

[PL/SQL ORACLE]

Les fondamentaux du langage procédural PL/SQL d'Oracle

Les Fondamentaux du langage PL/SQL

1	CONVENTION D'ÉCRITURE	7
I.	INTRODUCTION AU LANGAGE PL/SQL.....	8
I.1	STRUCTURE D'UN BLOC PL.....	9
I.1.1	LES COMMENTAIRES.....	10
I.1.2	BLOC NOMMÉ, BLOC ANONYME	10
I.1.3	LA SECTION DÉCLARATIVE	10
I.2	DÉCLARATION DE VARIABLE	10
I.2.1	OPÉRATEURS D'AFFECTATION.....	11
I.2.2	LES TYPES EN PL.....	11
I.2.3	MON PREMIER PROGRAMME PL/SQL !.....	12
II.	ÉCRIRE DES INSTRUCTIONS EXÉCUTABLES	14
II.1	LES COMMENTAIRES	14
II.2	IMBRICATIONS DE BLOCS.....	14
II.2.1	PORTÉE DES VARIABLES	14
III.	LES STRUCTURES DE CONTRÔLE.....	16
III.1	L'INSTRUCTION IF	16
III.1.1	CAS DE LA VALEUR NULL	16
III.2	L'INSTRUCTION CASE	16
III.3	CONTRÔLES D'ITÉRATIONS	17
III.3.1	L'INSTRUCTION LOOP	17
III.3.2	L'INSTRUCTION WHILE	17
III.3.3	L'INSTRUCTION FOR	17
III.3.4	BOUCLES IMBRIQUÉES	18
IV.	TYPES DE DONNÉES COMPOSITES	19
IV.1	ENREGISTREMENTS PL/SQL	19
IV.2	TABLEAUX ASSOCIATIFS (TABLES INDEX BY)	19
IV.2.1	MÉTHODES DES TABLES INDEX BY	20
IV.3	TABLE IMBRIQUÉE (NESTED TABLE).....	20
IV.4	TABLEAU DE TAILLE FIXE (VARRAY)	21
V.	LES CURSEURS	22
V.1	CURSEURS IMPLICITES	22
V.1.1	ATTRIBUTS D'UN CURSEUR IMPLICITE	22

V.2	CURSEURS EXPLICITES	22
V.3	ATTRIBUTS D'UN CURSEUR	22
V.4	LA BOUCLE FOR DE CURSEUR (CURSEUR IMPLICITE)	23
V.4.1	CURSEUR AVEC PARAMÈTRES	24
V.4.2	LE VERROU SELECT ... FOR UPDATE	24
VI.	LES EXCEPTIONS	26
VI.1	DÉFINITION.....	26
VI.2	SYNTAXE	26
VI.3	INTERCEPTER LES ERREURS NON PRÉDÉFINIES	27
VI.4	EXCEPTIONS DÉFINIES PAR LE DÉVELOPPEUR.....	27
VI.5	FONCTIONS LIÉES AUX EXCEPTIONS	27
VII.	LES PROCÉDURES ET FONCTIONS STOCKÉES	28
VII.1	LES PROCÉDURES	28
VII.2	LES FONCTIONS.....	28
VII.3	LES TRIGGERS	29
VIII.	LES PACKAGES	31
IX.	ANNEXES.....	34
IX.1	MANIPULER DES LOBs.....	34
IX.2	MODÈLE HR	35
IX.3	CONVERTIR UN FICHIER RTF EN TEXTE	36
X.	INDEX	39

Je n'ai jamais rencontré d'homme si ignorant qu'il n'eut quelque chose à m'apprendre. (Galilée)

AVANT PROPOS

Ce support de cours est un outil personnel, il ne constitue pas un guide de référence.

C'est un outil pédagogique élaboré dans un souci de concision : il décrit les actions essentielles à connaître pour appréhender le sujet de la formation.

Objectifs pédagogiques :

- Manipuler de l'information à l'aide des différentes opérations de l'algèbre relationnelle
- Appliquer les opérations de l'algèbre relationnelle aux requêtes SQL
- Composer des requêtes SQL simples
- Composer des requêtes SQL avancées

Prérequis : Être familier avec les concepts basiques des langages de programmation :

- Les fonctions
- Les types de variables (entier, décimal, réel, chaîne de caractères, booléen, date)
- Les opérateurs logiques (ET, OU, NON)
- Les booléens TRUE et FALSE

Si vous ne les connaissez pas, ce n'est pas encore rédhibitoire.



Bien que le langage SQL soit normalisé, il existe, néanmoins des différences entre les différents systèmes de gestion de base de données relationnelle (SGBDR).

Les exemples fournis dans ce support seront écrits avec la syntaxe SQL Oracle.

1 Convention d'écriture

La police `courier` est utilisée pour les exemples de commandes SQL :

```
SELECT last_name FROM Employees ;
```

Les MAJUSCULES sont utilisées pour les mots clé SQL.

Les minuscules sont utilisées pour les noms des colonnes et le nom des tables seront écrits avec l'initiale en Majuscule.

Les termes Oracle sont en *italiques*.

Dans la syntaxe d'une instruction :

```
SELECT [DISTINCT] { * | NomCol1 | ExprSQL [AS etiq1] [, Nomcol2 | ExprSQL [etiq2] ... ] }  
FROM NomTable ;
```

Les symboles suivants définissent :

- [] le caractère optionnel d'une directive
- { } une liste d'éléments alternatifs
- | le choix possible parmi une liste d'éléments
- AS (souligné) un terme par défaut

I. Introduction au langage PL/SQL

Le langage SQL est le langage utilisé permettant l'accès aux données et à leur modification dans les bases relationnelles. C'est un langage déclaratif non prévu pour :

- La gestion des variables et types
- Les instructions conditionnelles dans leur forme la plus générale
- Les boucles (instructions répétitives)
- La gestion des erreurs
- La création de bibliothèques de fonctions et procédures ou d'autres objets procéduraux
- Etc...

PL/SQL signifie « *Procedural Language extension to SQL* ». Ce langage propose une extension au SQL en introduisant une logique conditionnelle et répétitive. Il propose toutes les structures de programmation procédurales disponibles dans les langages de programmation de 3^{ème} génération (3GL).

Ainsi, le PL/SQL sera utilisé par le développeur.

Inspiré des langages **Ada**¹ ou **Pascal**, ce langage est apparu dans la version 7 d'Oracle Database avec l'option procédurale. À l'origine interprété, il est depuis la version 9i, compilé.

Le moteur Oracle n'est pas seulement un moteur SQL mais aussi PL/SQL. Et ce moteur PL se trouve au-dessus du moteur SQL.

Le langage PL/SQL ajoute des structures de contrôles et de procédures au langage SQL comme :

- Des variables, constantes et types de données
- Des structures de contrôle, telles que les instructions conditionnelles et les boucles.
- Programmes réutilisables

De plus, il est possible d'utiliser les commandes des langages DML (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) et DCL (`COMMIT`, `ROLLBACK`, `SAVEPOINT`) dans un programme PL/SQL.

La commande **SELECT** ne sera plus utilisée pour afficher les résultats d'une requête mais pour affecter des valeurs à des variables.

Par contre, les commandes du langage DDL (`CREATE`, `ALTER`, `DROP`) ne sont pas autorisées.

¹ Langage de programmation développé par le français Jean Ichbiah à la demande du département de la Défense des États-Unis (DOD). Le nom "Ada" a été choisi en l'honneur d'Ada Lovelace (*10/12/1815 †27/11/1852) considérée comme la première informaticienne.

Il est néanmoins possible d'exécuter des commandes DDL ou DCL par l'utilisation du package **DBMS_SQL**.

Les procédures PL pourront être stockées en base et profiteront des ressources de l'architecture du serveur BASE DE DONNÉES et de la proximité des données.

Le code PL/SQL généré est du p-code (*Pre-compiled code*). Depuis la version 11g ce code est stocké dans le *TABLESPACE SYSTEM*.

On appelle objet procédural les objets suivants :

- Les fonctions
- Les procédures
- Les « packages »
- Les triggers

I.1 Structure d'un bloc

Le code PL/SQL est écrit sous forme de bloc défini comme suit :

- Une partie déclarative (optionnelle)
- Une partie exécutable (corps du bloc).
- Une partie de traitement des exceptions (erreurs générées lors de l'exécution et optionnelle).

À l'intérieur de la partie exécutable, on pourra trouver du SQL encapsulé « *embedded sql* »

```
SQL
IF ... THEN
    SQL
ELSE
    SQL
END IF
...
```

[DECLARE]

Variables, constantes, curseurs, types, exceptions définies par l'utilisateur

BEGIN

- Instructions SQL
- Instructions PL/SQL

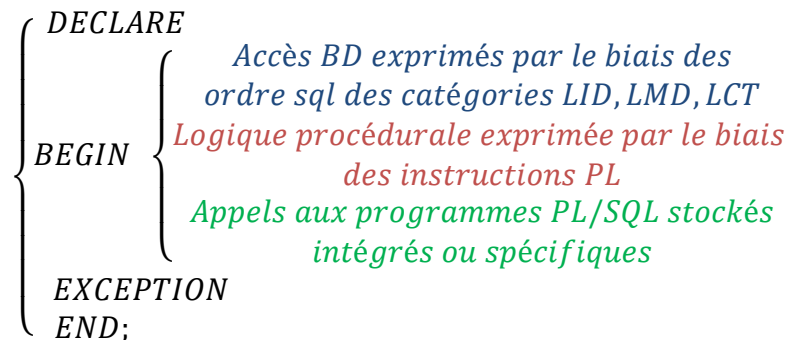
[EXCEPTION]

Actions à entreprendre lorsque des exceptions (erreurs pendant l'exécution) se produisent

END ;

Il existe 2 langages PL :

- Le PL serveur
- Le PL enrichi à l'environnement de développement (Forms, Reports,...).



I.1.1 Les commentaires

Les commentaires seront :

- Sur une ligne, précédés par `--` (double tiret)
- Sur plusieurs lignes, encadrés par `/*` et `*/`

I.1.2 Bloc nommé, bloc anonyme

Le bloc nommé commence soit par les termes :

- PROCEDURE
- FUNCTION

La procédure et la fonction sont interchangeable mais la forme d'appel n'est pas la même.

En réalité, la procédure fait un traitement et la fonction ramène un résultat.

I.1.3 La section déclarative

Il existe 5 types de variables :

- Les types équivalents aux types SQL Oracle : `NUMBER`, `VARCHAR`, `DATE`, ...
- Les types déclarés à partir des objets oracle, colonnes ou tables (`%TYPE`, `%ROWTYPE`). On parlera de type référencé.
- Les booléens : `BOOLEAN` et les constantes `TRUE` et `FALSE`.
- Les types composites ou enregistrements (`RECORD`)
- Les tableaux de valeurs (`TABLE`)

Structure de données

- Colonne de table : `Nom_table.Nom_Col` (en SQL seulement et non en PL)
- Variable PL (dans un ordre PL ou SQL) `Nom_Var`

I.2 Déclaration de variable

Il existe 3 natures de variables :

- Les variables PL/SQL
- Les variables SQL*Plus préfixées par `&` (l'esperluette)

- Les *bind variables*, variables d'un langage hôte, préfixées par : (deux points)

Les variables dans la partie déclarative seront typées et éventuellement initialisées.

Nom_Var <<type>> [:= valeur] ;

Nom_Var [CONSTANT] type [NOT NULL] [:= | DEFAULT ExprPL] ;

ExprPL ::= < opérande1 [opérateur opérande] >

1.2.1 Opérateurs d'affectation

Il existe 2 opérateurs d'affectation :

- :=
- SELECT ... INTO ...

Exemple : l'assignation sql :

```
SELECT expr1, expr2, ... exprn
INTO var1, var2, ... varn
FROM ...
```

La commande **ACCEPT** permet d'inviter l'utilisateur à saisir une donnée.

```
ACCEPT v_ville PROMPT "Saisir une ville : "
```

À chaque fonction SQL (substr, ...) Oracle fournit sa contrepartie en PL.

Exemple : *vResu* := round(*vsal**12.15) ;

1.2.2 Les types en PL

Dans la partie déclarative, il est possible de définir de nouveaux types en plus des types connus du SQL :

- Scalaire
- Référencé
- LOB (Large Object)
- Composite (tableau, tableau associatif, enregistrement)
- Variables non PL/SQL : variables attachées ou variables applicatives déclarées dans un environnement externe (appelé aussi, Host variable ou Bind variable).

:NomVarAppli

Syntaxe de déclaration de type :

<<type>> := < tous types SQL /
NomTable.NomCol%TYPE /
NomTable%ROWTYPE /

BOOLEAN / PLS_INTEGER, BINARY_INTEGER >

%TYPE type de la colonne ou de la variable, l'utilisation de ce type est particulièrement intéressante pour coller les types de variables à d'éventuelles modifications de type de colonnes.

%ROWTYPE type de l'enregistrement. L'utilisation de ce type est particulièrement intéressante pour coller les types de variables à d'éventuelles modifications de type de colonnes.

NomTable%ROWTYPE définit un type identique à la table.

Exemple :

```
vclient CLIENT%ROWTYPE ;    -- variable de type enregistrement
                             -- implicite de même structure qu'une
                             -- ligne de la table CLIENT
vclient.nom := SUBSTR(vclient.nom,1,3) ;
```

Exemple de type référencé :

```
Debit      NUMBER(8,2) ;
Credit     Debit%TYPE ;
-- la variable Credit est du même type que la variable Debit
```

I.2.3 Mon premier programme PL/SQL !



Pour soumettre le résultat à l'interface utilisateur où on exécute le programme on devra ouvrir le canal et afficher le message :

```
SET SERVEROUTPUT ON ;
DBMS_OUTPUT.PUT_LINE ('chaine') ;
```

```
SET SERVEROUTPUT ON

DECLARE
v_Nom employees.last_name%TYPE ;
v_Mesg VARCHAR2(80) ;
BEGIN
SELECT last_name INTO v_Nom FROM employees
WHERE employee_id=100 ;
v_Mesg := 'Hello ' || v_Nom || '! Aujourd'hui nous sommes le ' ||
TO_CHAR(sysdate, 'DD/MM/YYYY') ;
dbms_output.put_line (v_Mesg) ;
END;
```


II. Écrire des instructions exécutables

Un programme PL/SQL est composé :

- De commentaires
- D'instructions SQL
- D'instructions PL/SQL

II.1 Les commentaires

Les commentaires s'expriment par :

- Double tiret --
- /* ...
... */

II.2 Imbrications de blocs

Un programme PL/SQL peut être composé de plusieurs blocs imbriqués. Pour faciliter la lecture, il est possible d'indiquer des étiquettes de blocs.

La propagation des exceptions sera étudiée par la suite.

BEGIN <<NomBloc>>

DECLARE

BEGIN

 DECLARE

 BEGIN

 END ;

END ;

END NomBloc ;

II.2.1 Portée des variables

Une variable est connue dans le bloc où elle est déclarée et dans tous les sous blocs.

- Si la variable est modifiée dans un bloc interne la modification est propagée vers les blocs externes.
- Si elle est déclarée de nouveau, sa portée sera locale à ce bloc.
- L'appartenance d'une variable à un bloc sera définie en la préfixant par le nom de son bloc.

Exemple :

```
<<bloc1>>
DECLARE
    x NUMBER := 1;
    y NUMBER := 2;
BEGIN
    <<bloc2>>
    DECLARE
        y NUMBER := 4;
        z NUMBER := 3;
    BEGIN
        dbms_output.put_line( 'Variable x de premier: '||x );
        dbms_output.put_line( 'Variable y de premier: '||bloc1.y );
        dbms_output.put_line( 'Variable y de second: '||y );
        bloc1.y := 5;
    END;
    dbms_output.new_line;
    dbms_output.put_line( 'Variable y de premier: '||y );
END;
```

III. Les structures de contrôle

- Structure de contrôle conditionnelle : **IF ... END IF ;**
- Itérative : **LOOP ... END LOOP ;**

III.1 L'instruction IF

Syntaxe élémentaire :

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

Syntaxe complète :

```
IF condition THEN
  instructions ;
[ ELSIF condition THEN
  instructions ; ]
[ ELSE
  instructions ; ]
END IF ;
```

III.1.1 Cas de la valeur NULL

Si la condition renvoie NULL, dans ce cas, l'instruction de contrôle passe à l'instruction ELSE.

III.2 L'instruction CASE

Il existe deux syntaxes pour l'instruction CASE : l'expression d'affectation et l'instruction conditionnelle.

1^{ère} Syntaxe : L'expression CASE

```
Var := CASE selecteur
  WHEN expression1 THEN result1
  WHEN expression2 THEN result2
  ...
  WHEN expressionN THEN resultN
  [ ELSE resultN+1 ]
END ;
```

Dans ce cas, l'expression renvoie une valeur qu'on devra affecter à une variable.

2^{ème} syntaxe : L'instruction CASE

```
CASE selecteur
  WHEN expression1 THEN result1
  WHEN expression2 THEN result2
  ...
```



```
WHEN expressionN THEN resultN  
[ ELSE resultN+1 ]  
END CASE ;
```

III.3 Contrôles d'itérations

Il existe plusieurs formes d'instructions itératives :

- L'instruction LOOP ... END LOOP
- L'instruction WHILE LOOP ... END LOOP
- L'instruction FOR LOOP ... END LOOP

III.3.1 L'instruction LOOP

Syntaxe :

```
[ <<label>> ] LOOP  
    instruction1 ;  
    ...  
    EXIT [ WHEN condition ] ;  
    instructionN ;  
    ...  
END LOOP [ <<label>> ] ;
```

La sortie de cette boucle ne peut être exécutée que par l'instruction EXIT à l'intérieure de celle-ci.

III.3.2 L'instruction WHILE

Syntaxe :

```
[ <<label>> ] WHILE condition LOOP  
    instruction1 ;  
    instruction2 ;  
    ... ;  
END LOOP [ <<label>> ] ;
```

III.3.3 L'instruction FOR

Syntaxe :

```
[ <<label>> ] FOR indice IN [ REVERSE ] debut..fin  
LOOP  
    instruction1 ;  
    instruction2 ;  
    ...  
END LOOP [ <<label>> ] ;
```

Le pas est obligatoirement de 1.

L'indice peut être consulté mais non modifié dans la boucle et il est implicitement du type NUMBER.



L'indice n'est connu qu'à l'intérieur de l'instruction FOR.

III.3.4 Boucles imbriquées

Dans ce cas, il sera judicieux d'étiqueter les boucles. L'étiquette sera encadrée par les délimiteurs << >> et placée avant le mot LOOP et dans les boucles FOR et WHILE avant ces mots FOR WHILE.

L'instruction EXIT provoque un débranchement de la boucle (sortie complète de la boucle).

L'instruction CONTINUE provoque une nouvelle itération de la boucle.

<pre> FOR i IN 1..10 LOOP instruction1 ; instruction2 ; CONTINUE WHEN i > 5 ; instruction3 ; ... END LOOP ; instruction4 ; ... </pre> 	<pre> FOR i IN 1..10 LOOP instruction1 ; instruction2 ; EXIT WHEN i > 5 ; instruction3 ; ... END LOOP ; instruction4 ... </pre> 
INSTRUCTION CONTINUE	INSTRUCTION EXIT

Il est possible de débrancher de plusieurs niveaux, dans ce cas on devra utiliser des étiquettes de boucles.

Ce type de débranchement est déconseillé pour des raisons de lisibilité du code écrit.

IV. Types de données composites

- Enregistrements (RECORD)
- Tableaux (TABLE)

IV.1 Enregistrements PL/SQL

Syntaxe en 2 étapes :

- Déclaration du type
- Déclaration de la variable de ce type enregistrement.

```
TYPE nom_type IS RECORD
  (declaration_champ1
    [, declaration_champ2 ]
    ... ) ;
v_enregistrement nom_type ;
```

declaration_champ ::= { nom_champ | variable%TYPE | table.col%TYPE |
table%ROWTYPE }
[[NOT NULL] { := | DEFAULT } expr]

Un enregistrement %ROWTYPE permet de simplifier le code en le compactant. Dans ce cas, il sera inutile de déclarer autant de variables qu'il y a de colonne dans la table.

Ainsi, les instructions INSERT et UPDATE pourront s'écrire simplement.

IV.2 Tableaux associatifs (tables INDEX BY)

Tableau de 2 colonnes :

- Clé primaire d'index du tableau de type entier ou chaîne
- Colonne contenant un type scalaire ou RECORD.

Syntaxe, déclaration du type et déclaration de la variable :

```
TYPE nom_type IS TABLE OF
  { type_col | variable%TYPE
    | table.col%TYPE } [ NOT NULL ]
  | table%ROWTYPE
  | INDEX BY PLS_INTEGER | BINARY_INTEGER | VARCHAR2 (taille) ;
var_type_table nom_type ;
```

Pour référencer la ligne *n* du tableau associatif, écrivez *mon_tableau(n)*.

Le tableau sera utilisé pour stocker des données temporaires.

Exemple :

```
TYPE dept_table_type IS TABLE OF
departments.department_name%TYPE
INDEX BY PLS_INTEGER ;
```

```
my_dept_table dept_table_type ;
```

IV.2.1 Méthodes des tables INDEX BY

Les méthodes suivantes permettent de parcourir, compter et supprimer les lignes du tableau.

- EXISTS (n)
- COUNT
- FIRST
- LAST
- PRIOR (n)
- NEXT (n)
- DELETE [(m [,n])]

IV.3 TABLE imbriquée (NESTED TABLE)

Il s'agit d'une table dans une table donc stockée en base.

```
TYPE dept_table_type IS TABLE OF nom_du_type
```

La différence de déclaration réside dans l'absence de la clause **INDEX BY** PLS_INTEGER | BINARY_INTEGER

```
DECLARE TYPE t_tab_imb IS TABLE OF VARCHAR2(80);
v_tab_imb t_tab_imb := t_tab_imb();
BEGIN
-- Initialisation de la table imbriquée.
v_tab_imb.extend;
v_tab_imb(indice) := 'message1';
END;
```

IV.3.1 Méthodes prédéfinies pour les tables imbriquées

COUNT	Nombre d'éléments
DELETE	Suppression de tous les éléments
DELETE(n)	Suppression de l'élément d'indice n
DELETE(m, n)	Suppression des éléments d'indices compris entre m et n
EXISTS	Existence d'un élément. TRUE si l'élément existe.
EXTEND	Ajout d'un élément en fin de table imbriquée
EXTEND(n)	Ajout de n éléments en fin de table imbriquée
EXTEND(n , i)	Ajout de n copies de l'élément d'indice i en fin de table
FIRST	Indice du premier élément
LAST	Indice du dernier élément
NEXT	Indice du prochain élément
PRIOR	Indice de l'élément précédent

TRIM	Suppression d'un élément en fin de table imbriquée
TRIM(n)	Suppression de n éléments en fin de table imbriquée

IV.4 Tableau de taille fixe (VARRAY)

Tableau de type associatif dont on définit sa taille lors de sa déclaration. Dès lors, le nombre d'éléments contenus dans ce tableau sera fixe.

```
TYPE employees_type IS VARRAY(10) OF employees.last_name%TYPE ;  
Liste_emp employees_type;
```

La variable Liste_emp contiendra au maximum 10 valeurs. Si ce nombre d'éléments est dépassé, le message d'erreur « *Subscript outsideof limit* » vous sera renvoyé.

V. Les curseurs

V.1 Curseurs implicites

Ces curseurs sont gérés par le noyau Oracle et créés lors d'une commande DML.

Ils représentent le nom d'un emplacement mémoire utilisé par Oracle pour analyser un objet.

Un curseur PL/SQL est une zone nommée et se comporte comme un pointeur sur une ligne.

Les curseurs PL/SQL sont des curseurs "avant" (*Forward-only*).

La gestion des curseurs peut être implicite ou explicite.

V.1.1 Attributs d'un curseur implicite

SQL%FOUND	Booléen, vaut TRUE si la dernière instruction SQL a affectée au moins une ligne
SQL%NOTFOUND	Booléen, vaut TRUE si la dernière instruction SQL n'a affectée aucune ligne
SQL%ROWCOUNT	Valeur entière qui représente le nombre de lignes affectées par la dernière instruction SQL

V.2 Curseurs explicites

Dès lors que vous voudrez l'ensemble des lignes d'une requête SQL, il vous faudra utiliser un curseur.

Contrairement à un curseur implicite qui sera déclaré et géré par le compilateur, un curseur explicite sera déclaré et géré par le développeur.

1. Déclaration du curseur
2. Ouverture du curseur
3. Demande d'acquisition de ligne, ligne par ligne.
4. Libération des ressources consommées par le curseur par sa fermeture.

1. **CURSOR** Nom_Curseur IS SELECT ... ;
2. **OPEN** Nom_Curseur ;
3. **FETCH** Nom_Curseur INTO variable ;
4. **CLOSE** Nom_Curseur ;

V.3 Attributs d'un curseur

Nom_Curseur%FOUND	Booléen, vaut TRUE si la dernière instruction SQL a affectée au moins une ligne
Nom_Curseur %NOTFOUND	Booléen, vaut TRUE si la dernière instruction SQL n'a affecté aucune ligne

Nom_Curseur %ROWCOUNT	Valeur entière qui représente le nombre de lignes affectées par la dernière instruction SQL
Nom_Curseur%ISOPEN	Booléen, vaut TRUE si le curseur est ouvert (OPEN).

Exemple de syntaxe :

DECLARE

```

CURSOR c_emp_cursor IS
  SELECT employe_id, last_name FROM employees
  WHERE department_id = 30 ;

BEGIN
  OPEN c_emp_cursor ;
  LOOP
    FETCH c_emp_cursor INTO v_empno, v_lname ;
    EXIT WHEN c_emp_cursor%NOTFOUND ;
  END LOOP ;
  CLOSE c_emp_cursor ;
END ;
/

```

Déclaration d'une variable enregistrement du type source du curseur :

v_emp_record c_emp_cursor%**ROWTYPE** ;

V.4 La boucle FOR de curseur (curseur implicite)

1^{ère} syntaxe :

```

FOR Nom-Rec IN Nom_Curseur
LOOP
  instruction1 ;
  ...
END LOOP ;

```

NOTE : la variable *Nom-Rec* est créée implicitement du type *Nom_Curseur%RowType*.

2^{ème} syntaxe :

```

FOR Nom-Rec IN ( SELECT ... FROM ... )
LOOP
  instruction1 ;
  ...
END LOOP ;

```

De cette façon, vous n'avez plus à gérer l'ouverture, le fetch, l'atteinte de la fin du curseur et sa fermeture qui se font implicitement par la boucle FOR.

Idéal pour faire des balayages complets de tables.

V.4.1 Curseur avec paramètres

```
DECLARE
    CURSOR c_emp_cursor (deptno NUMBER) IS
    SELECT employee_id, last_name
    FROM employees
    WHERE department_id = deptno ;
    ...
BEGIN
    OPEN c_emp_cursor (10) ;
    ...
    CLOSE c_emp_cursor ;
    OPEN c_emp_cursor (20) ;
    ...
```

V.4.2 Le verrou SELECT ... FOR UPDATE

Pose un verrou de type « ROW SHARE » qui sera transformé en « ROW EXCLUSIVE » pour l'utilisateur qui sera le premier à utiliser la ressource.

V.4.2.1 La clause WHERE CURRENT OF

Fait référence à la ligne en cours du curseur explicite ouvert avec la clause FOR UPDATE.

Cette instruction permet de référencer une ligne dans une table correspondant à l'enregistrement courant du curseur.

```
DECLARE
    CURSOR c_emp is SELECT ROUND(salary/1000) FROM emp
                     FOR UPDATE;
    v_asterisk    emp.stars%TYPE := NULL;
    v_sal         emp.salary%TYPE;
BEGIN
    OPEN c_emp;
    LOOP
        FETCH c_emp INTO v_sal;
        EXIT WHEN c_emp%NOTFOUND;
        v_asterisk := NULL;
        FOR i IN 1..v_sal
        LOOP
            v_asterisk := v_asterisk || '*';
        END LOOP ;
    END LOOP ;
```



```
        UPDATE emp SET stars = v_asterisk
        WHERE CURRENT OF c_emp;
    END LOOP ;
    CLOSE c_emp;
    COMMIT;

END;
/
```

VI. Les exceptions

VI.1 Définition

Une exception est une erreur PL/SQL provoquée lors de l'exécution du programme.

Elle peut être associée à une instruction exécutée dans le cas d'une anomalie du bloc PL/SQL.

Une exception est soit :

- Implicite et gérée par le noyau
- Explicite et gérée par le développeur

Une exception sera traitée dans le bloc PL le plus proche contenant la section EXCEPTION par propagation sinon par le programme appelant.

Les exceptions implicites se classe en 2 catégories :

- Prédéfinie par le serveur (ex : NO_DATA_FOUND, TOO_MANY_ROWS, ZERO_DIVIDE...)
- Non prédéfinie.

Les exceptions explicites sont, quant à elles, définies par le développeur.

VI.2 Syntaxe

DECLARE

...

BEGIN

...

EXCEPTION

WHEN exception1 [OR exception2 ...] **THEN**

instruction1 ;

instruction2 ;

... ;

[**WHEN** Exception3 **THEN** ...]

[**WHEN OTHERS THEN**

instruction1 ;

instruction2 ; ...]

END ;

Oracle fournit des noms d'exceptions prédéfinies pour les plus courantes.

WHEN OTHERS THEN ... Signifie toutes les exceptions non listées précédemment (équivalent du ELSE du CASE).

VI.3 Intercepter les erreurs non prédéfinies

1. Déclarer
2. Associer
3. Traiter

DECLARE

e_except1 EXCEPTION ;

PRAGMA EXCEPTION_INIT (e_except1, -01400) ;

BEGIN

...

IF condition **THEN RAISE e_except1 ;**

END IF ;

...

EXCEPTION

WHEN e_except1 THEN

...

END ;

/

VI.4 Exceptions définies par le développeur

3 étapes sont nécessaires :

1. Déclarer
2. Déclencher
3. Référencer

L'instruction **RAISE** permet de lever (déclencher) une exception dans le bloc PL.

VI.5 Fonctions liées aux exceptions

Il existe 2 fonctions liées aux exceptions qui permettent de connaître le code d'une erreur et son message.

- **SQLCODE**
- **SQLERRM**

VII. Les procédures et fonctions stockées

Une procédure stockée est un ensemble d'instructions désigné par un nom compilé et stocké sur le serveur.

Il en est de même pour les fonctions stockées.

Les avantages des procédures stockées sont les suivants :

- Un seul envoi : Le client envoie seulement une requête d'exécution pouvant contenir plusieurs ordres SQL et ainsi n'utilise qu'une seule connexion au serveur.
- Une centralisation des traitements : Tous les applicatifs peuvent utiliser les procédures stockées quel que soit le langage d'écriture, la plateforme d'exécution.
- Une plus grande rapidité d'exécution : Du fait de la pré-compilation et de l'envoi d'appels plutôt que de requêtes SQL les applicatifs sont d'exécution plus rapides.

Il existe 4 types d'objets procéduraux :

- Les procédures
- Les fonctions
- Les packages
- Les triggers

VII.1 Les procédures

```
CREATE [OR REPLACE] PROCEDURE nom_procedure  
[(nom_parametre [IN | OUT | IN OUT] type [, ...])]  
{IS | AS} var1 type1;  
BEGIN  
< corps_de_la_procedure >  
EXCEPTION  
< traitement des exceptions >  
END [nom_procedure];
```

VII.2 Les fonctions

```
CREATE [OR REPLACE] FUNCTION nom_fonction  
[(nom_parametre [IN | OUT | IN OUT] type [, ...])]  
RETURN type_resultat  
{IS | AS} var1 type1;  
BEGIN  
< corps_de_la_fonction >  
RETURN (valeur)  
END [nom_fonction];
```

VII.3 Les triggers

Un trigger est un morceau de code qui sera déclenché en fonction d'un évènement.

Il existe plusieurs types de triggers :

- Les triggers sur les tables
- Les triggers sur les vues
- Les triggers sur les actions utilisateur
- Les triggers sur les actions système

Un trigger sur table s'applique aux mises à jour (*Insert, Update, Delete*).

Il peut être déclenché avant (contrôle a priori par exemple) ou après (instructions cascade) l'action de mise à jour.

Sur l'action Update il est possible de préciser une colonne. Dans ce cas il ne sera déclenché que lorsque la modification aura lieu sur cette colonne.

Lorsque plusieurs mises à jour sont effectuées il est possible de ne déclencher le trigger qu'une seule fois ou pour chaque ligne affectée. C'est la clause ***For Each Row*** qui détermine le comportement du trigger.

Enfin le trigger ne peut être déclenché que lorsqu'une colonne prend une valeur particulière ; c'est la dernière clause ***WHEN*** ; cette clause ne peut être spécifiée que lorsque la clause *For Each Row* est elle-même présente.

Toutes ces combinaisons impliquent qu'il est possible de créer de multiples *triggers* sur une même table.

```
CREATE [OR REPLACE] TRIGGER nom_trigger  
[BEFORE | AFTER | INSTEAD OF ]  
[UPDATE [OR] | INSERT [OR] | DELETE]  
[OF nom_col]  
ON NOM_TABLE  
[FOR EACH ROW]  
[WHEN (condition)]
```

Bloc-PL/SQL

Exemple

```
CREATE TRIGGER maj_employees  
BEFORE  
DELETE OR INSERT OR UPDATE  
ON employees  
FOR EACH ROW  
DECLARE  
...  
BEGIN
```

```
if inserting then
insert into journal
values (:new.employees_id,sysdate);
elsif deleting then
insert into journal
values (:old.employees_id,sysdate);
end if;
END ;
/
```

VIII. LES PACKAGES

Un package est un regroupement de procédures ou de fonctions ayant un lien logique entre elles, ou gérées de la même façon : accorder un droit d'exécution sur un package donne le droit d'exécuter tous les éléments déclarés dans la partie spécifications.

Différents éléments peuvent constituer un PACKAGE :

- Procédure,
- Fonctions,
- Variable,
- Curseur,
- Constante,
- Exception.

Un package est construit en deux parties :

- Une partie spécification,
- Une partie corps (*BODY*).

La partie spécification contient les signatures des procédures et fonctions publiques du package. Ce sont des interfaces. Les variables qui sont déclarées dans cette partie sont globales (à l'utilisateur). Les fonctions, procédures, variables de la partie *spécification* sont globales à l'utilisateur.

La partie body contient les procédures et les fonctions spécifiées dans le bloc *spécification*. Les déclarations de cette deuxième partie sont internes au package.

Il est aussi possible de créer des fonctions, des procédures, des variables locales dans le *body*.

Deux types de déclarations à l'intérieur d'un PACKAGE :

- Déclarations de type public :
 - Accessibles par tous les utilisateurs autorisés,
 - Déclarées dans la partie SPECIFICATION,
 - Définies dans la partie BODY.
- Déclarations de type privé :
 - Accessibles uniquement par les composants du PACKAGE,
 - Déclarées et définies uniquement dans la partie BODY.

Appel des éléments d'un PACKAGE

Exemple

```
CREATE [OR REPLACE] PACKAGE gere_employe AS
FUNCTION embauche (nom VARCHAR2, boulot VARCHAR2, chef NUMBER,
                  date_emb DATE, sal NUMBER comm NUMBER,
                  service NUMBER)

RETURN NUMBER;
PROCEDURE debauche (emp_id NUMBER);
END gere_employe;
/

CREATE PACKAGE BODY gere_employe AS
FUNCTION embauche (nom VARCHAR2, boulot VARCHAR2, chef NUMBER,
                  hiredate DATE, sal NUMBER, comm NUMBER,
                  Service NUMBER)

RETURN NUMBER
IS
new_empno NUMBER (10);
BEGIN
SELECT emp_sequence.NEXTVAL INTO new_empno
FROM dual;
INSERT INTO emp VALUES (new_empno, nom,
boulot, chef, date_emb, sal, comm, service);
RETURN (new_empno);
END embauche;

PROCEDURE debauche (emp_id IN NUMBER)
IS
BEGIN
DELETE FROM emp WHERE empno = emp_id;
IF SQL%NOTFOUND THEN
raise_application_error (-20011, 'numéro
employé inconnu'||TO_CHAR (emp_id));
END IF;
END debauche;
END gere_employe;
/
```

Appel

- Depuis un applicatif :

```
nom_procedure (paramètres);
```

- Appel explicite :

```
EXECUTE nom_procedure (paramètres);
```

• Modification

En cas de modification d'objets référencés dans un package, le noyau ORACLE recompilera automatiquement tout le package.

On pourra recompiler manuellement le package.

```
ALTER PACKAGE nom-package COMPILE [PACKAGE | BODY] ;
```

- **Suppression**

```
DROP PACKAGE nom-package ;
```

- **Avantages d'utilisation**

- Sécurité :

- Accès uniquement aux déclarations de type public,

- Les droits d'exécution ne sont donnés que sur le package et non plus individuellement sur tous les composants du package.

- Etat persistant :

- Conservation des valeurs des variables pour toute une session,

- Conservation des contextes des curseurs pour toute une session.

- Performance :

- Réduction du nombre d'appels à la base,

- Seul le premier appel charge en mémoire tout le package.

- Productivité :

- Stockage dans la même entité des fonctions et des procédures

- Gestion facile de l'organisation des développements.

IX. ANNEXES

IX.1 Manipuler des LOBs

Pour illustrer l'utilisation des types LOB (*Large Object*), nous utiliserons la table suivante :

```
CREATE TABLE tablob
(idtablob number(5),
 nom varchar2(100),
 texte clob);

declare
v_clob varchar2(32767);
begin
for i in 1..1000 loop
v_clob := v_clob||'abcdefghijklmnopqrstuvwxyz';
end loop;
insert into tablob values (1,'A',v_clob);
commit;
end;
```

Exemple de procédure qui utilise le package *dbms_lob* pour rechercher et remplacer du texte dans un CLOB.

```
-- 1) clob src - the CLOB source to be replaced.
-- 2) replace str - the string to be replaced.
-- 3) replace with - the replacement string.

FUNCTION replaceClob (
srcClob IN CLOB,
replaceStr IN VARCHAR2,
replaceWith IN VARCHAR2)
RETURN CLOB IS

vBuffer      VARCHAR2 (32767);
l_amount     BINARY_INTEGER := 32767;
l_pos        PLS_INTEGER := 1;
l_clob_len   PLS_INTEGER;
newClob      CLOB := EMPTY_CLOB;

BEGIN
  -- initialize the new clob
  dbms_lob.createtemporary(newClob,TRUE);

  l_clob_len := dbms_lob.getlength(srcClob);
```

```

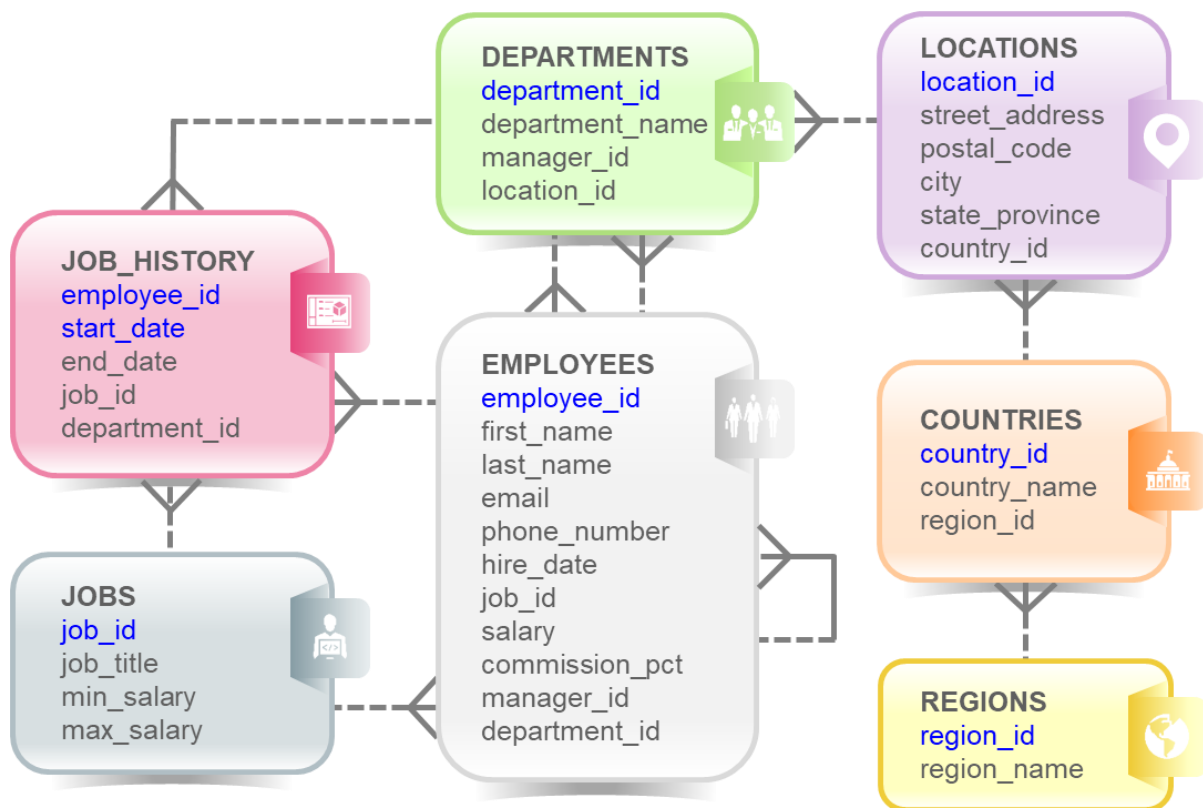
WHILE l_pos <= l_clob_len
LOOP
    dbms_lob.read(srcClob, l_amount, l_pos, vBuffer);

    IF vBuffer IS NOT NULL THEN
        -- replace the text
        vBuffer := replace(vBuffer, replaceStr, replaceWith);
        -- write it to the new clob
        dbms_lob.writeappend(newClob, LENGTH(vBuffer), vBuffer);
    END IF;
    l_pos := l_pos + l_amount;
END LOOP;

RETURN newClob;
EXCEPTION
    WHEN OTHERS THEN
        RAISE;
END;
/

```

IX.2 Modèle HR



IX.3 Convertir un fichier RTF en texte

```
CREATE TABLE demo
( id int PRIMARY KEY,
  theblob blob,
  theclob clob
);
```

Table created.

```
CREATE TABLE filter ( query_id number, document clob );
```

Table created.

```
CREATE INDEX demo_idx ON demo(theblob) INDEXTYPE is
ctxsys.context;
```

Index created.

```
CREATE SEQUENCE s;
```

Sequence created.

```
CREATE OR REPLACE DIRECTORY my_files AS
'/home/tkyte/Desktop/'
```

Directory created.

@trace

```
alter session set events '10046 trace name context forever, level 12';
```

Session altered.

```
DECLARE
  l_blob blob;
  l_clob clob;
  l_id number;
```

```

l_bfile bfile;
BEGIN
  insert into demo values ( s.nextval, empty_blob(),
empty_clob() )
  returning id, theblob, theclob into l_id, l_blob, l_clob;

  l_bfile := bfilename( 'MY_FILES', 'asktom.rtf' );
  dbms_lob.fileopen( l_bfile );

  dbms_lob.loadfromfile( l_blob, l_bfile,
dbms_lob.getlength( l_bfile ) );

  dbms_lob.fileclose( l_bfile );

  ctx_doc.ifilter( l_blob, l_clob );
  commit;
  ctx_doc.filter( 'DEMO_IDX', l_id, 'FILTER', l_id, TRUE );
END;
/

```

PL/SQL procedure successfully completed.

set long 500

select utl_raw.cast_to_varchar2(dbms_lob.substr(theblob,500,1)) from demo;

UTL_RAW.CAST_TO_VARCHAR2(DBMS_LOB.SUBSTR(THEBLOB,500,1))

```

-----
{\rtf1\ansi\ansicpg1252\uc1 \deff0\deflang1033\deflangfe1033{\fonttbl{\f0\froma
n\fcharset0\fprq2{\*\panose 02020603050405020304}Times New Roman;}{\f1\fswiss\fa
charset0\fprq2{\*\panose 020b0604020202020204}Arial;}
{\f2\fmmodern\fcharset0\fprq1{\*\panose 02070309020205020404}Courier New;}{\f23\
froman\fcharset128\fprq1{\*\panose 00000000000000000000}MS Mincho{\*\falt MS ??
};}{\f28\froman\fcharset128\fprq1{\*\panose 00000000000000000000}@MS Mincho;}
{\f29\froman\fcharset238\fprq2 Times New Roman CE;

```

select theclob from demo;

THECLOB

```

-----
<HTML><BODY>
<h1><font size="5" face="Arial">Primary key index in Partitioning</font>
</h1>

```

I have a table accounts which has 80 million records (OLTP system). I would like to partition the table by acct_by_date column. I will be going with range partition and global indexes. My concern is regd the primary key acct_id. The index that will be created for primary key should it be local or global and which should I opt for?

select document from filter;

DOCUMENT

Primary key index in Partitioning

I have a table accounts which has 80 million records (OLTP system). I would like to partition the table by acct_by_date column. I will be going with range partition and global indexes. My concern is regd the primary key acct_id. The index that will be created for primary key should it be local or global and which should I opt for?

Well, this is an easy one. The primary key index can be local IF and ONLY IF, the primary key is in fact the (or part of the

X. INDEX

<i>%ROWTYPE</i>	10
<i>%TYPE</i>	10
<i>Ada</i>	8
<i>BEGIN</i>	9
<i>CASE</i>	14
<i>CONTINUE</i>	16
<i>Curseurs explicites</i>	19
<i>CURSOR</i>	19
<i>CLOSE</i>	19
<i>FETCH</i>	19
<i>OPEN</i>	19
<i>DBMS_SQL</i>	8
DECLARE	9
<i>Données composites</i>	17
<i>END</i>	9
<i>enregistrement</i>	10
<i>Enregistrements</i>	Voir RECORD
EXCEPTION	9, 23
<i>EXCEPTION_INIT</i>	Voir PRAGMA
<i>EXIT</i>	16
<i>ExprPL</i>	11
<i>FOR</i>	15
<i>Forms</i>	9
<i>IF</i> 14	
<i>LOOP</i>	15
<i>Pascal</i>	8
<i>PRAGMA</i>	24
<i>RAISE</i>	24
<i>RECORD</i>	10, 17
<i>Reports</i>	9
<i>ROWTYPE</i>	11
<i>TABLE</i>	10
<i>type référencé</i>	10
<i>types composites</i>	10
<i>WHILE</i>	15