNER                    --> 1650
ADAMS                     --> 1210
JAMES                     --> 1045
FORD                      --> 3300
MILLER                    --> 1430

Procédure PL/SQL terminée avec succès.
```

Une variable Tlignes est déclarée avec le type PKG_TEST.TYP_TAB_EMP

qui est une collection d'enregistrements de type EMP%ROWTYPE

Cette collection est alimentée par le curseur C_EMP et transmise à la procédure qui en affiche une partie du contenu.

Vous pouvez également ne transmettre qu'une ligne de la collection

```
SQL> Declare
  2    Tlignes PKG_TEST.TYP_TAB_EMP ;
  3    Cursor  C_EMP is Select * From EMP ;
  4  Begin
  5   Open  C_EMP ;
  6   Fetch C_EMP BULK COLLECT into Tlignes ;
  7   Close C_EMP ;
  8   PKG_TEST.Affiche_ligne( Tlignes(2) ) ;
  9  End ;
 10  /
```

```
ALLEN                     --> 1760

Procédure PL/SQL terminée avec succès.
```

La procédure Affiche_ligne() reçoit en argument une variable de type EMP%ROWTYPE.

Dans l'appel, seule la deuxième ligne de la collection lui est transmise.

Pour pouvoir en modifier la valeurs des éléments, la collection doit être transmise en mode IN OUT

```
SQL> Declare
  2    Tlignes PKG_TEST.TYP_TAB_EMP ;
  3    Cursor  C_EMP is Select * From EMP ;
  4  Begin
  5
  6   Open  C_EMP ;
  7   Fetch C_EMP BULK COLLECT into Tlignes ;
  8   Close C_EMP ;
  9
 10   PKG_TEST.Modifie_lignes( Tlignes ) ;
 11
 12  End ;
 13  /
SMITH                     --> 968
ALLEN                     --> 1936
WARD                      --> 1512,5
JONES                     --> 3600,3
MARTIN                    --> 1512,5
BLAKE                     --> 3448,5
CLARK                     --> 2964,5
SCOTT                     --> 3630
KING                      --> 6050
TURNER                    --> 1815
ADAMS                     --> 1331
JAMES                     --> 1149,5
FORD                      --> 3630
MILLER                    --> 1573

Procédure PL/SQL terminée avec succès.
```

Lors du passage de collections ou éléments de collection à des procédures ou fonctions, les arguments sont passés par copie.

Une copie de l'objet est faite dans un espace mémoire particulier.

Dans le cas où des collections très volumineuses sont passées en argument, où de nombreux appels sont effectuées avec des collections

ou éléments de collection, il est préférable de passer les arguments en mode NOCOPY.

En effet dans ce mode, aucun espace mémoire supplémentaire n'est alloué puisque la variable est passée par référence (Le pointeur sur la variable)

Voir la procédure PKG_TEST.Affiche_ligne_nocopy()

Collections en retour de fonction

Soit la fonction PKG_TEST.Affiche_lignes

```
FUNCTION Affiche_lignes Return TYP_TAB_EMP
IS
  Tlignes PKG_TEST.TYP_TAB_EMP ;
  Cursor  C_EMP is Select * From EMP ;
Begin
  Open  C_EMP ;
  Fetch C_EMP BULK COLLECT into Tlignes ;
  Close C_EMP ;
  Return( Tlignes ) ;
End ;
```

Appelée par le bloc PL/SQL suivant :

```
SQL> Declare
  2    Tlignes PKG_TEST.TYP_TAB_EMP ;
  3  Begin
  4    Tlignes := PKG_TEST.Affiche_lignes ;
  5    For i IN Tlignes.first..Tlignes.last Loop
  6       dbms_output.put_line( Rpad( Tlignes(i).ename, 25 ) || ' --> ' || To_char( Tlignes(i).sal )
) ;
  7    End loop ;
  8  End ;
  9  /
SMITH                     --> 880
ALLEN                     --> 1760
WARD                      --> 1375
JONES                     --> 3273
MARTIN                    --> 1375
BLAKE                     --> 3135
CLARK                     --> 2695
SCOTT                     --> 3300
KING                      --> 5500
TURNER                    --> 1650
ADAMS                     --> 1210
JAMES                     --> 1045
FORD                      --> 3300
MILLER                    --> 1430

Procédure PL/SQL terminée avec succès.
```

La fonction Affiche_lignes retourne une collection de type PKG_TEST.TYP_TAB_EMP,

ouvre un curseur sur la table EMP pour remplir la collection et la retourne au bloc PL/SQL appelant qui peut en afficher les valeurs

Enregistrements

Lorsque l'on veut appeler une fonction retournant un type RECORD, il faut respecter la syntaxe suivante pour référencer un champ de l'enregistrement :

Nom_fonction(argument).nom_champ

```
SQL> Declare
  2    TYPE Emp_rec IS RECORD
  3    (
  4      empno  emp.empno%type,
  5      salaire  emp.sal%type
  6    );
```

```
    7    salaire emp.sal%type ;
    8    Function xx(ln$numemp in emp.EMPNO%type) return Emp_rec
    9    is
   10      emp_info EMp_rec ;
   11    Begin
   12      Select empno, sal into emp_info.empno, emp_info.salaire
   13      from emp where empno = ln$numemp ;
   14      return emp_info ;
   15    End ;
   16  Begin
   17    salaire := xx(7499).salaire ;
   18    dbms_output.put_line( 'Salaire de 7499 : ' || to_char(salaire) ) ;
   19  End  ;
   20  /
Salaire de 7499 : 1760

Procédure PL/SQL terminée avec succès.
```

6 - Les déclencheurs

Un déclencheur est un bloc PL/SQL associé à une vue ou une table, qui s'exécutera lorsqu'une instruction du langage

de manipulation de données (DML) sera exécutée

L'avantage principal du déclencheur réside dans le fait que le code est centralisé dans la base de données, et se déclenchera quel que soit l'outil utilisé pour mettre à jour ces données, donnant ainsi l'assurance qu'une utilisation d'un ordre DML depuis Sql*Plus, Forms ou n'importe quelle application tierce procurera un résultat identique sur les données

l'inconvénient principal du déclencheur réside dans le fait que son exécution utilise des ressources qui peuvent augmenter sensiblement les temps de traitement, notamment lors de modifications massives apportées sur une table

Un déclencheur s'exécute dans le cadre d'une transaction. Il ne peut donc pas contenir d'instruction COMMIT ou ROLLBACK ou toute instruction générant une fin de transaction implicite (ordre DDL)

Les ordres SQL (SELECT, INSERT, UPDATE, DELETE) contenus dans le bloc PL/SQL et qui se réfèrent à la table sur laquelle s'exécute le déclencheur peuvent générer l'exception ORA-04091 TABLE IS MUTATING

(pour une explication détaillée du problème des tables en mutation, référez-vous à l'article Résolution du problème de la table mutante (ora-04091) par Pomalaix

Le bloc PL/SQL qui constitue le trigger peut être exécuté avant ou après la vérification des contraintes d'intégrité

Il peut être exécuté pour chaque ligne affectée par l'ordre DML ou bien une seule fois pour la commande

Seules les colonnes de la ligne en cours de modification sont accessibles par l'intermédiaire de 2 variables de type enregistrement **OLD** et **NEW**

OLD représente la valeur avant modification

OLD n'est renseignée que pour les ordres DELETE et UPDATE. Elle n'a aucune signification pour un ordre INSERT, puisqu'aucune ancienne valeur n'existe

NEW représente la nouvelle valeur

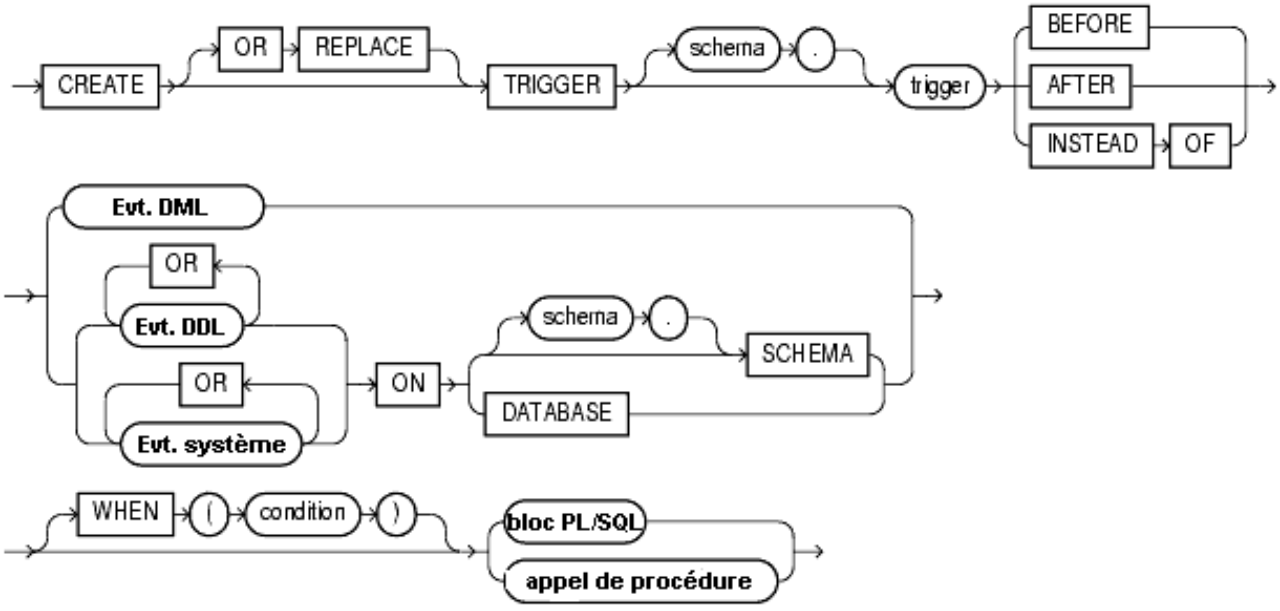
NEW n'est renseignée que pour les ordres INSERT et UPDATE. Elle n'a aucune signification pour un ordre DELETE, puisqu'aucune nouvelle valeur n'existe

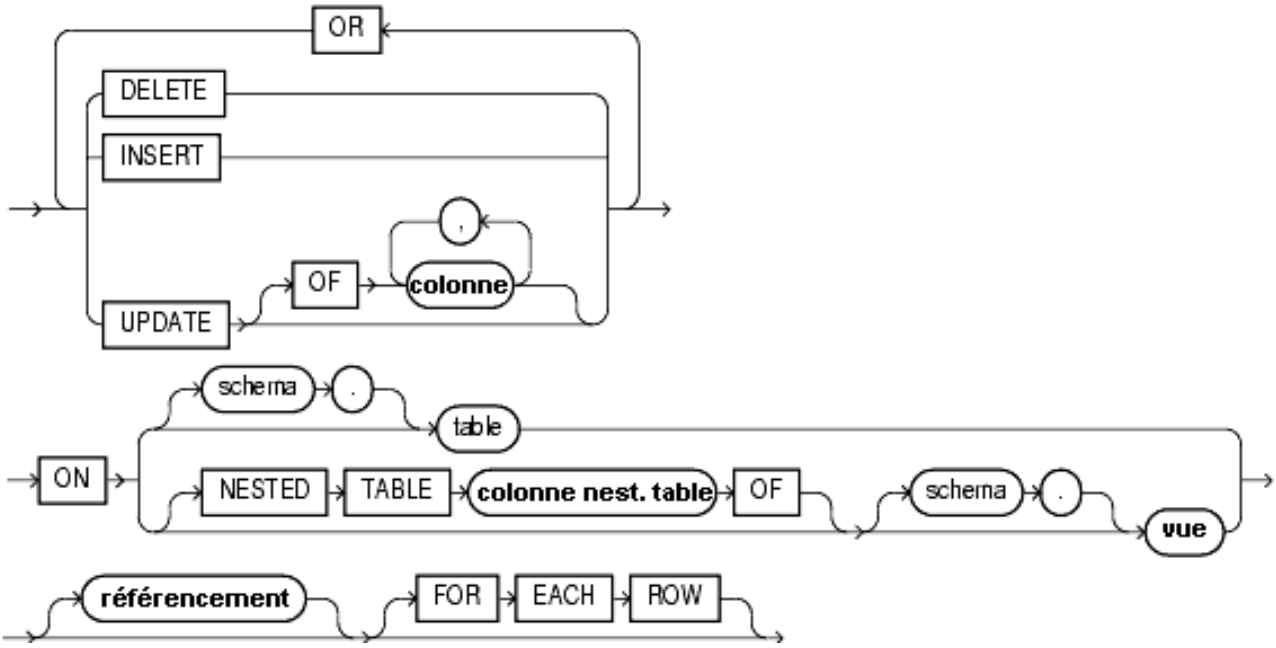
Ces deux variables peuvent être utilisées dans la clause **WHEN** du déclencheur et dans la section exécutable

Dans cette section, elle doivent être préfixées comme des variables hôtes avec l'opérateur **:**

Les noms de ces deux variables sont fixés par défaut, mais il est possible de les modifier en précisant les nouveaux noms dans la clause **REFERENCING**

REFERENCING OLD AS nouveau_nom NEW AS nouveau_nom





Evt. DML représente l'évènement INSERT ou UPDATE ou DELETE

Evt. DDL représente un évènement utilisateur

Evt. système représente un évènement système

colonne nest. table représente le nom d'une colonne de table imbriquée

référencement représente le renommage des valeurs OLD et NEW

Dans le cas d'un déclencheur **BEFORE UPDATE** ou **AFTER UPDATE**, la clause **OF** peut être ajoutée après le mot clé **UPDATE** pour spécifier la liste des colonnes modifiées.

Cela permet de restreindre l'activation du déclencheurs sur les seules colonnes visées.

Le mot clé **WHEN(condition)** permet également de restreindre le champs d'activation du déclencheur en ajoutant une clause restrictive

6.1 - Les déclencheurs sur TABLE

Créons un déclencheur très basique qui ne fait qu'afficher le numéro et le nom d'un employé que l'on veut supprimer de la table EMP

```
SQL> CREATE OR REPLACE TRIGGER TRG_BDR_EMP
  2     BEFORE DELETE -- avant supression
  3     ON EMP        -- sur la table EMP
  4     FOR EACH ROW  -- pour chaque ligne
```

```
  5  Declare
  6    LC$Chaine VARCHAR2(100);
  7  Begin
  8    dbms_output.put_line( 'Suppression de l''employé n° ' || To_char( :OLD.empno )
  9       || ' -> ' || :OLD.ename ) ;
 10  End ;
 11  /

Déclencheur créé.
```

Supprimons maintenant un employé

```
SQL> set serveroutput on
SQL> delete from emp where empno = 7369
  2  /
Suppression de l'employé n° 7369 -> SMITH

1 ligne supprimée.

SQL> rollback;

Annulation (rollback) effectuée.
```

La DRH annonce que désormais, tout nouvel employé devra avoir un numéro supérieur ou égal à 10000

Il faut donc interdire toute insertion qui ne reflète pas cette nouvelle directive

```
SQL> CREATE OR REPLACE TRIGGER TRG_BIR_EMP
  2     BEFORE INSERT -- avant insertion
  3     ON EMP        -- sur la table EMP
  4     FOR EACH ROW  -- pour chaque ligne
  5  Begin
  6    If :NEW.empno < 10000 Then
  7        RAISE_APPLICATION_ERROR ( -20010, 'Numéro employé inférieur à 10000' ) ;
  8    End if ;
  9  End ;
 10  /

Déclencheur créé.
```

Tentons d'insérer un nouvel employé avec le numéro 9999

```
SQL>  insert into emp (empno, ename, job) values( 9999, 'Burger', 'CLERK' ) ;
 insert into emp (empno, ename, job) values( 9999, 'Burger', 'CLERK' )
             *
ERREUR à la ligne 1 :
ORA-20010: Numéro employé inférieur à 10000
ORA-06512: à "SCOTT.TRG_BIR_EMP", ligne 3
ORA-04088: erreur lors d'exécution du déclencheur 'SCOTT.TRG_BIR_EMP'
```

L'ordre d'insertion est rejeté

Il est possible de gérer dans le même déclencheur des ordres DML différents en combinant les termes de la clause BEFORE avec le mot clé **OR**

```
SQL> CREATE OR REPLACE TRIGGER TRG_BIUDR_EMP
  2     BEFORE INSERT OR UPDATE OR DELETE-- avant insertion, modification ou suppression
  3     ON EMP        -- sur la table EMP
  4     FOR EACH ROW  -- pour chaque ligne
  5  Begin
  6    If INSERTING Then
```

```
  7       dbms_output.put_line( 'Insertion dans la table EMP' ) ;
  8    End if ;
  9    If UPDATING Then
 10       dbms_output.put_line( 'Mise à jour de la table EMP' ) ;
 11    End if ;
 12    If DELETING Then
 13       dbms_output.put_line( 'Suppression dans la table EMP' ) ;
 14    End if ;
 15 End ;
 16 /

Déclencheur créé.
```

```
SQL> DROP TRIGGER TRG_BIR_EMP ;

Déclencheur supprimé.

SQL>  insert into emp (empno, ename, job) values( 9993, 'Burger', 'CLERK') ;
Insertion dans la table EMP

1 ligne créée.

SQL> update emp set sal = 5000 where empno = 9993 ;
Mise à jour de la table EMP

1 ligne mise à jour.

SQL> delete from emp where empno = 9993 ;
Suppression dans la table EMP
Suppression de l'employé n° 9993 -> Burger

1 ligne supprimée.

SQL> rollback;

Annulation (rollback) effectuée.
```

Notez au passage que dans l'exemple de la suppression, les deux déclencheurs de type BEFORE DELETE ont été exécutés

6.2 - Les déclencheurs sur VUE

La syntaxe d'un déclencheur sur vue est identique à celle du déclencheur sur table, à la différence que la clause INSTEAD OF est ajoutée

Ce type de déclencheur est particulier dans la mesure ou son exécution remplace celle de la commande DML à laquelle il est associé

Ce type de déclencheur n'est définissable que sur les vues et lui seul peut être mis en place sur les vues

Nous mettons à la disposition de certains utilisateurs une vue permettant de sélectionner les employés qui ont le job CLERK

```
SQL> CREATE OR REPLACE VIEW VW_EMP_CLERK AS
  2    Select empno "Numéro", ename "Nom", deptno "Dept.", sal "Salaire"
  3    From   EMP
```

```
  4    Where  JOB = 'CLERK';

Vue créée.

SQL> select * from VW_EMP_CLERK ;

    Numéro Nom             Dept.    Salaire
---------- ---------- ---------- ----------
      7369 SMITH              20        880
      7876 ADAMS              20       1210
      7900 JAMES              30       1045
      7934 MILLER             10       1430
      9991 Dupontont
      9992 Duboudin

6 ligne(s) sélectionnée(s).
```

A travers cette vue, ces utilisateurs peuvent insérer des lignes

```
SQL> Insert into VW_EMP_CLERK values( 9994, 'Schmoll', 20, 2500 ) ;
Insertion dans la table EMP

1 ligne créée.
```

Cependant, ils ne peuvent pas voir leurs insertions car la colonne job (inutile dans ce cas) ne fait pas partie de la vue et donc de l'insertion !

```
SQL>  select * from VW_EMP_CLERK ;

    Numéro Nom             Dept.    Salaire
---------- ---------- ---------- ----------
      7369 SMITH              20        880
      7876 ADAMS              20       1210
      7900 JAMES              30       1045
      7934 MILLER             10       1430
      9991 Dupontont
      9992 Duboudin

6 ligne(s) sélectionnée(s).
```

Nous allons donc créer un déclencheur sur vue qui va résoudre ce problème

```
SQL> CREATE OR REPLACE TRIGGER TRG_BIR_VW_EMP_CLERK
  2    INSTEAD OF INSERT -- à la place de l'insertion
  3    ON VW_EMP_CLERK   -- sur la vue VW_EMP_CLERK
  4    FOR EACH ROW      -- pour chaque ligne
  5  Begin
  6    Insert into EMP ( empno, ename, deptno, sal, job ) -- on valorise la colonne JOB
  7    Values (:NEW."Numéro", :NEW."Nom", :NEW."Dept.", :NEW."Salaire", 'CLERK' ) ;
  8  End ;
  9  /

Déclencheur créé.
```

L'utilisateur peut désormais visualiser ses insertions

```
SQL> Insert into VW_EMP_CLERK values( 9994, 'Schmoll', 20, 2500 ) ;
Insertion dans la table EMP

1 ligne créée.

SQL>  select * from VW_EMP_CLERK ;

    Numéro Nom             Dept.    Salaire
---------- ---------- ---------- ----------
      7369 SMITH              20        880
      7876 ADAMS              20       1210
      7900 JAMES              30       1045
      7934 MILLER             10       1430
```

```
        9991 Dupontont
        9992 Duboudin
        9994 Schmoll            20       2500

  7 ligne(s) sélectionnée(s).
```

6.3 - Les déclencheurs sur évènements système ou utilisateur

depuis la version Oracle8i, il est désormais possible d'utiliser des déclencheurs pour suivre les changements d'état du système ainsi que les connexions/déconnexions utilisateur et la surveillance des ordres DDL et DML

Lors de l'écriture de ces déclencheurs, il est possible d'utiliser des attributs pour identifier précisément l'origine des évènements et adapter les traitements en conséquence

6.3.1 - Les attributs

- **ora_client_ip_adress**
- **ora_database_name**
- **ora_des_encrypted_password**
- **ora_dict_obj_name**
- **ora_dict_obj_name_list**
- **ora_dict_obj_owner**
- **ora_dict_obj_owner_list**
- **ora_dict_obj_type**
- **ora_grantee**
- **ora_instance_num**
- **ora_is_alter_column**
- **ora_is_creating_nested_table**
- **ora_is_drop_column**
- **ora_is_servererror**
- **ora_login_user**
- **ora_privileges**
- **ora_revokee**
- **ora_server_error**
- **ora_sysevent**
- **ora_with_grant_option**

6.3.2 - Les évènements système

CREATE TRIGGER nom_déclencheur {BEFORE|AFTER} évènement_système ON{DATABASE|SCHEMA} bloc PL/SQL

- **STARTUP**
- **SHUTDOWN**

- **SERVERERROR**

6.3.3 - Les évènements utilisateur

CREATE TRIGGER nom_déclencheur {BEFORE|AFTER} évènement_utilisateur ON{DATABASE|SCHEMA} bloc PL/SQL

- **LOGON**
- **LOGOFF**
- **CREATE**
- **ALTER**
- **DROP**
- **ANALYZE**
- **ASSOCIATE STATISTICS**
- **AUDIT**
- **NOAUDIT**
- **COMMENT**
- **DDL**
- **DISSOCIATE STATISTICS**
- **GRANT**
- **RENAME**
- **REVOKE**
- **TRUNCATE**

6.4 - Maintenance des déclencheurs

Activation/désactivation d'un déclencheur.

Il est possible de désactiver un déclencheur avec la commande suivante

ALTER TRIGGER nom_déclencheur DISABLE

et de l'activer avec la commande suivante

ALTER TRIGGER nom_déclencheur ENABLE

De la même façon, on peut désactiver tous les déclencheurs définis sur une table

ALTER TABLE nom_table DISABLE ALL TRIGGERS

et de les activer avec la commande suivante

ALTER TABLE nom_table ENABLE ALL TRIGGERS

Les informations sur les déclencheurs sont visibles à travers les vues du dictionnaire de données

USER_TRIGGERS pour les déclencheurs appartenant au schéma

ALL_TRIGGERS pour les déclencheurs appartenant aux schémas accessibles

DBA_TRIGGERS pour les déclencheurs appartenant à tous les schémas

La colonne BASE_OBJECT_TYPE permet de savoir si le déclencheur est basé sur une table, une vue, un schéma ou la totalité de la base

La colonne TRIGGER_TYPE permet de savoir s'il s'agit d'un déclencheur BEFORE, AFTER ou INSTEAD OF

si son mode est FOR EACH ROW ou non

s'il s'agit d'un déclencheur évènementiel ou non

La colonne TRIGGERING_EVENT permet de connaître l'évèvement concerné par le déclencheur

La colonne TRIGGER_BODY contient le code du bloc PL/SQL

7 - Le paquetage DBMS_OUTPUT

Les procédures de ce paquetage vous permettent d'écrire des lignes dans un tampon depuis un bloc PL/SQL anonyme, une procédure ou un déclencheur.

Le contenu de ce tampon est affiché à l'écran lorsque le sous-programme ou le déclencheur est terminé.

L'utilité principale est d'afficher à l'écran des informations de trace ou de débogage

La taille maximum du tampon est de un million de caractères

La taille maximum d'une ligne est de 255 caractères

La capacité maximum d'une ligne étant de 255 caractères, voyez la procédure DEBUG, présentée au chapitre Fonctions et procédure, qui permet de s'affranchir de cette limitation

Fonctions et procédures du paquetage

- **DBMS_OUTPUT.ENABLE (taille_tampon IN INTEGER DEFAULT 20000)**

Cette procédure permet d'initialiser le tampon d'écriture et d'accepter les commandes de lecture et d'écriture dans ce tampon.

taille_tampon représente la taille maximum en octets allouée au tampon.

Les valeurs doivent être indiquées dans une plage de valeur allant de 2000 (minimum) et 1 million (maximum). Sa valeur par défaut est 20000.

Exceptions générées

ORA-20000: Buffer overflow, limit of (buffer_limit) bytes.

ORU-10027:

- **DBMS_OUTPUT.DISABLE**

Cette procédure désactive les appels de lecture et écriture dans le tampon et purge ce dernier.

Procédures d'écriture dans le tampon

- **DBMS_OUTPUT.PUT (item IN NUMBER)**
- **DBMS_OUTPUT.PUT (item IN VARCHAR2)**
- **DBMS_OUTPUT.PUT (item IN DATE)**

- **DBMS_OUTPUT.PUT_LINE (item IN NUMBER)**
- **DBMS_OUTPUT.PUT_LINE (item IN VARCHAR2)**
- **DBMS_OUTPUT.PUT_LINE (item IN DATE)**

- **DBMS_OUTPUT.NEW_LINE**

item représente un littéral ou une variable

DBMS_OUTPUT.PUT permet d'ajouter des informations dans la ligne en cours du tampon

DBMS_OUTPUT.PUT_LINE permet de générer une ligne entière dans le tampon. Un caractère fin de ligne est automatiquement ajouté en fin de ligne

DBMS_OUTPUT.NEW_LINE permet d'ajouter au tampon un caractère fin de ligne

Exceptions générées

ORA-20000 Buffer overflow, limit of (buf_limit) bytes.

ORU-10027:

ORA-20000 Line length overflow, limit of 255 bytes per line.

ORU-10028:

Procédures de lecture du tampon

- **DBMS_OUTPUT.GET_LINE (ligne OUT VARCHAR2, statut OUT INTEGER)**

ligne représente une ligne du tampon retournée, à l'exclusion du caractère NL final

La longueur maximum de ligne est de 255 caractères

statut retourne la valeur 0 si la lecture s'est déroulée correctement et 1 s'il n'y a plus de lignes dans le tampon

- **DBMS_OUTPUT.GET_LINES (lignes OUT tableau_char, nombre_lignes IN OUT INTEGER)**

tableau_char représente un tableau de chaînes de caractères. La taille maximum de chaque ligne est de 255 caractères

nombre_lignes en entrée représente le nombre de lignes du tampon que l'on souhaite lire.

En sortie, il indique le nombre de lignes effectivement lues depuis le tampon

Exemple

```
SQL> set serveroutput on
SQL> Declare
  2     CURSOR C_EMP Is
  3     Select * From EMP ;
  4  Begin
  5     -- Ouverture du tampon --
  6     DBMS_OUTPUT.ENABLE( 1000000 ) ;
  7     -- Boucle sur la table EMP --
  8     For C in C_EMP Loop
  9        -- Ligne avec caractère fin de ligne --
 10        DBMS_OUTPUT.PUT_LINE( 'Employé -> ' || C.EMPNO ) ;
 11        -- Ligne composée de différents champs --
 12        DBMS_OUTPUT.PUT( ' ' || 'Nom=' || C.ENAME ) ;
 13        DBMS_OUTPUT.PUT( ' ' || 'Job=' || C.JOB ) ;
 14        DBMS_OUTPUT.PUT( ' ' || 'Dept=' || C.DEPTNO ) ;
 15        DBMS_OUTPUT.PUT( ' ' || 'Salaire=' || C.SAL ) ;
 16        -- Saut de ligne --
 17        DBMS_OUTPUT.NEW_LINE ;
 18     End loop ;
 19  End ;
 20  /
Employé -> 7369
```

```
Nom=SMITH Job=CLERK Dept=20 Salaire=880
Employé -> 7499
Nom=ALLEN Job=SALESMAN Dept=30 Salaire=1936
Employé -> 7521
Nom=WARD Job=SALESMAN Dept=30 Salaire=1375
Employé -> 7566
Nom=JONES Job=MANAGER Dept=20 Salaire=3273
...
Employé -> 9994
Nom=Schmoll Job=CLERK Dept=20 Salaire=2500

Procédure PL/SQL terminée avec succès.

SQL>
```

Notez, sous Sql*Plus la commande SET SERVEROUTPUT ON qui permet de rendre actives les fonctions du paquetage DBMS_OUTPUT

8 - Le paquetage UTL_FILE

Les procédures et fonctions de ce paquetage vous permettent de lire et écrire dans des fichiers texte situés dans les répertoires du système d'exploitation.

Elles sont exécutées par le noyau Oracle, donc sur la partie serveur.

Pour bénéficier des mêmes fonctionnalités sur la partie cliente, il faut utiliser le paquetage TEXT_IO.

Sous Oracle 9i, l'accès à ces fichiers est restreint aux **répertoires(DIRECTORIES)** déclarés au niveau du serveur.

Dans les version antérieures, la description des répertoires autorisés était spécifiée dans le paramètre UTL_FILE_DIR du fichier d'initialisation INIT.ORA, dont toute modification imposait évidement un arrêt et relance de la base

Pour créer un répertoire, il faut avoir le privilège CREATE DIRECTORY, accordé par défaut aux seuls schémas SYS et SYSTEM

Pour accéder en lecture aux répertoires déclarés dans les répertoires, l'utilisateur doit avoir le droit READ sur le ou les répertoires

Pour accéder en écriture aux répertoires déclarés dans les répertoires, l'utilisateur doit avoir le droit WRITE sur le ou les répertoires

Créons sous l'utilisateur SYSTEM deux répertoires pour les fichiers en entrée et ceux en sortie

```
SQL> CREATE DIRECTORY FICHIERS_IN AS 'd:\fichiers\in' ;
Répertoire créé.
SQL> GRANT READ ON DIRECTORY FICHIERS_IN TO PUBLIC ;
Autorisation de privilèges (GRANT) acceptée.
SQL> CREATE DIRECTORY FICHIERS_OUT AS 'd:\fichiers\out' ;
Répertoire créé.
SQL> GRANT READ, WRITE ON DIRECTORY FICHIERS_OUT TO PUBLIC ;
Autorisation de privilèges (GRANT) acceptée.
```

Le répertoire contenant les fichiers en entrée se voit attribuer le droit de lecture (READ) pour tous les utilisateurs

Le répertoire contenant les fichiers en sortie se voit attribuer les droits de lecture (READ) et écriture (WRITE) pour tous les utilisateurs

Il n'y a pas de système de récursivité dans les sous-répertoires

CREATE DIRECTORY FICHIERS_IN AS 'd:\fichiers\in' n'autorise pas l'accès aux éventuels sous-répertoires

8.1 - Procédures et fonctions du paquetage

8.1.1 - Liste des procédures et fonctions version 8i

- Fonction **IS_OPEN** Teste si le pointeur de fichier se réfère à un fichier ouvert
- Procédure **FCLOSE** Fermeture d'un fichier
- Procédure **FCLOSE_ALL** Fermeture de tous les fichiers ouverts
- Fonction **FOPEN** Ouverture d'un fichier pour lecture ou écriture
- Procédure **FFLUSH** Ecriture physique des tampons sur le disque
- Procédure **GET_LINE** Lecture d'une ligne depuis un fichier ouvert
- Procédure **PUT** Ecriture d'une ligne (sans caractère fin de ligne) dans un fichier ouvert
- Procédure **PUT_LINE** Ecriture d'une ligne (avec caractère fin de ligne) dans un fichier ouvert
- Procédure **PUTF** Ecriture d'une ligne formatée
- Procédure **PUTF** Ecriture d'une ligne formatée

8.1.2 - Liste des procédures et fonctions version 9i

- Fonction **IS_OPEN** Test si le pointeur de fichier se réfère à un fichier ouvert
- Procédure **FCLOSE** Fermeture d'un fichier
- Procédure **FCLOSE_ALL** Fermeture de tous les fichiers ouverts
- Fonction **FCOPY** Copie d'un fichier sur le disque
- Fonction **FOPEN** Ouverture d'un fichier pour lecture ou écriture
- Fonction **FOPEN_NCHAR** Ouverture d'un fichier en unicode pour lecture ou écriture
- Procédure **FFLUSH** Ecriture physique des tampons sur le disque
- Procédure **FGETATTR** Lecture des attributs d'un fichier
- Fonction **FGETPOS** Retourne la position du pointeur de lecture/écriture
- Fonction **FREMOVE** Suppression d'un fichier sur le disque
- Fonction **FRENAME** Renommage d'un fichier
- Fonction **FSEEK** Déplacement du pointeur de lecture/écriture dans le fichier
- Procédure **GET_LINE** Lecture d'une ligne depuis un fichier ouvert
- Procédure **GET_LINE_NCHAR** Lecture d'une ligne en unicode depuis un fichier ouvert
- Procédure **GET_RAW** Lecture d'une ligne de type RAW depuis un fichier ouvert avec ajustement du pointeur de position
- Procédure **NEW_LINE** Ecriture d'un caractère fin de ligne dans un fichier ouvert
- Procédure **PUT** Ecriture d'une ligne (sans caractère fin de ligne) dans un fichier ouvert
- Procédure **PUT_NCHAR** Ecriture d'une ligne en unicode (sans caractère fin de ligne) dans un fichier ouvert
- Procédure **PUT_RAW** Ecriture d'une ligne de type RAW dans un fichier ouvert
- Procédure **PUT_LINE** Ecriture d'une ligne (avec caractère fin de ligne) dans un fichier ouvert
- Procédure **PUT_LINE_NCHAR** Ecriture d'une ligne en unicode (avec caractère fin de ligne) dans un fichier

ouvert
- Procédure **PUTF** Ecriture d'une ligne formatée
- Procédure **PUTF_NCHAR** Ecriture d'une ligne en unicode formatée

8.2 - Syntaxe des procédures et fonctions

8.2.1 - IS_OPEN

Vérification de l'ouverture d'un fichier

UTL_FILE.IS_OPEN (

pointeur IN FILE_TYPE)

RETURN BOOLEAN

pointeur représente une variable de type UTL_FILE.FILE_TYPE préalablement obtenue par un appel aux fonctions FOPEN() ou FOPEN_NCHAR()

La fonction retourne TRUE si le fichier est ouvert, sinon FALSE

8.2.2 - FCLOSE

Fermeture d'un fichier

UTL_FILE.FCLOSE (

pointeur IN OUT FILE_TYPE)

pointeur représente une variable de type UTL_FILE.FILE_TYPE préalablement obtenue par un appel aux fonctions FOPEN() ou FOPEN_NCHAR()

Exceptions générées

WRITE_ERROR

INVALID_FILEHANDLE

8.2.3 - FCLOSE_ALL

Fermeture de tous les fichiers ouverts de la session en cours

UTL_FILE.FCLOSE_ALL

Exceptions générées

WRITE_ERROR

8.2.4 - FCOPY

Copie d'un fichier

UTL_FILE.FCOPY (

dir_source IN VARCHAR2,

fichier_source IN VARCHAR2,

dir_cible IN VARCHAR2,

fichier_cible IN VARCHAR2,

ligne_debut IN PLS_INTEGER DEFAULT 1,

ligne_fin IN PLS_INTEGER DEFAULT NULL)

dir_source représente le répertoire Oracle du fichier source

fichier_source représente le nom du fichier source avec son extension

dir_cible représente le répertoire de destination

fichier_cible représente le nom du fichier de destination

ligne_debut représente le numéro de ligne de début de copie (par défaut 1, début du fichier)

ligne_fin représente le numéro de ligne de fin de copie (par défaut NULL, fin du fichier)

Cette fonction permet de créer un fichier à partir d'un autre en recopiant tout ou partie du fichier source.

8.2.5 - FOPEN

Ouverture d'un fichier

Oracle 8i, 9i

UTL_FILE.FOPEN (

repertoire IN VARCHAR2,

fichier IN VARCHAR2,

mode IN VARCHAR2,

taille_ligne_maxi IN BINARY_INTEGER)

RETURN UTL_FILE.FILE_TYPE

repertoire représente le chemin d'accès (8i) ou le répertoire (9i)

fichier représente le nom du fichier avec son extension

mode représente le mode d'ouverture du fichier qui peut prendre l'une des trois valeurs suivantes

- **R** le fichier est ouvert en lecture (Read)
- **W** le fichier est ouvert en écriture (Write)
- **A** le fichier est ouvert en ajout (Append)

taille_ligne_maxi représente la taille maximum en octets d'une ligne lue ou écrite. La plage des valeurs acceptables est comprise entre 1 et 32767 (par défaut environ 1000)

La fonction retourne un enregistrement de type UTL_FILE.FILE_TYPE

Exceptions générées

Oracle 8i, 9i

INVALID_PATH: Nom de répertoire ou de fichier invalide.

INVALID_MODE: Mode d'ouverture invalide.

INVALID_OPERATION: Le fihcier ne peut être ouvert.

INVALID_MAXLINESIZE: La valeur de taille_ligne_maxi est trop grande ou trop petite.

8.2.6 - FOPEN_NCHAR

(9i)

Ouverture d'un fichier

Identique à la fonction FOPEN, mais traite des fichiers en mode unicode

UTL_FILE.FOPEN_NCHAR (

repertoire IN VARCHAR2,

fichier IN VARCHAR2,

mode IN VARCHAR2,

taille_ligne_maxi IN BINARY_INTEGER)

RETURN UTL_FILE.FILE_TYPE

8.2.7 - FFLUSH

Ecriture physique des tampons sur le disque

UTL_FILE.FFLUSH (

pointeur IN FILE_TYPE)

pointeur représente une variable de type UTL_FILE.FILE_TYPE préalablement obtenue par un appel aux fonctions FOPEN() ou FOPEN_NCHAR()

Cette fonction permet de forcer l'écriture du tampon sur le disque

Exceptions générées

INVALID_FILEHANDLE

INVALID_OPERATION

WRITE_ERROR

8.2.8 - FGETATTR

(9i)

Lecture des attributs d'un fichier

UTL_FILE.FGETATTR(

repertoire IN VARCHAR2,

fichier IN VARCHAR2,

exists OUT BOOLEAN,

taille_fichier OUT NUMBER,

taille_bloc OUT NUMBER)

repertoire représente le répertoire Oracle

fichier représente le nom du fichier avec son extension

exists vaut TRUE si le fichier existe, sinon FALSE

taille_fichier représente la taille du fichier en octets

taille_bloc représente la taille d'un bloc système en octets

Cette fonction teste l'existence d'un fichier et récupère, dans l'affirmative, la taille du fichier et la taille du bloc système

8.2.9 - FGETPOS

Position du pointeur de lecture/écriture du fichier

UTL_FILE.FGETPOS (

pointeur IN file_type)

RETURN PLS_INTEGER

pointeur représente une variable de type UTL_FILE.FILE_TYPE préalablement obtenue par un appel aux fonctions FOPEN() ou FOPEN_NCHAR()

Cette fonction retourne la position en octets actuelle de pointeur du fichier ouvert

8.2.10 - FREMOVE

(9i)

Suppression d'un fichier sur disque

UTL_FILE.FREMOVE (

repertoire IN VARCHAR2,

fichier IN VARCHAR2)

repertoire représente le répertoire Oracle

fichier représente le nom du fichier avec son extension

8.2.11 - FRENAME

(9i)

Renommage d'un fichier sur disque

UTL_FILE.FRENAME (

dir_source IN VARCHAR2,

fichier_source IN VARCHAR2,

dir_cible IN VARCHAR2,

fichier_cible IN VARCHAR2,

remplacer IN BOOLEAN DEFAULT FALSE)

dir_source représente le répertoire Oracle du fichier source

fichier_source représente le nom du fichier source avec son extension

dir_cible représente le répertoire de destination

fichier_cible représente le nom du fichier de destination

remplacer positionné à TRUE permet d'écraser un fichier existant

Cette fonction permet de renommer un fichier avec possibilité de déplacement, comme de la commande Unix **mv**

8.2.12 - FSEEK

(9i)

Positionnement du pointeur de lecture/écriture dans le fichier

UTL_FILE.FSEEK (

pointeur IN utl_file.file_type,

déplacement_absolu IN PL_INTEGER DEFAULT NULL,

déplacement_relatif IN PLS_INTEGER DEFAULT NULL)

pointeur représente une variable de type UTL_FILE.FILE_TYPE préalablement obtenue par un appel aux fonctions FOPEN() ou FOPEN_NCHAR()

déplacement_absolu représente l'octet du fichier sur lequel on veut pointer

déplacement_relatif représente un déplacement vers l'avant ou vers l'arrière par rapport à la position courante

si **déplacement_relatif** est positif, le déplacement s'effectue vers la fin du fichier

si **déplacement_relatif** est négatif, le déplacement s'effectue vers le début du fichier

8.2.13 - GET_LINE

Lecture d'une ligne depuis un fichier

Oracle 8i

UTL_FILE.GET_LINE (

pointeur IN FILE_TYPE,

tampon OUT VARCHAR2);

Oracle 9i

UTL_FILE.GET_LINE (

pointeur IN FILE_TYPE,

tampon OUT VARCHAR2,

longueur IN NUMBER,

len IN PLS_INTEGER DEFAULT NULL)

pointeur représente une variable de type UTL_FILE.FILE_TYPE préalablement obtenue par un appel aux fonctions FOPEN() ou FOPEN_NCHAR()

tampon représente la variable destinée à recevoir les données lues

longueur représente le nombre maximum d'octets à lire

len représente le nombre d'octets à lire. Par défaut il est à NULL ce qui signifie qu'il vaut la taille maximum d'un RAW

La variable tampon doit être suffisamment dimensionnée pour accueillir la ligne lue depuis le fichier sous peine de générer l'exception VALUE_ERROR

longueur ne peut pas excéder la valeur définie par le paramètre taille_ligne_maxi de la fonction FOPEN()

Lorsque la lecture ne ramène aucune donnée (fin de fichier), l'exception NO_DATA_FOUND est générée

Exceptions générées

INVALID_FILEHANDLE

INVALID_OPERATION

READ_ERROR

NO_DATA_FOUND

VALUE_ERROR

8.2.14 - GET_LINE_NCHAR

(9i)

Lecture d'une ligne en unicode depuis un fichier

UTL_FILE.GET_LINE_NCHAR (

pointeur IN FILE_TYPE,

tampon OUT NVARCHAR2,

len IN PLS_INTEGER DEFAULT NULL)

pointeur représente une variable de type UTL_FILE.FILE_TYPE préalablement obtenue par un appel aux fonctions FOPEN() ou FOPEN_NCHAR()

tampon représente la variable de type NVARCHAR2 retournée par la procédure

len représente le nombre d'octets à lire. Par défaut il est à NULL ce qui signifie qu'il vaut la taille maximum d'un RAW

Exceptions générées

INVALID_FILEHANDLE

INVALID_OPERATION

READ_ERROR

NO_DATA_FOUND

VALUE_ERROR

8.2.15 - GET_RAW

(9i)

Lecture d'une ligne dans une variable RAW depuis un fichier

UTL_FILE.GET_RAW (

pointeur IN utl_file.file_type,

tampon OUT NOCOPY RAW,

len IN PLS_INTEGER DEFAULT NULL)

pointeur représente une variable de type UTL_FILE.FILE_TYPE préalablement obtenue par un appel aux fonctions FOPEN() ou FOPEN_NCHAR()

tampon représente la variable de type RAW retournée par la procédure

len représente le nombre d'octets à lire. Par défaut il est à NULL ce qui signifie qu'il vaut la taille maximum d'un RAW

Exceptions générées

INVALID_FILEHANDLE

INVALID_OPERATION

READ_ERROR

NO_DATA_FOUND

VALUE_ERROR

8.2.16 - NEW_LINE

Ecriture d'un caractère fin de ligne dans un fichier

UTL_FILE.NEW_LINE (

pointeur IN FILE_TYPE,

nombre_lignes IN NATURAL := 1)

pointeur représente une variable de type UTL_FILE.FILE_TYPE préalablement obtenue par un appel aux fonctions FOPEN() ou FOPEN_NCHAR()

nombre_lignes représente le nombre de terminateurs fin de ligne que l'on souhaite écrire dans le fichier (par défaut 1)

Exceptions générées

INVALID_FILEHANDLE

INVALID_OPERATION

WRITE_ERROR

8.2.17 - PUT

Ecriture d'une ligne (sans caractère fin de ligne) dans un fichier

UTL_FILE.PUT (

pointeur IN FILE_TYPE,

tampon IN VARCHAR2)

pointeur représente une variable de type UTL_FILE.FILE_TYPE préalablement obtenue par un appel aux fonctions FOPEN() ou FOPEN_NCHAR()

tampon représente la variable destinée à contenir les données à écrire

Exceptions générées

INVALID_FILEHANDLE

INVALID_OPERATION

WRITE_ERROR

8.2.18 - PUT_NCHAR

(9i)

Ecriture d'une ligne en unicode (sans caractère fin de ligne) dans un fichier

UTL_FILE.PUT_INCHAR (

pointeur IN FILE_TYPE,

tampon IN NVARCHAR2)

pointeur représente une variable de type UTL_FILE.FILE_TYPE préalablement obtenue par un appel aux fonctions FOPEN() ou FOPEN_NCHAR()

tampon représente la variable de type NVARCHAR2 destinée à contenir les données à écrire

Exceptions générées

INVALID_FILEHANDLE

INVALID_OPERATION

WRITE_ERROR

8.2.19 - PUT_RAW

(9i)

Ecriture d'une ligne de type RAW dans un fichier

UTL_FILE. PUT_RAW (

pointeur IN utl_file.file_type,

tampon IN RAW,

autoflush IN BOOLEAN DEFAULT FALSE)

pointeur représente une variable de type UTL_FILE.FILE_TYPE préalablement obtenue par un appel aux fonctions FOPEN() ou FOPEN_NCHAR()

tampon représente la variable de type RAW destinée à contenir les données à écrire

autoflush, positionné à TRUE permet de forcer l'écriture du tampon sur disque.

Exceptions générées

INVALID_FILEHANDLE

INVALID_OPERATION

WRITE_ERROR

8.2.20 - PUT_LINE

Ecriture d'une ligne (avec caractère fin de ligne) dans un fichier

Oracle 8i

UTL_FILE.PUT_LINE (

pointeur IN FILE_TYPE,

tampon IN VARCHAR2)

Oracle 9i

UTL_FILE.PUT_LINE (

pointeur IN FILE_TYPE,

tampon IN VARCHAR2,

autoflush IN BOOLEAN DEFAULT FALSE)

pointeur représente une variable de type UTL_FILE.FILE_TYPE préalablement obtenue par un appel aux fonctions FOPEN() ou FOPEN_NCHAR()

tampon représente la variable destinée à contenir les données à écrire

autoflush, positionné à TRUE permet de forcer l'écriture du tampon sur disque.

Un caractère fin de ligne est ajouté en fin de tampon

8.2.21 - PUT_LINE_NCHAR

(9i)

Ecriture d'une ligne en unicode (avec caractère fin de ligne) dans un fichier

UTL_FILE.PUT_LINE_NCHAR (

pointeur IN FILE_TYPE,

tampon IN NVARCHAR2,

autoflush IN BOOLEAN DEFAULT FALSE)

pointeur représente une variable de type UTL_FILE.FILE_TYPE préalablement obtenue par un appel aux fonctions FOPEN() ou FOPEN_NCHAR()

tampon représente la variable de type NVARCHAR2 destinée à contenir les données à écrire

autoflush, positionné à TRUE permet de forcer l'écriture du tampon sur disque.

Un caractère fin de ligne est ajouté en fin de tampon

8.2.22 - PUTF

Ecriture d'une ligne formatée dans un fichier

UTL_FILE.PUTF (

pointeur IN FILE_TYPE,

format IN VARCHAR2,

[arg1 IN VARCHAR2 DEFAULT NULL,

. . .

arg5 IN VARCHAR2 DEFAULT NULL])

pointeur représente une variable de type UTL_FILE.FILE_TYPE préalablement obtenue par un appel aux fonctions FOPEN() ou FOPEN_NCHAR()

format représente une chaîne pouvant inclure les caractères %s et \n

arg1.. arg5 représentent jusqu'à 5 paramètres dont chaque valeur remplace un caractère %s du format donné.

Cette fonction ressemble à une version limitée de fprintf() du langage C

Exceptions générées

INVALID_FILEHANDLE

INVALID_OPERATION

WRITE_ERROR

8.2.23 - PUTF_NCHAR

Ecriture d'une ligne unicode formatée dans un fichier

UTL_FILE.PUTF_NCHAR (

pointeur IN FILE_TYPE,

format IN NVARCHAR2,

[arg1 IN NVARCHAR2 DEFAULT NULL,

. . .

arg5 IN NVARCHAR2 DEFAULT NULL])

pointeur représente une variable de type UTL_FILE.FILE_TYPE préalablement obtenue par un appel aux fonctions FOPEN() ou FOPEN_NCHAR()

format représente une chaîne pouvant inclure les caractères %s et \n

arg1.. arg5 représentent jusqu'à 5 paramètres dont chaque valeur remplace un caractère %s du format donné.

Exceptions générées

INVALID_FILEHANDLE

INVALID_OPERATION

WRITE_ERROR

8.3 - Exceptions générées par le paquetage

8.3.1 - Exceptions de la version 8i

- **INVALID_PATH** La directorie ou le nom de fichier est invalide
- **INVALID_MODE** Mode d'ouverture invalide pour la fonction FOPEN
- **INVALID_FILEHANDLE** Pointeur de fichier invalide
- **INVALID_OPERATION** Le fichier ne peut être ouvert ou manipulé
- **READ_ERROR** Erreur système pendant une opération de lecture
- **WRITE_ERROR** Erreur système pendant une opération d'écriture
- **INTERNAL_ERROR** Erreur PL/SQL non spécifiée

8.3.2 - Exceptions de la version 9i

- **INVALID_PATH** La directorie ou le nom de fichier est invalide
- **INVALID_MODE** Mode d'ouverture invalide pour la fonction FOPEN
- **INVALID_FILEHANDLE** Pointeur de fichier invalide
- **INVALID_OPERATION** Le fichier ne peut être ouvert ou manipulé

- **READ_ERROR** Erreur système pendant une opération de lecture
- **WRITE_ERROR** Erreur système pendant une opération d'écriture
- **INTERNAL_ERROR** Erreur PL/SQL non spécifiée
- **CHARSETMISMATCH** Utilisation de fonctions non NCHAR après une ouverture avec FOPEN_NCHAR
- **FILE_OPEN** L'opération a échoué car le fichier est ouvert
- **INVALID_MAXLINESIZE** La taille MAXLINESIZE pour la fonction FOPEN doit être entre 1 to 32767
- **INVALID_FILENAME** Nom de fichier invalide
- **ACCESS_DENIED** Vous n'avez pas les droits d'accès au fichier
- **INVALID_OFFSET** Le déplacement déclaré dans la fonction FSEEK() doit être supérieur à 0 et inférieur à la taille du fichier
- **DELETE_FAILED** L'opération de suppression à échoué
- **RENAME_FAILED** L'opération de renommage a échoué

8.4 - Exemples concrets

- Ouverture d'un fichier en lecture et écriture du contenu dans un autre

```
Declare
  -- Noms des fichiers --
  LC$Fic_in   Varchar2(128) := 'EMP.TXT' ;       -- a adapter sur votre configuration
  LC$Fic_out  Varchar2(128) := 'EMP2.TXT' ;      -- a adapter sur votre configuration
  -- Noms des répertoires --
  LC$Dir_in   Varchar(30)   := 'FICHIERS_IN';    -- a adapter sur votre configuration
  LC$Dir_out  Varchar(30)   := 'FICHIERS_OUT' ;  -- a adapter sur votre configuration
  -- Pointeurs de fichier --
  LF$FicIN UTL_FILE.FILE_TYPE ;
  LF$FicOUT UTL_FILE.FILE_TYPE ;
  -- Tampon de travail --
  LC$Ligne Varchar2(32767) ;
  -- Message --
  LC$Msg    Varchar2(256) ;
  -- Exception --
  LE$Fin    Exception ;
Begin

  -- Ouverture du fichier en entrée
  Begin
    LF$FicIN := UTL_FILE.FOPEN( LC$Dir_in, LC$Fic_in, 'R', 32764 ) ;
  Exception
    When OTHERS Then
        LC$Msg := SQLERRM || ' [' || LC$Dir_in || '] -> ' || LC$Fic_in;
        Raise LE$Fin ;
  End ;

  -- Ouverture du fichier en sortie
  Begin
    LF$FicOUT := UTL_FILE.FOPEN( LC$Dir_out, LC$Fic_out, 'W', 32764 ) ;
  Exception
    When OTHERS Then
        LC$Msg := SQLERRM || ' [' || LC$Dir_out || '] -> ' || LC$Fic_out;
        Raise LE$Fin ;
  End ;

  -- Traitement --
  Begin
     Loop
        -- lecture du fichier en entrée --
        UTL_FILE.GET_LINE( LF$FicIN,  LC$Ligne ) ;
        -- écriture du fichier en sortie --
        UTL_FILE.PUT_LINE( LF$FicOUT, LC$Ligne ) ;
     End loop ;
  Exception
     When NO_DATA_FOUND Then -- Fin du fichier en entrée
        -- Fermeture des fichiers --
```

```
         UTL_FILE.FCLOSE( LF$FicIN ) ;
         UTL_FILE.FCLOSE( LF$FicOUT ) ;
   End  ;

Exception
   When LE$Fin Then
     UTL_FILE.FCLOSE_ALL ;
     RAISE_APPLICATION_ERROR( -20100, LC$Msg ) ;
End ;
```

- Extraction d'une table dans un fichier

Voici le code d'une procédure qui permet d'extraire le contenu d'une table dans un fichier

Cette procédure a besoin des 3 premiers paramètres pour fonctionner

PC$Table reçoit le nom d'une table Oracle

PC$Fichier reçoit le nom du fichier de sortie

PC$Repertoire reçoit le nom du chemin (8i) ou de la directorie (9i)

PC$Separateur reçoit le caractère de séparation voulu (défaut ,)

PC$DateFMT reçoit le format des dates (défaut : DD/MM/YYYY)

PC$Where reçoit une éventuelle clause WHERE

PC$Order reçoit une éventuelle clause ORDER BY

le paramètre **PC$Entetes** permet les actions suivantes

- positionné à O, il indique une sortie des lignes de la table avec une ligne d'entête
- positionné à I, il indique la génération des ordres INSERT pour chaque ligne
- différent de O et I, il indique une sortie des lignes de la table sans ligne d'entête

```
CREATE OR REPLACE PROCEDURE Extraction_Table
        (
                PC$Table      in Varchar2,                      -- Nom de la table a extraire
                PC$Fichier    in Varchar2,                      -- Nom du fichier de sortie
                PC$Repertoire in Varchar2,                      -- Nom du directory de sortie
                PC$Separateur in Varchar2 Default ',',          -- Caractere de separation
                PC$Entetes    in Varchar2 Default 'O',          -- Affichage de l'entete des
colonnes
                PC$DateFMT    in Varchar2 Default 'DD/MM/YYYY', -- Format des dates
                PC$Where      in Varchar2 Default Null,         -- Clause Where de filtrage
                PC$Order      in Varchar2 Default Null          -- Colonne de tri
        ) IS


LF$Fichier  UTL_FILE.FILE_TYPE ;
```

```
LC$Ligne    Varchar2(32767) ;
LI$I        Integer ;
LC$DateFMT  Varchar2(40) := '''' || PC$DateFMT || '''' ;


TYPE REFCUR1 IS REF CURSOR ;
cur         REFCUR1;


-- Colonnes de la table --
  CURSOR C_COLTAB ( PC$Tab IN VARCHAR2 ) IS
  SELECT
         COLUMN_NAME,
       DATA_TYPE
  FROM
         USER_TAB_COLUMNS
  WHERE
         TABLE_NAME = PC$Tab
  AND
         DATA_TYPE IN ('CHAR','VARCHAR2','NUMBER','DATE','FLOAT')
  ;

LC$Separateur Varchar2(2) := PC$Separateur ;
LC$Requete Varchar2(10000) ;
LC$Desc         Varchar2(10000) ;
LC$SQLW         VARCHAR2(10000):= 'SELECT ';
LC$Col         VARCHAR2(256);


----------------------------------------
-- Ouverture d'un fichier d'extraction --
----------------------------------------
FUNCTION Ouvrir_fichier
        (
                PC$Dir in Varchar2,
                PC$Nom_Fichier in Varchar2
        ) RETURN UTL_FILE.FILE_TYPE
IS
  Fichier UTL_FILE.FILE_TYPE ;
  LC$Msg Varchar2(256);

Begin

  Fichier := UTL_FILE.FOPEN( PC$Dir, PC$Nom_Fichier, 'W', 32764 ) ;

  If not UTL_FILE.IS_OPEN( Fichier ) Then
        LC$Msg := 'Erreur ouverture du fichier (' || PC$Dir || ') ' || PC$Nom_Fichier ;
          RAISE_APPLICATION_ERROR( -20100, LC$Msg ) ;
  End if ;

  Return( Fichier ) ;

Exception

When UTL_FILE.INVALID_PATH Then
        LC$Msg := PC$Dir || PC$Nom_Fichier || ' : ' || 'File location is invalid.';
        RAISE_APPLICATION_ERROR( -20070, LC$Msg ) ;
When UTL_FILE.INVALID_MODE Then
        LC$Msg := PC$Dir || PC$Nom_Fichier || ' : ' || 'The open_mode parameter in FOPEN is
invalid.';
        RAISE_APPLICATION_ERROR( -20070, LC$Msg ) ;
When UTL_FILE.INVALID_FILEHANDLE Then
        LC$Msg := PC$Dir || PC$Nom_Fichier || ' : ' || 'File handle is invalid.';
        RAISE_APPLICATION_ERROR( -20070, LC$Msg ) ;
When UTL_FILE.INVALID_OPERATION Then
        LC$Msg := PC$Dir || PC$Nom_Fichier || ' : ' || 'File could not be opened or operated on as
requested.';
        RAISE_APPLICATION_ERROR( -20070, LC$Msg ) ;
When UTL_FILE.READ_ERROR Then
        LC$Msg := PC$Dir || PC$Nom_Fichier || ' : ' || 'Operating system error occurred during the
read operation.';
        RAISE_APPLICATION_ERROR( -20070, LC$Msg ) ;
When UTL_FILE.WRITE_ERROR Then
        LC$Msg := PC$Dir || PC$Nom_Fichier || ' : ' || 'Operating system error occurred during the
write operation.';
        RAISE_APPLICATION_ERROR( -20070, LC$Msg ) ;
When UTL_FILE.INTERNAL_ERROR then
        LC$Msg := PC$Dir || PC$Nom_Fichier || ' : ' || 'Unspecified PL/SQL error';
        RAISE_APPLICATION_ERROR( -20070, LC$Msg ) ;
```

```
   ------------------------------------------------------------
   -- Les exceptions suivantes sont spécifiques à la version 9i --
   -- A mettre en commentaire pour une version antérieure       --
   ------------------------------------------------------------
   When UTL_FILE.CHARSETMISMATCH Then
           LC$Msg := PC$Dir || PC$Nom_Fichier || ' : ' || 'A file is opened using FOPEN_NCHAR,'
             || ' but later I/O operations use nonchar functions such as PUTF or GET_LINE.';
           RAISE_APPLICATION_ERROR( -20070, LC$Msg ) ;
   When UTL_FILE.FILE_OPEN Then
           LC$Msg := PC$Dir || PC$Nom_Fichier || ' : ' || 'The requested operation failed because the
   file is open.';
           RAISE_APPLICATION_ERROR( -20070, LC$Msg ) ;
   When UTL_FILE.INVALID_MAXLINESIZE Then
           LC$Msg := PC$Dir || PC$Nom_Fichier || ' : ' || 'The MAX_LINESIZE value for FOPEN() is
   invalid;'
             || ' it should be within the range 1 to 32767.';
           RAISE_APPLICATION_ERROR( -20070, LC$Msg ) ;
   When UTL_FILE.INVALID_FILENAME Then
           LC$Msg := PC$Dir || PC$Nom_Fichier || ' : ' || 'The filename parameter is invalid.';
           RAISE_APPLICATION_ERROR( -20070, LC$Msg ) ;
   When UTL_FILE.ACCESS_DENIED Then
           LC$Msg := PC$Dir || PC$Nom_Fichier || ' : ' || 'Permission to access to the file location is
   denied.';
           RAISE_APPLICATION_ERROR( -20070, LC$Msg ) ;
   When UTL_FILE.INVALID_OFFSET Then
           LC$Msg := PC$Dir || PC$Nom_Fichier || ' : ' || 'The ABSOLUTE_OFFSET parameter for FSEEK() is
   invalid;'
             ||' it should be greater than 0 and less than the total number of bytes in the file.';
           RAISE_APPLICATION_ERROR( -20070, LC$Msg ) ;
   When UTL_FILE.DELETE_FAILED Then
           LC$Msg := PC$Dir || PC$Nom_Fichier || ' : ' || 'The requested file delete operation
   failed.';
           RAISE_APPLICATION_ERROR( -20070, LC$Msg ) ;
   When UTL_FILE.RENAME_FAILED Then
           LC$Msg := PC$Dir || PC$Nom_Fichier || ' : ' || 'The requested file rename operation
   failed.';
           RAISE_APPLICATION_ERROR( -20070, LC$Msg ) ;
   ------------------------------------------------------------------
   -- Les exceptions précédentes sont spécifiques à la version 9i --
   --     mettre en commentaire pour une version antérieure       --
   ------------------------------------------------------------------
   When others Then
           LC$Msg := 'Erreur : ' || To_char( SQLCODE ) || ' sur ouverture du fichier ('
              || PC$Dir || ') ' || PC$Nom_Fichier ;
           RAISE_APPLICATION_ERROR( -20070, LC$Msg ) ;

   End Ouvrir_fichier ;

   Begin

     -- Ouverture du fichier --
     LF$Fichier := Ouvrir_fichier( PC$Repertoire, PC$Fichier ) ;

     -- Affichage des entetes de colonne ? --
     If Upper(PC$Entetes) = 'O' Then
           LI$I := 1 ;
           For COLS IN C_COLTAB( PC$Table ) Loop
              If LI$I = 1 Then
                 LC$Ligne := LC$Ligne || COLS.COLUMN_NAME ;
              Else
                 LC$Ligne := LC$Ligne || LC$Separateur || COLS.COLUMN_NAME ;
              End if ;
              LI$I := LI$I + 1 ;
           End loop ;
           -- Ecriture ligne entetes --
           UTL_FILE.PUT_LINE( LF$Fichier, LC$Ligne ) ;
     ElsIf Upper(PC$Entetes) = 'I' Then
       LC$Separateur := ',' ;
       LC$Desc := 'INSERT INTO ' || PC$Table || ' (' ;
       LI$I := 1 ;
       For COLS IN C_COLTAB( PC$Table ) Loop
         If LI$I = 1 Then
                LC$Desc := LC$Desc || COLS.COLUMN_NAME ;
         Else
                LC$Desc := LC$Desc || LC$Separateur || COLS.COLUMN_NAME ;
         End if ;
         LI$I := LI$I + 1 ;
       End loop ;
       LC$Desc := LC$Desc || ' ) VALUES (' ;
```

```
    End if ;

    -- Construction de la requete --
    LI$I := 1 ;

    FOR COLS IN C_COLTAB( PC$Table ) LOOP
      IF LI$I > 1 THEN
         LC$SQLW := LC$SQLW || '||' ;
      END IF ;

      If COLS.DATA_TYPE IN ('NUMBER','FLOAT') Then
              LC$Col := 'Decode(' || COLS.COLUMN_NAME || ',NULL, ''NULL'',To_char("'
                 || COLS.COLUMN_NAME || '"))' ;
      ElsIf COLS.DATA_TYPE = 'DATE' Then
         If Upper(PC$Entetes) = 'I' Then
             LC$Col := 'Decode(' || COLS.COLUMN_NAME || ',NULL,''NULL'',''to_date(''''''||'
                || 'To_char("' || COLS.COLUMN_NAME || '",'|| LC$DateFMT ||')' || '||''''''',''''||
LC$DateFMT||''')'')' ;
         Else
              LC$Col := 'To_char("'|| COLS.COLUMN_NAME || '",'|| LC$DateFMT ||')' ;
         End if ;
      Else
         If Upper(PC$Entetes) = 'I' Then
             LC$Col := 'Decode(' || COLS.COLUMN_NAME || ',NULL, ''NULL'',' || ''''''''''
              || '|| REPLACE("'|| COLS.COLUMN_NAME || '",CHR(39),CHR(39)||CHR(39))' || '||' ||
''''''''')' ;
         Else
             LC$Col := '"'|| COLS.COLUMN_NAME || '"' ;
         End if ;
      End if ;

      IF LI$I = 1 THEN
         LC$SQLW := LC$SQLW || LC$Col ;
      ELSE
         LC$SQLW := LC$SQLW || '''' || LC$Separateur || '''' || '||' || LC$Col  ;
      END IF ;
      LI$I := LI$I + 1 ;
    END LOOP ;

    LC$Requete := LC$SQLW || ' FROM ' || PC$Table ;

    If PC$Where is not null Then
      -- ajout de la clause WHERE --
      LC$Requete := LC$Requete || ' WHERE ' || PC$Where ;
    End if ;
    If PC$Order is not null Then
      -- ajout de la clause ORDER BY --
      LC$Requete := LC$Requete || ' ORDER BY ' || PC$Order ;
    End if ;

    F_TRACE( LC$Requete, 'T' ) ;
    -- Extraction des lignes --
    Open cur For LC$Requete ;
    Loop
          Fetch cur Into LC$Ligne ;
          Exit when cur%NOTFOUND ;
          -- Ecriture du fichier de sortie --
          If Upper(PC$Entetes) = 'I' Then
            UTL_FILE.PUT_LINE( LF$Fichier, LC$Desc || LC$Ligne || ' );' ) ;
          Else
            UTL_FILE.PUT_LINE( LF$Fichier, LC$Ligne ) ;
          End if ;
    End loop ;

    Close cur ;

    -- Fermeture fichier --
    UTL_FILE.FCLOSE( LF$Fichier ) ;

  End ;
  /
```

Nous allons maintenant extraire les lignes de la table EMP dans le fichier EMP.TXT

```
  SQL> execute extraction_table( 'EMP','EMP.TXT','FICHIERS_OUT' ) ;

  Procédure PL/SQL terminée avec succès.
```

dont voici le contenu

EMPNO,ENAME,JOB,MGR,HIREDATE,SAL,COMM,DEPTNO

7369,SMITH,CLERK,7902,18/12/1980,880,,20

7499,ALLEN,SALESMAN,7698,21/02/1981,1936,300,30

7521,WARD,SALESMAN,7698,23/02/1981,1375,500,30

7566,JONES,MANAGER,7839,03/04/1981,3273,,20

7654,MARTIN,SALESMAN,7698,29/09/1981,1375,1400,30

7698,BLAKE,MANAGER,7839,02/05/1981,3135,,30

7782,CLARK,MANAGER,7839,10/06/1981,2695,,10

7788,SCOTT,ANALYST,7566,20/04/1987,3300,,20

7839,KING,PRESIDENT,,18/11/1981,5500,,10

7844,TURNER,SALESMAN,7698,09/09/1981,1650,0,30

7876,ADAMS,CLERK,7788,24/05/1987,1210,,20

7900,JAMES,CLERK,7698,04/12/1981,1045,,30

7902,FORD,ANALYST,7566,04/12/1981,3300,,20

7934,MILLER,CLERK,7782,24/01/1982,1430,,10

9991,Dupontont,CLERK,,,,,

9992,Duboudin,CLERK,,,,,

9994,Schmoll,CLERK,,,2500,,20

Maintenant, utilisons cette procédure pour générer les ordres INSERT pour les employés ayant le job CLERK dans un fichier EMP.INS

```
SQL>  execute extraction_table( 'EMP','EMP.INS','FICHIERS_OUT', ',', 'I', 'JOB = ''CLERK''' ) ;
Procédure PL/SQL terminée avec succès.
```

dont voici le contenu

```
INSERT INTO EMP (EMPNO,ENAME,JOB,MGR,HIREDATE,SAL,COMM,DEPTNO ) VALUES (7369,'SMITH','CLERK',7902,to_date('18/12/1980','DD/MM/YYYY'),880,NULL,20 );

INSERT INTO EMP (EMPNO,ENAME,JOB,MGR,HIREDATE,SAL,COMM,DEPTNO ) VALUES (7876,'ADAMS','CLERK',7788,to_date('24/05/1987','DD/MM/YYYY'),1210,NULL,20 );

INSERT INTO EMP (EMPNO,ENAME,JOB,MGR,HIREDATE,SAL,COMM,DEPTNO ) VALUES (7900,'JAMES','CLERK',7698,to_date('04/12/1981','DD/MM/YYYY'),1045,NULL,30 );

INSERT INTO EMP (EMPNO,ENAME,JOB,MGR,HIREDATE,SAL,COMM,DEPTNO ) VALUES (7934,'MILLER','CLERK',7782,to_date('24/01/1982','DD/MM/YYYY'),1430,NULL,10 );

INSERT INTO EMP (EMPNO,ENAME,JOB,MGR,HIREDATE,SAL,COMM,DEPTNO ) VALUES (9991,'Dupontont','CLERK',NULL,NULL,NULL,NULL,NULL );

INSERT INTO EMP (EMPNO,ENAME,JOB,MGR,HIREDATE,SAL,COMM,DEPTNO ) VALUES (9992,'Duboudin','CLERK',NULL,NULL,NULL,NULL,NULL );

INSERT INTO EMP (EMPNO,ENAME,JOB,MGR,HIREDATE,SAL,COMM,DEPTNO ) VALUES (9994,'Schmoll','CLERK',NULL,NULL,2500,NULL,20 );
```

9 - Le paquetage DBMS_LOB

Ce paquetage permet de manipuler les grands objets Oracle (LOB) permanents ou temporaires

Il permet la lecture et l'écriture des grands objets de type

- BLOB
- CLOB
- NCLOB

Il permet la lecture des grand objets de type

- BFILE

Une colonne de type BFILE ne stocke qu'un pointeur vers le fichier enregistré au niveau du système d'exploitation

Types utilisés pour la gestion des LOB

- **BLOB** LOB binaire source ou destination.
- **RAW** Tampon de type RAW source ou destination (utilisé avec un BLOB)
- **CLOB** LOB caractères source ou destination (incluant NCLOB)
- **VARCHAR2** Tampon de type caractères source ou destination (utilisé avec CLOB et NCLOB)
- **INTEGER** Specifie la taille d'un tampon ou d'un LOB, le décalage dans un LOB, ou la quantité à lire ou écrire
- **BFILE** Objet binaire stocké en dehors de la base

constantes utilisées pour la gestion des LOB

8i,9i

- **file_readonly** CONSTANT BINARY_INTEGER := 0;
- **lob_readonly** CONSTANT BINARY_INTEGER := 0;
- **lob_readwrite** CONSTANT BINARY_INTEGER := 1;
- **lobmaxsize** CONSTANT INTEGER := 4294967295;
- **call** CONSTANT PLS_INTEGER := 12;
- **session** CONSTANT PLS_INTEGER := 10;
- **warn_inconvertible_char** CONSTANT INTEGER := 1;
- **default_csid** CONSTANT INTEGER := 0;
- **default_lang_ctx** CONSTANT INTEGER := 0;
- **no_warning** CONSTANT INTEGER := 0;

10g

- **file_readonly** CONSTANT BINARY_INTEGER := 0;
- **lob_readonly** CONSTANT BINARY_INTEGER := 0;
- **lob_readwrite** CONSTANT BINARY_INTEGER := 1;
- **lobmaxsize** CONSTANT INTEGER := 18446744073709551615;
- **call** CONSTANT PLS_INTEGER := 12;
- **session** CONSTANT PLS_INTEGER := 10;
- **warn_inconvertible_char** CONSTANT INTEGER := 1;

- **default_csid** CONSTANT INTEGER := 0;
- **default_lang_ctx** CONSTANT INTEGER := 0;
- **no_warning** CONSTANT INTEGER := 0;

En 10*g*, la taille maximum d'un objet LOB est égale à la valeur du paramètre d'initialisation : **db_block_size** multiplié par **4294967295**.

Cette taille maximum peut donc être comprise entre 8 et 128 teraoctets.

En 9*i*, la taille maximal d'un objet LOB est de 4 gigaoctets (2**32)

Soit 4294967295 caractères pour un BLOB ou CLOB mono-octet

Ou 4294967295 / 2 = 2147483647 caractères pour un CLOB bi-octets

Les LOBs temporaires sont stockés dans le TABLESPACE temporaire et ne survivent pas à la fin de session.

9.1 - Procédures et fonctions du paquetage

9.1.1 - Procédures et fonctions des versions 8i et 9i

- Procédure **APPEND**
- Procédure **CLOSE**
- Fonction **COMPARE**
- Procédure **COPY**
- Procédure **CREATETEMPORARY**
- Procédure **ERASE**
- Procédure **FILECLOSE**
- Procédure **FILECLOSEALL**
- Fonction **FILEEXISTS**
- Procédure **FILEGETNAME**
- Fonction **FILEISOPEN**
- Procédure **FILEOPEN**
- Procédure **FREETEMPORARY**
- Fonction **GETCHUNKSIZE**
- Fonction **GETLENGTH**
- Fonction **INSTR**
- Fonction **ISOPEN**
- Fonction **ISTEMPORARY**
- Procédure **LOADFROMFILE**
- Procédure **OPEN**
- Procédure **READ**
- Fonction **SUBSTR**
- Procédure **TRIM**
- Procédure **WRITE**
- Procédure **WRITEAPPEND**

9.1.2 - Procédures de la version 9i

- Procédure **LOADBLOBFROMFILE**
- Procédure **LOADCLOBFROMFILE**

9.1.3 - Procédures de la version 10g

- Procédure **CONVERTTOBLOB**
- Procédure **CONVERTTOCLOB**

9.2 - Syntaxe des procédures et fonctions

9.2.1 - APPEND

Cette fonction permet d'ajouter le contenu d'un LOB à un autre

DBMS_LOB.APPEND (

dest_lob IN OUT NOCOPY BLOB,

src_lob IN BLOB)

DBMS_LOB.APPEND (

dest_lob IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,

src_lob IN CLOB CHARACTER SET dest_lob%CHARSET)

dest_lob représente le BLOB ou CLOB de destination

src_lob représente le BLOB ou CLOB source

Exceptions générées

VALUE_ERROR Le LOB source ou destination est NULL.

```
DECLARE
  dest_lob BLOB;
  src_lob BLOB;
BEGIN
  -- Initialisation des variables LOB --
  SELECT b_lob INTO dest_lob
  FROM lob_table
  WHERE key_value = 12 FOR UPDATE;

  SELECT b_lob INTO src_lob
  FROM lob_table
  WHERE key_value = 21;

  -- Ajout de src_lob en fin de dest_lob --
  DBMS_LOB.APPEND(dest_lob, src_lob);
  COMMIT;

END;
```

9.2.2 - CLOSE

Ferme un LOB préalablement ouvert

DBMS_LOB.CLOSE (

lob_loc IN OUT NOCOPY BLOB)

DBMS_LOB.CLOSE (

lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS)

DBMS_LOB.CLOSE (

file_loc IN OUT NOCOPY BFILE)

9.2.3 - COMPARE

Comparaison de tout ou partie de 2 LOBs

Les 2 LOBs doivent être de même type

Cette fonction retourne 0 si les deux objets sont identiques, sinon une valeur différente de 0

Elle retourne NULL si

- amount < 1
- amount > LOBMAXSIZE
- déplacement_1 ou déplacement_2 < 1

- déplacement_1 ou déplacement_2 > LOBMAXSIZE

DBMS_LOB.COMPARE (

lob_1 IN BLOB,

lob_2 IN BLOB,

amount IN INTEGER := 4294967295,

déplacement_1 IN INTEGER := 1,

déplacement_2 IN INTEGER := 1)

RETURN INTEGER

DBMS_LOB.COMPARE (

lob_1 IN CLOB CHARACTER SET ANY_CS,

lob_2 IN CLOB CHARACTER SET lob_1%CHARSET,

amount IN INTEGER := 4294967295,

déplacement_1 IN INTEGER := 1,

déplacement_2 IN INTEGER := 1)

RETURN INTEGER

DBMS_LOB.COMPARE (

lob_1 IN BFILE,

lob_2 IN BFILE,

amount IN INTEGER,

déplacement_1 IN INTEGER := 1,

déplacement_2 IN INTEGER := 1)

RETURN INTEGER

lob_1 représente le premier LOB

lob_2 représente le deuxième LOB

amount représente le nombre d'octets (BLOB) ou de caractères (CLOB) à comparer

déplacement_1 représente le décalage en octets ou caractères à partir du début (défaut 1) du premier LOB

déplacement_2 représente le décalage en octets ou caractères à partir du début (défaut 1) du deuxième LOB

Exceptions générées

UNOPENED_FILE Le fichier n'est pas ouvert

NOEXIST_DIRECTORY le répertoire n'existe pas

NOPRIV_DIRECTORY Privilèges insuffisants sur le répertoire

INVALID_DIRECTORY le répertoire a été invalidé après l'ouverture du fichier

INVALID_OPERATION Fichier inexistant ou privilèges insuffisants

```
DECLARE
  lob_1 BLOB;
  lob_2 BLOB;
  retval INTEGER;
BEGIN
  SELECT b_col INTO lob_1 FROM lob_table
  WHERE key_value = 45;

  SELECT b_col INTO lob_2 FROM lob_table
  WHERE key_value = 54;

  retval := dbms_lob.compare(lob_1, lob_2, dbms_lob.lobmaxsize, 1, 1);

  IF retval = 0 THEN
     dbms_output.put_line( 'LOBs identiques' ) ;
  ELSE
     dbms_output.put_line( 'LOBs différents' ) ;
  END IF;
END;
```

9.2.4 - CONVERTTOBLOB

(10g) Convertion d'une source CLOB ou NCLOB en format binaire BLOB

DBMS_LOB.CONVERTTOBLOB(

dest_lob IN OUT NOCOPY BLOB,

scr_clob IN CLOB CHARACTER SET ANY_CS,

amount IN INTEGER,

dest_offset IN OUT INTEGER,

src_offset IN OUT INTEGER,

blob_csid IN NUMBER,

lang_context IN OUT INTEGER,

warning OUT INTEGER)

dest_lob représente le BLOB de destination

src_clob représente le CLOB ou NCLOB source

amount représente le nombre de caractères à convertir (défaut lobmaxsize)

dest_offset représente le décalage en octets ou caractères à partir du début (défaut 1) du BLOB destination

src_offset représente le décalage en octets ou caractères à partir du début (défaut 1) du CLOB source

blob_csid ID du jeu de caractères (défaut default_csid) du BLOB destination

lang_context représente le contexte de language (défaut default_lang_ctx) du BLOB destination

warning représente le code avertissement retourné par la fonction

Exceptions générées

VALUE_ERROR L'un des paramètres est NULL ou invalide

INVALID_ARGVAL si

- src_offset ou dest_offset < 1.
- src_offset ou dest_offset > LOBMAXSIZE.
- amount < 1.
- amount > LOBMAXSIZE

9.2.5 - CONVERTTOCLOB

(10g) Convertion d'une source binaire BLOB en format CLOB

DBMS_LOB.CONVERTTOCLOB(

dest_lob IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,

src_blob IN BLOB,

amount IN INTEGER,

dest_offset IN OUT INTEGER,

src_offset IN OUT INTEGER,

blob_csid IN NUMBER,

lang_context IN OUT INTEGER,

warning OUT INTEGER)

dest_lob représente le CLOB de destination

src_clob représente le BLOB source

amount représente le nombre de caractères à convertir (défaut lobmaxsize)

dest_offset représente le décalage en octets ou caractères à partir du début (défaut 1) du BLOB destination

src_offset représente le décalage en octets ou caractères à partir du début (défaut 1) du CLOB source

blob_csid ID du jeu de caractères (défaut default_csid) du BLOB destination

lang_context représente le contexte de language (défaut default_lang_ctx) du BLOB destination

warning représente le code avertissement retourné par la fonction

Exceptions générées

VALUE_ERROR L'un des paramètres est NULL ou invalide

INVALID_ARGVAL si

- src_offset ou dest_offset < 1.
- src_offset ou dest_offset > LOBMAXSIZE.
- amount < 1.
- amount > LOBMAXSIZE

9.2.6 - COPY

Copie de tout ou partie d'un LOB dans un autre

DBMS_LOB.COPY (

dest_lob IN OUT NOCOPY BLOB,

src_lob IN BLOB,

amount IN INTEGER,

dest_déplacement IN INTEGER := 1,

src_déplacement IN INTEGER := 1)

DBMS_LOB.COPY (

dest_lob IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,

src_lob IN CLOB CHARACTER SET dest_lob%CHARSET,

amount IN INTEGER,

dest_déplacement IN INTEGER := 1,

src_déplacement IN INTEGER := 1)

dest_lob représente le LOB destination

src_lob représente le LOB source

amout représente le nombre d'octets (BLOB) ou de caractères (CLOB) que l'on souhaite copier

dest_déplacement représente le déplacement par rapport au début du LOB destination

src_déplacement représente le déplacement par rapport au début du LOB source

Exceptions générées

VALUE_ERROR l'un des paramètres en entrée est NULL ou invalide

INVALID_ARGVAL si:

- src_déplacement ou dest_déplacement < 1

- src_déplacement ou dest_déplacement > LOBMAXSIZE

- amount < 1

- amount > LOBMAXSIZE

```
DECLARE
  lobd BLOB;
  lobs BLOB;
  dest_déplacement INTEGER := 1
  src_déplacement INTEGER := 1
  amt INTEGER := 3000;
BEGIN
  SELECT b_col INTO lobd
  FROM lob_table
  WHERE key_value = 12 FOR UPDATE;

  SELECT b_col INTO lobs
  FROM lob_table
  WHERE key_value = 21;

  DBMS_LOB.COPY(lobd, lobs, amt, dest_déplacement, src_déplacement);
  COMMIT;

END;
```

9.2.7 - CREATETEMPORARY

Création d'un LOB temporaire dans le TABLESPACE temporaire

DBMS_LOB.CREATETEMPORARY (

lob_loc IN OUT NOCOPY BLOB,

cache IN BOOLEAN,

durée IN PLS_INTEGER := 10)

DBMS_LOB.CREATETEMPORARY (

lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,

cache IN BOOLEAN,

duree IN PLS_INTEGER := 10)

cache spécifie si le lob doit être lu en cache

durée spécifie le moment ou le LOB est vidé (10 = SESSION (défaut), 12 = CALL)

9.2.8 - ERASE

Effacement de tout ou partie d'un LOB

La taille du LOB n'est pas affectée par cette fonction. La partie effacée est en fait remplacée par des CHR(0) pour les BLOB et CHR(32) pour les CLOB

DBMS_LOB.ERASE (

lob_loc IN OUT NOCOPY BLOB,

amount IN OUT NOCOPY INTEGER,

déplacement IN INTEGER := 1)

DBMS_LOB.ERASE (

lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,

amount IN OUT NOCOPY INTEGER,

déplacement IN INTEGER := 1)

amout représente le nombre d'octets (BLOB) ou de caractères (CLOB) a effacer

déplacement représente le déplacement par rapport au début du LOB

Exceptions générées

VALUE_ERROR L'un des paramètres est NULL

INVALID_ARGVAL si:

- amount < 1 ou amount > LOBMAXSIZE

- déplacement < 1 ou déplacement > LOBMAXSIZE

```
DECLARE
  lobd BLOB;
  amt INTEGER := 3000;
BEGIN
  SELECT b_col INTO lobd
  FROM lob_table
  WHERE key_value = 12 FOR UPDATE;

  dbms_lob.erase(dest_lob, amt, 2000);

  COMMIT;
END;
```

9.2.9 - FILECLOSE

Fermeture d'un objet BFILE préalablement ouvert

DBMS_LOB.FILECLOSE (

file_loc IN OUT NOCOPY BFILE)

Exceptions générées

VALUE_ERROR Le paramètre en entrée file_loc est NULL

UNOPENED_FILE Le fichier n'était pas ouvert

NOEXIST_DIRECTORY le répertoire n'existe pas

NOPRIV_DIRECTORY Privilèges insuffisants sur le répertoire

INVALID_DIRECTORY le répertoire a été invalidé après l'ouverture du fichier

INVALID_OPERATION Fichier inexistant ou privilèges insuffisants

9.2.10 - FILECLOSEALL

Fermeture de tous les objet BFILE préalablement ouvert dans la session

DBMS_LOB.FILECLOSEALL

Exceptions générées

UNOPENED_FILE Aucun fichier ouvert dans la session

9.2.11 - FILEEXISTS

Vérifie que le pointeur stocké dans le BFILE correspond à un fichier existant du système de fichiers

Cette fonction retourne 1 si le fichier physique existe, sinon elle retourne 0

DBMS_LOB.FILEEXISTS (

file_loc IN BFILE)

RETURN INTEGER

Exceptions générées

NOEXIST_DIRECTORY le répertoire n'existe pas

NOPRIV_DIRECTORY Privilèges insuffisants sur le répertoire

INVALID_DIRECTORY le répertoire a été invalidé après l'ouverture du fichier

```
DECLARE
  fil BFILE;
BEGIN
  SELECT f_lob INTO fil FROM lob_table WHERE key_value = 12;
  IF (dbms_lob.fileexists(fil)) THEN
      dbms_output.put_line( 'fichier existant' ) ;
  ELSE
      dbms_output.put_line( 'fichier inexistant' ) ;
  END IF;

END;
```

9.2.12 - FILEGETNAME

Retourne le répertoire et le nom de fichier pointé par l'objet BFILE

En aucun cas elle ne teste l'existence physique du fichier

DBMS_LOB.FILEGETNAME (

file_loc IN BFILE,

repertoire OUT VARCHAR2,

fichier OUT VARCHAR2)

repertoire indique le répertoire Oracle

fichier indique le nom du fichier

Exceptions générées

VALUE_ERROR L'un des paramètres en entrée est NULL ou invalide

INVALID_ARGVAL Le paramètre repertoire ou fichier est NULL

Voir l'exemple 3 en fin de chapitre

9.2.13 - FILEISOPEN

Vérifie si le BFILE a été ouvert

Cette fonction retourne 1 si le fichier est ouvert, sinon elle retourne 0

DBMS_LOB.FILEISOPEN (

file_loc IN BFILE)

RETURN INTEGER

Exceptions gérées

NOEXIST_DIRECTORY le répertoire n'existe pas

NOPRIV_DIRECTORY Privilèges insuffisants sur la Directory

INVALID_DIRECTORY le répertoire a été invalidé après l'ouverture du fichier

```
DECLARE
  fil BFILE;
  pos INTEGER;
  pattern VARCHAR2(20);
BEGIN
  SELECT f_lob INTO fil FROM lob_table
  WHERE key_value = 12;

  -- ouverture du fichier --
  ...
  IF (dbms_lob.fileisopen(fil)) THEN
    dbms_output.put_line( 'fichier ouvert' ) ;
    dbms_lob.fileclose(fil);
  ELSE
    dbms_output.put_line( 'fichier non ouvert' ) ;
  END IF;
END;
```

9.2.14 - FILEOPEN

Ouverture d'un BFILE en lecture seule

DBMS_LOB.FILEOPEN (

file_loc IN OUT NOCOPY BFILE,

open_mode IN BINARY_INTEGER := file_readonly)

Exceptions générées

VALUE_ERROR Le paramètre file_loc ou open_mode est NULL

INVALID_ARGVAL open_mode différent de FILE_READONLY

OPEN_TOOMANY Trop de fichiers ouverts dans la session

NOEXIST_DIRECTORY le répertoire n'existe pas

INVALID_DIRECTORY le répertoire a été invalidé après l'ouverture du fichier

INVALID_OPERATION Fichier inexistant ou privilèges insuffisants

Voir les exemples 1 et 2 en fin de chapitre

9.2.15 - FREETEMPORARY

Libération d'un BLOB ou CLOB du TABLESPACE temporaire

DBMS_LOB.FREETEMPORARY (

lob_loc IN OUT NOCOPY BLOB)

DBMS_LOB.FREETEMPORARY (

lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS)

Voir l'exemple 2 en fin de chapitre

9.2.16 - GETCHUNKSIZE

Retourne la place utilisée dans le chunk pour stocker le LOB

Lorsqu'une table contenant une colonne LOB est créée, l'on peut indiquer le facteur de "chunking" qui doit être un multiple d'un bloc Oracle

Cet espace (chunk) est utilisé pour accéder ou modifier la valeur du LOB. Une partie de cet espace est réservé au système, l'autre au stockage de la valeur

La valeur retournée pour un BLOB est exprimée en octets, et en caractères pour un CLOB

DBMS_LOB.GETCHUNKSIZE (

lob_loc IN BLOB)

RETURN INTEGER

DBMS_LOB.GETCHUNKSIZE (

lob_loc IN CLOB CHARACTER SET ANY_CS)

RETURN INTEGER

9.2.17 - GETLENGTH

Retourne la longueur en octets d'un BLOB et en caractères d'un CLOB et la taille du fichier physique pour un BFILE

La valeur retournée est NULL si le LOB est NULL ou si le lob_loc est NULL

DBMS_LOB.GETLENGTH (

lob_loc IN BLOB)

RETURN INTEGER

DBMS_LOB.GETLENGTH (

lob_loc IN CLOB CHARACTER SET ANY_CS)

RETURN INTEGER

DBMS_LOB.GETLENGTH (

file_loc IN BFILE)

RETURN INTEGER

```
DECLARE
  lobd BLOB;
  length INTEGER;
BEGIN
  -- Initialisation du LOB
  SELECT b_lob INTO lobd FROM lob_table
  WHERE key_value = 42;

  length := dbms_lob.getlength(lobd);

  IF length IS NULL THEN
     dbms_output.put_line('LOB NULL');
  ELSE
     dbms_output.put_line('Taile du LOB : ' || length);
  END IF;
END;
```

9.2.18 - INSTR

Indication de la position d'un caractère ou d'une chaîne dans un LOB

DBMS_LOB.INSTR (

lob_loc IN BLOB,

pattern IN RAW,

déplacement IN INTEGER := 1,

nth IN INTEGER := 1)

RETURN INTEGER

DBMS_LOB.INSTR (

lob_loc IN CLOB CHARACTER SET ANY_CS,

pattern IN VARCHAR2 CHARACTER SET lob_loc%CHARSET,

déplacement IN INTEGER := 1,

nth IN INTEGER := 1)

RETURN INTEGER

DBMS_LOB.INSTR (

file_loc IN BFILE,

pattern IN RAW,

déplacement IN INTEGER := 1,

nth IN INTEGER := 1)

RETURN INTEGER

pattern représente une variable de type RAW pour les BLOB et BFILE ou une variable de type VARCHAR2 pour les CLOB

déplacement représente l'octet de départ (BLOB) ou le caractère de départ (CLOB) de la recherche (défaut 1)

nth représente le numéro d'occurrence à rechercher (défaut 1)

Cette fonction retourne la position du premier octet ou caractère recherché, ou retourne la valeur 0 si pattern n'est pas trouvé

Elle retourne NULL pour les conditions suivantes :

- Un ou plusieurs paramètres en entrée sont NULL ou invalides
- déplacement < 1 ou déplacement > LOBMAXSIZE
- nth < 1
- nth > LOBMAXSIZE

Exceptions générées

UNOPENED_FILE Fichier non ouvert

NOEXIST_DIRECTORY le répertoire n'existe pas

NOPRIV_DIRECTORY Privilèges insuffisants sur le répertoire

INVALID_DIRECTORY le répertoire a été invalidé après l'ouverture du fichier

INVALID_OPERATION Fichier inexistant ou privilèges insuffisants

```
DECLARE
  lobd CLOB;
  pattern VARCHAR2 := 'abcde';
  position INTEGER := 10000;
BEGIN
  -- Initialisation du LOB
  SELECT b_col INTO lobd
  FROM lob_table
  WHERE key_value = 21;

  position := DBMS_LOB.INSTR(lobd,pattern, 1, 1);

  IF position = 0 THEN
     dbms_output.put_line('Pattern non trouvée');
  ELSE
     dbms_output.put_line('Pattern trouvée en position ' || position);
```

```
    END IF;
  END;
```

9.2.19 - ISOPEN

Vérification de l'état d'ouverture d'un LOB

Cette fonction retourne la valeur TRUE si le LOB est temporaire, sinon elle retourne la valeur FALSE

DBMS_LOB.ISOPEN (

lob_loc IN BLOB)

RETURN INTEGER

DBMS_LOB.ISOPEN (

lob_loc IN CLOB CHARACTER SET ANY_CS)

RETURN INTEGER

DBMS_LOB.ISOPEN (

file_loc IN BFILE)

9.2.20 - ISTEMPORARY

Vérifie si le LOB est temporaire

DBMS_LOB.ISTEMPORARY (

lob_loc IN BLOB)

RETURN INTEGER

DBMS_LOB.ISTEMPORARY (

lob_loc IN CLOB CHARACTER SET ANY_CS)

RETURN INTEGER

9.2.21 - LOADFROMFILE

Copie de tout ou partie d'un fichier externe dans un LOB

DBMS_LOB.LOADFROMFILE (

dest_lob IN OUT NOCOPY BLOB,

src_file IN BFILE,

amount IN INTEGER,

dest_déplacement IN INTEGER := 1,

src_déplacement IN INTEGER := 1)

dest_lob représente le LOB de destination

src_file représente le BFILE source

amount représente le nombre d'octets à lire

dest_déplacement représente le décalage d'écriture dans le LOB destinataire (défaut 1)

src_déplacement représente le décalage de lecture depuis le fichier externe (défaut 1)

Exceptions générées

VALUE_ERROR L'un des paramètres en entré est NULL ou invalide

INVALID_ARGVAL si:

- src_déplacement ou dest_déplacement < 1
- src_déplacement ou dest_déplacement > LOBMAXSIZE
- amount < 1
- amount > LOBMAXSIZE

```
DECLARE
  lobd BLOB;
  fils BFILE := BFILENAME('SOME_DIR_OBJ','some_file');
  amt INTEGER := 4000;
BEGIN
  SELECT b_lob INTO lobd FROM lob_table WHERE key_value = 42 FOR UPDATE;
  dbms_lob.fileopen(fils, dbms_lob.file_readonly);
  dbms_lob.loadfromfile(lobd, fils, amt);
  COMMIT;
  dbms_lob.fileclose(fils);
END;
```

9.2.22 - LOADBLOBFROMFILE

Copie de tout ou partie d'un fichier externe dans un BLOB

DBMS_LOB.LOADBLOBFROMFILE (

dest_lob IN OUT NOCOPY BLOB,

src_bfile IN BFILE,

amount IN INTEGER,

dest_déplacement IN OUT INTEGER,

src_déplacement IN OUT INTEGER)

les valeurs pour **amount**, **dest_déplacement** et **src_déplacement** sont exprimées en octets

Voir les exemples 1 et 2 en fin de chapitre

9.2.23 - LOADCLOBFROMFILE

Copie de tout ou partie d'un fichier externe dans un CLOB

DBMS_LOB.LOADCLOBFROMFILE (

dest_lob IN OUT NOCOPY BLOB,

src_bfile IN BFILE,

amount IN INTEGER,

dest_déplacement IN OUT INTEGER,

src_déplacement IN OUT INTEGER,

src_csid IN NUMBER,

lang_context IN OUT INTEGER,

warning OUT INTEGER)

les valeurs pour **amount** et **src_déplacement** sont exprimées en octets

dest_déplacement représente le décalage en caractères d'écriture dans le CLOB

src_déplacement représente le décalage en octets de lecture depuis le fichier externe

src_csid représente le jeu de caractères du fichier externe

lang_context représente le code de la langue

warning représente l'éventuel code d'erreur

Exceptions générées

VALUE_ERROR L'un des paramètres en entrée est NULL ou invalide

INVALID_ARGVAL si:

- src_déplacement ou dest_déplacement < 1
- src_déplacement ou dest_déplacement > LOBMAXSIZE
- amount < 1
- amount > LOBMAXSIZE

Voir les exemples 1 et 2 en fin de chapitre

9.2.24 - OPEN

Ouverture d'un LOB

DBMS_LOB.OPEN (

lob_loc IN OUT NOCOPY BLOB,

mode IN BINARY_INTEGER);

DBMS_LOB.OPEN (

lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,

mode IN BINARY_INTEGER);

DBMS_LOB.OPEN (

file_loc IN OUT NOCOPY BFILE,

mode IN BINARY_INTEGER := file_readonly);

mode représente l'une des valeurs suivantes

- file_readonly
- lob_readonly
- lob_readwrite

9.2.25 - READ

Lecture de tout ou partie d'un LOB

DBMS_LOB.READ (

lob_loc IN BLOB,

amount IN OUT NOCOPY BINARY_INTEGER,

déplacement IN INTEGER,

tampon OUT RAW)

DBMS_LOB.READ (

lob_loc IN CLOB CHARACTER SET ANY_CS,

amount IN OUT NOCOPY BINARY_INTEGER,

déplacement IN INTEGER,

tampon OUT VARCHAR2 CHARACTER SET lob_loc%CHARSET)

DBMS_LOB.READ (

file_loc IN BFILE,

amount IN OUT NOCOPY BINARY_INTEGER,

déplacement IN INTEGER,

tampon OUT RAW)

amount représente le nombre d'octets (BLOB) ou de caractère (CLOB) à lire

déplacement représente le décalage en octets (BLOB) ou en caractères (CLOB)

tampon reçoit les octets (BLOB) ou caractères (CLOB) lus

Exceptions générées

VALUE_ERROR Le ou les paramètres lob_loc, amount, ou déplacement sont NULL

INVALID_ARGVAL si:

- amount < 1
- amount > MAXBUFSIZE
- déplacement < 1
- déplacement > LOBMAXSIZE
- amount supérieur, en octets ou caractères, à la capacité du tampon

NO_DATA_FOUND Détection de fin de LOB

Exceptions générées pour BFILE

UNOPENED_FILE Fichier non ouvert

NOEXIST_DIRECTORY le répertoire n'existe pas

NOPRIV_DIRECTORY Privilèges insuffisants sur le répertoire

INVALID_DIRECTORY le répertoire a été invalidé après l'ouverture du fichier

INVALID_OPERATION Fichier inexistant ou privilèges insuffisants

```
DECLARE
  src_lob BLOB;
  buffer RAW(32767);
  amt BINARY_INTEGER := 32767;
  pos INTEGER := 2147483647;
BEGIN
  SELECT b_col INTO src_lob
  FROM lob_table
  WHERE key_value = 21;
  LOOP
     dbms_lob.read (src_lob, amt, pos, buffer);
     -- traitement du tampon
     -- .....
     pos := pos + amt;
  END LOOP;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    dbms_output.put_line('Fin des données');
```

```
END;
```

9.2.26 - SUBSTR

Extraction d'une partie d'un LOB

DBMS_LOB.SUBSTR (

lob_loc IN BLOB,

amount IN INTEGER := 32767,

déplacement IN INTEGER := 1)

RETURN RAW

DBMS_LOB.SUBSTR (

lob_loc IN CLOB CHARACTER SET ANY_CS,

amount IN INTEGER := 32767,

déplacement IN INTEGER := 1)

RETURN VARCHAR2 CHARACTER SET lob_loc%CHARSET

DBMS_LOB.SUBSTR (

file_loc IN BFILE,

amount IN INTEGER := 32767,

déplacement IN INTEGER := 1)

RETURN RAW

amount représente le nombre d'octets (BLOB) ou de caractères (CLOB) à lire

déplacement représente le décalage en octets (BLOB) ou en caractères (CLOB) depuis le début du LOB

Les valeurs retournées sont

RAW pour les BLOB ou BFILE

VARCHAR2 pour les CLOB

NULL si:

- Un paramètre en entrée est NULL
- amount < 1
- amount > 32767
- déplacement < 1
- déplacement > LOBMAXSIZE

Exceptions générées

UNOPENED_FILE Fichier non ouvert

NOEXIST_DIRECTORY le répertoire n'existe pas

NOPRIV_DIRECTORY Privilèges insuffisants sur le répertoire

INVALID_DIRECTORY le répertoire a été invalidé après l'ouverture du fichier

INVALID_OPERATION Fichier inexistant ou privilèges insuffisants

```
DECLARE
  src_lob CLOB;
  pos INTEGER := 2147483647;
  buf VARCHAR2(32000);
BEGIN
  SELECT c_lob INTO src_lob FROM lob_table
  WHERE key_value = 21;

  buf := DBMS_LOB.SUBSTR(src_lob, 32767, pos);
  -- Traitement
  -- ...

END;
```

9.2.27 - TRIM

Réduction de la taille d'un LOB

DBMS_LOB.TRIM (

lob_loc IN OUT NOCOPY BLOB,

newlen IN INTEGER)

DBMS_LOB.TRIM (

lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,

newlen IN INTEGER)

newlen représente la taille finale désirée exprimée en octets (BLOB) ou en caractères (CLOB)

Exceptions générées

VALUE_ERROR Le paramètre lob_loc est NULL

INVALID_ARGVAL si:

- new_len < 0
- new_len > LOBMAXSIZE

```
DECLARE
  lob_loc BLOB;
BEGIN
  -- Initialisation du LOB
  SELECT b_col INTO lob_loc
  FROM lob_table
  WHERE key_value = 42 FOR UPDATE;

  -- réduction du LOB à 4000 octets --
  dbms_lob.trim(lob_loc, 4000);

  COMMIT;
END;
```

9.2.28 - WRITE

Ecriture dans un LOB

DBMS_LOB.WRITE (

lob_loc IN OUT NOCOPY BLOB,

amount IN BINARY_INTEGER,

déplacement IN INTEGER,

tampon IN RAW)

DBMS_LOB.WRITE (

lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,

amount IN BINARY_INTEGER,

déplacement IN INTEGER,

tampon IN VARCHAR2 CHARACTER SET lob_loc%CHARSET)

amount représente le nombre d'octets (BLOB) ou de caractères (CLOB) à écrire

déplacement représente le décalage en octets (BLOB) ou en caractères (CLOB) depuis le début du LOB

tampon représente la variable contenant les données à écrire

Exceptions générées

VALUE_ERROR L'un des paramètres en entrée lob_loc, amount, ou déplacement sont NULL, hors plage, ou invalides

INVALID_ARGVAL si:

- amount < 1
- amount > MAXBUFSIZE
- déplacement < 1
- déplacement > LOBMAXSIZE

```
DECLARE
  lob_loc BLOB;
  buffer RAW;
  amt BINARY_INTEGER := 32767;
  pos INTEGER := 2147483647;
  i INTEGER;
BEGIN
  SELECT b_col INTO lob_loc
  FROM lob_table
  WHERE key_value = 12 FOR UPDATE;

  FOR i IN 1..3 LOOP
    dbms_lob.write (lob_loc, amt, pos, buffer);
    -- ...
    pos := pos + amt;
  END LOOP;

END;
```

9.2.29 - WRITEAPPEND

Ajout de données à la fin d'un LOB

DBMS_LOB.WRITEAPPEND (

lob_loc IN OUT NOCOPY BLOB,

amount IN BINARY_INTEGER,

tampon IN RAW)

DBMS_LOB.WRITEAPPEND (

lob_loc IN OUT NOCOPY CLOB CHARACTER SET ANY_CS,

amount IN BINARY_INTEGER,

tampon IN VARCHAR2 CHARACTER SET lob_loc%CHARSET)

amount représente le nombre d'octets (BLOB) ou de caractères (CLOB) à écrire

tampon représente la variable contenant les données à écrire

Exceptions générés

VALUE_ERROR L'un des paramètres en entrée lob_loc, amount, ou déplacement sont NULL, hors plage, ou invalides

INVALID_ARGVAL si:

- amount < 1
- amount > MAXBUFSIZE

```
DECLARE
  lob_loc BLOB;
  buffer RAW;
  amt BINARY_INTEGER := 32767;
  i INTEGER;
BEGIN
  SELECT b_col INTO lob_loc
  FROM lob_table
  WHERE key_value = 12 FOR UPDATE;

  FOR i IN 1..3 LOOP
     -- Alimentation du tampon à ajouter au LOB
     -- ....
     dbms_lob.writeappend (lob_loc, amt, buffer);
  END LOOP;

END;
```

9.3 - Exceptions générées par le paquetage

- **invalid_argval 21560** Argument NULL invalide ou hors plage d'utilisation
- **access_error 22925** Tentative d'écriture de trop de données dans le LOB. La taille maxi est limitée à 4 gigabytes
- **noexist_directory 22285** Le répertoire désigné n'existe pas
- **nopriv_directory 22286** Privilèges insuffisants sur le répertoire ou sur le fichier
- **invalid_directory 22287** Le répertoire spécifié n'est pas valide ou a été modifiée par le DBA en cours d'utilisation
- **operation_failed 22288** L'opération demandée sur le fichier a échoué
- **unopened_file 22289** Tentative d'opération sur un fichier non ouvert
- **open_toomany 22290** Le nombre maximum de fichiers ouverts est atteint

9.4 - Exemples

- Exemple 1

Voici une procédure permettant de charger en table diverses colonnes LOB

Description de la table

```
CREATE TABLE DOCUMENT (
  ID           NUMBER (5) PRIMARY KEY,
  TYP          VARCHAR2 (20),
  UTILISE      VARCHAR2 (30),
  LOB_TEXTE    CLOB,
  LOB_DATA     BLOB,
  LOB_FICHIER  BFILE,
  NOM_DOC      VARCHAR2 (100))
```

La colonne ID stocke le numéro unique du document

La colonne TYP stocke le type du document

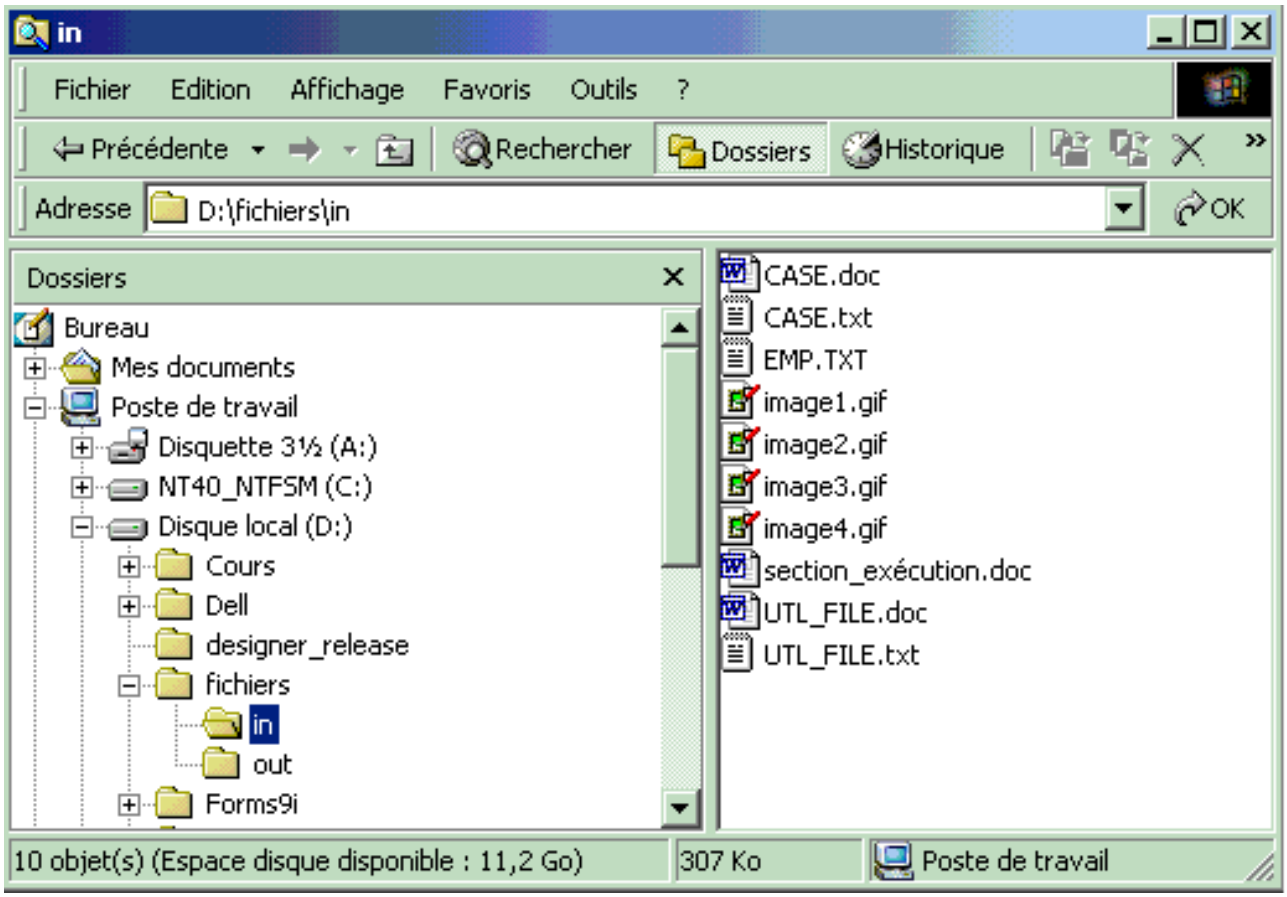
La colonne UTILISE stocke l'utilisateur qui modifie le document

La colonne LOB_TEXTE stocke un texte long

La colonne LOB_DATA stocke une image

La colonne LOB_FICHIER stocke un pointeur vers le fichier natif externe

Ainsi que les fichiers externes suivants



La procédure Insert_document() permet d'insérer une ligne dans la table et d'alimenter les colonnes LOB

```
SQL> CREATE OR REPLACE PROCEDURE Insert_document
  2   (
  3     PC$Type IN DOCUMENT.TYP%TYPE
  4    ,PC$Nom IN DOCUMENT.NOM_DOC%TYPE
  5    ,PC$Texte IN VARCHAR2
  6    ,PC$Image IN VARCHAR2
  7    ,PC$Fichier IN VARCHAR2
  8   ) IS
  9  L$Blob BLOB;
 10  L$Clob CLOB;
 11  L$Bfile BFILE;
 12  LN$Len NUMBER := dbms_lob.lobmaxsize;
 13  LN$Num NUMBER ;
 14  LN$src_off PLS_INTEGER := 1 ;
 15  LN$dst_off PLS_INTEGER := 1 ;
 16  LN$Langctx NUMBER := dbms_lob.default_lang_ctx ;
 17  LN$Warn NUMBER;
 18  BEGIN
 19    -- Création d'une nouvelle ligne --
 20    If PC$Fichier is not null Then
 21       L$Bfile := BFILENAME( 'FICHIERS_IN', PC$Fichier );
 22    End if ;
 23    Select SEQ_DOCUMENT.NEXTVAL Into LN$Num From DUAL ;
 24    Insert into DOCUMENT (ID, NOM_DOC, TYP, UTILISE, LOB_TEXTE, LOB_DATA, LOB_FICHIER)
 25           Values( LN$Num, PC$Nom, PC$Type, NULL, EMPTY_CLOB(), EMPTY_BLOB(), L$Bfile ) ;
 26    Select
 27      LOB_TEXTE
 28     ,LOB_DATA
 29    Into
 30      L$Clob
 31     ,L$Blob
```

```
 32    From
 33      DOCUMENT
 34    Where
 35      ID = LN$Num
 36    ;
 37    -- Chargement du texte dans la colonne CLOB --
 38    If PC$Texte is not null Then
 39       L$Bfile := BFILENAME( 'FICHIERS_IN', PC$Texte );
 40       dbms_lob.fileopen(L$Bfile, dbms_lob.file_readonly);
 41       If dbms_lob.fileexists( L$Bfile ) = 1 Then
 42         dbms_output.put_line(PC$Texte || ' ouvert' ) ;
 43         dbms_lob.loadclobfromfile(
 44                                   L$Clob,               -- CLOB de destination
 45                                   L$Bfile,              -- Pointeur fichier en entrée
 46                                   LN$Len,               -- Nombre d'octets à lire
 47                                   LN$src_off,           -- Position source de départ
 48                                   LN$dst_off,           -- Position destination de départ
 49                                   dbms_lob.default_csid, -- CSID
 50                                   LN$Langctx,           -- Contexte langue
 51                                   LN$Warn);             -- Message d'avertissement
 52         dbms_lob.fileclose(L$Bfile);
 53       Else
 54          raise_application_error( -20100, 'Erreur ouverture ' || PC$Texte ) ;
 55       End if ;
 56    End if ;
 57
 58    -- Chargement de l'image dans la colonne BLOB --
 59    If PC$Image is not null Then
 60       L$Bfile := BFILENAME( 'FICHIERS_IN', PC$Image );
 61       dbms_lob.fileopen(L$Bfile, dbms_lob.file_readonly);
 62       dbms_lob.loadblobfromfile(
 63                                 L$Blob,       -- BLOB de destination
 64                                 L$Bfile,      -- Pointeur de fichier en entrée
 65                                 LN$Len,       -- Nombre d'octets à lire
 66                                 LN$src_off,   -- Position source de départ
 67                                 LN$dst_off);  -- Position destination de départ
 68       dbms_lob.fileclose(L$Bfile);
 69    End if ;
 70
 71    -- Enregistrement --
 72    Update DOCUMENT
 73    Set
 74       LOB_TEXTE = L$Clob
 75       ,LOB_DATA = L$Blob
 76    Where
 77      ID = LN$Num ;
 78
 79  END;
 80  /

Procédure créée.

SQL>
```

Insérons une ligne dans la table

```
SQL>  execute Insert_document( 'Word','Document : CASE','CASE.txt','image1.gif','CASE.doc' ) ;
CASE.txt ouvert

Procédure PL/SQL terminée avec succès.
SQL> Select ID, NOM_DOC, TYP, LOB_TEXTE From DOCUMENT ;

        ID NOM_DOC
TYP
----------
--------------------------------------------------------------------------------------------------------
----
LOB_TEXTE
--------------------------------------------------------------------------------
         6 Document : CASE
Word
CASE

Cette instruction permet de mettre en place des structures de test condi
```

Dans cet exemple, nous initialisons les variables LOB par un select en table, qui implique que l'enregistrement existe, d'où l'instruction d'insertion en début de procédure

Nous pourrions tout aussi bien n'effectuer l'insertion en table qu'en fin de chargement des colonnes LOB

Dans ce cas, les variables LOB doivent avoir été préalablement initialisées avec la fonction CREATETEMPORARY comme dans l'exemple suivant

- Exemple 2

```
SQL> CREATE OR REPLACE PROCEDURE Insert_document2
  2   (
  3     PC$Type IN DOCUMENT.TYP%TYPE
  4    ,PC$Nom IN DOCUMENT.NOM_DOC%TYPE
  5    ,PC$Texte IN VARCHAR2
  6    ,PC$Image IN VARCHAR2
  7    ,PC$Fichier IN VARCHAR2
  8   ) IS
  9  L$Blob BLOB;
 10  L$Clob CLOB;
 11  L$Bfile BFILE;
 12  LN$Len NUMBER := dbms_lob.lobmaxsize;
 13  LN$Num NUMBER ;
 14  LN$src_off PLS_INTEGER := 1 ;
 15  LN$dst_off PLS_INTEGER := 1 ;
 16  LN$Langctx NUMBER := dbms_lob.default_lang_ctx ;
 17  LN$Warn NUMBER;
 18  BEGIN
 19    -- Création d'une nouvelle ligne --
 20    If PC$Fichier is not null Then
 21       L$Bfile := BFILENAME( 'FICHIERS_IN', PC$Fichier );
 22    End if ;
 23    -- Création des objets temporaires --
 24    dbms_lob.createtemporary( L$Clob, TRUE ) ;
 25    dbms_lob.createtemporary( L$Blob, TRUE ) ;
 26
 27    -- Chargement du texte dans la colonne CLOB --
 28    If PC$Texte is not null Then
 29       L$Bfile := BFILENAME( 'FICHIERS_IN', PC$Texte );
 30       dbms_lob.fileopen(L$Bfile, dbms_lob.file_readonly);
 31       If dbms_lob.fileexists( L$Bfile ) = 1 Then
 32         dbms_output.put_line(PC$Texte || ' ouvert' ) ;
 33         dbms_lob.loadclobfromfile(
 34                                   L$Clob,               -- CLOB de destination
 35                                   L$Bfile,              -- Pointeur fichier en entrée
 36                                   LN$Len,               -- Nombre d'octets à lire
 37                                   LN$src_off,           -- Position source de départ
 38                                   LN$dst_off,           -- Position destination de départ
 39                                   dbms_lob.default_csid, -- CSID
 40                                   LN$Langctx,           -- Contexte langue
 41                                   LN$Warn);             -- Message d'avertissement
 42         dbms_lob.fileclose(L$Bfile);
 43       Else
 44          raise_application_error( -20100, 'Erreur ouverture ' || PC$Texte ) ;
 45       End if ;
 46    End if ;
 47
 48    -- Chargement de l'image dans la colonne BLOB --
 49    If PC$Image is not null Then
 50       L$Bfile := BFILENAME( 'FICHIERS_IN', PC$Image );
 51       dbms_lob.fileopen(L$Bfile, dbms_lob.file_readonly);
 52       dbms_lob.loadblobfromfile(
 53                                 L$Blob,       -- BLOB de destination
 54                                 L$Bfile,      -- Pointeur de fichier en entrée
 55                                 LN$Len,       -- Nombre d'octets à lire
 56                                 LN$src_off,   -- Position source de départ
 57                                 LN$dst_off);  -- Position destination de départ
 58       dbms_lob.fileclose(L$Bfile);
 59    End if ;
 60
```

```
 61    -- Enregistrement --
 62    Insert into DOCUMENT (ID, NOM_DOC, TYP, UTILISE, LOB_TEXTE, LOB_DATA, LOB_FICHIER)
 63           Values( SEQ_DOCUMENT.NEXTVAL, PC$Nom, PC$Type, NULL, L$Clob, L$Blob, L$Bfile ) ;
 64    -- Libération des objets temporaires --
 65    dbms_lob.freetemporary( L$Clob ) ;
 66    dbms_lob.freetemporary( L$Blob ) ;
 67
 68  END;
 69  /

Procédure créée.

SQL>
```

- Exemple 3

Utilisation de la procédure FILEGETNAME pour afficher la directorie et le nom du fichier pointés par la colonne BFILE

```
SQL> CREATE OR REPLACE PROCEDURE Affiche_Infos_Fichier
  2  (
  3    PN$Id IN DOCUMENT.ID%TYPE
  4  )
  5  Is
  6  LC$Rep VARCHAR(100);
  7  LC$Fic VARCHAR2(100) ;
  8  L$Bfile BFILE;
  9  Begin
 10    Select LOB_FICHIER
 11    Into   L$Bfile
 12    From   DOCUMENT
 13    Where  ID = PN$Id
 14    ;
 15
 16    DBMS_LOB.FILEGETNAME (L$Bfile, LC$Rep, LC$Fic) ;
 17    Dbms_output.put_line( 'Dir=' || LC$Rep || '  Fic=' || LC$Fic ) ;
 18  End ;
 19  /

Procédure créée.
```

```
SQL> set serveroutput on
SQL> select ID from DOCUMENT ;

        ID
----------
         6

SQL> execute Affiche_Infos_Fichier( 6 ) ;
Dir=FICHIERS_IN  Fic=CASE.doc

Procédure PL/SQL terminée avec succès.
```

9.5 - Manipulations courantes des LOB de type caractères (CLOB)

La plupart des manipulations courantes sur les objets CLOB peuvent être effectuées avec les fonctions SQL

LENGTH

```
SQL> select length(lob_texte), length(lob_data) from document;

LENGTH(LOB_TEXTE) LENGTH(LOB_DATA)
----------------- -----------------
3131 8748
```

SUBSTR

```
SQL>  select substr(lob_texte,2000, 100) from document;

SUBSTR(LOB_TEXTE,2000,100)
--------------------------------------------------
/SQL terminée avec succès.


Cette fois l'opérateur est précisé sur chaque lign
```

INSTR

```
SQL> select instr(lob_texte, 'opérateur') from document;

INSTR(LOB_TEXTE,'OPÉRATEUR')
----------------------------
                         256
```

UPPER

```
SQL> select upper( substr( lob_texte, 1, 40 ) ) from document;

UPPER(SUBSTR(LOB_TEXTE,1,40))
-----------------------------------------------------------------
CASE


CETTE INSTRUCTION PERMET DE METT
```

LIKE

```
SQL> select id, nom_doc from document where lob_texte like '%opérateur%';

        ID NOM_DOC
---------- ----------------------------------------------------------------
         6 Document : CASE
```

Complément d'information

Vous pouvez également consulter l'article Les LOB par Helyos

Index de recherche

Liste alphabétique des mots clé

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Remerciements

Chaleureux remerciements à Developpez.com, l'équipe SGBD en général et Pomalaix en particulier.