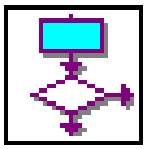
heg Haute école de gestion
de Neuchâtel

Formation professionnelles supérieures en
Informatique de gestion

Informaticien de Gestion / HES

Base de données I

Support de cours



Introduction au langage
PL/SQL

Document	sqlprog.doc
Date de création	10/09/97
Date de dernière modif.	29/11/01
Date d'impression	04/12/01
Version	2.2
Créé par	E.Meylan
E-Mail	Eddy.Meylan@hegne.ch

Introduction

SQL est un langage non procédural, il permet de façon simple et aisée à un utilisateur, de manipuler la base de données sans en spécifier le « comment ».

A l'opposé des langages comme C, ADA ou Pascal, sont des exemples de langages procéduraux et exigent de l'utilisateur des procédures (les algorithmes) d'accès aux données.

Un langage procédural est plus complexe à utiliser qu'un langage non procédural, mais il offre plus de flexibilité et de puissance.

Pour profiter des avantages du procédural et du non procédural, Oracle offre une extension procédurale à SQL appelée PL/SQL.

Le présent document est destiné aux utilisateurs ayant connaissance d'au moins un langage procédural.

Convention d'écriture

MAJUSCULES	Tous les mots clés d'Oracle sont écrits en majuscules Exemple : SELECT, INSERT, ...
italiques	Les variables de substitution sont représentées en italiques. Une variable de substitution est une chaîne de caractères à laquelle sera substituée une valeur d'un type donné Exemple *nom_table*, *nom_attribut*, ...
[]	Les crochets [] indiquent une présence optionnelle des éléments qu'ils contiennent Exemple IS [NOT] NULL
{}	Les crochets {} indiquent le choix d'un seul élément parmi la liste contenue à l'intérieur de ces accolades. Les éléments de la liste sont séparés par une barre verticale Exemple : {ARCHIVELOG|NOARCHIVELOG}
...	Les trois points indiquent que l'élément qui les précède peut être répété. Exemple : *nom_attribut*,...

Présentation de PL/SQL.

Un programme écrit en PL/SQL est composé de trois parties comme le montre le schéma ci dessous :

Syntaxe d'un bloc PL/SQL

[<<nom_de_bloc >>]

[**DECLARE**

Déclarations]
BEGIN

Commandes exécutables
[**EXCEPTION**

traitement des exceptions]
END[nom_de_bloc];

Une instruction PL/SQL se termine toujours par un point virgule (;).

Une ligne commentaire commence par le double signe -- et se termine à la fin de la ligne. Les commentaires de bloc sont délimités par les caractères /* et */

Exemple :

```
-- Ceci est un commentaire sur une ligne
/* Ceci est
un commentaire
formé de plusieurs lignes */
```

La partie « Déclarations » sert a définir les variables et les constantes utilisés dans le bloc; elle est optionnelle, et est délimitée par le mot **DECLARE** qui spécifie le début, et **BEGIN** qui signifie la fin.
La partie « Commandes exécutables » et « Gestion des exceptions » constituent le corps du bloc PL/SQL. La partie « Commandes exécutables » est obligatoire, alors que la « Gestion des exceptions » est optionnelle. Elles sont délimitées par les mot **BEGIN, EXCEPTION** et **END**.

La suite d'instruction d'une structure de bloc peut elle-même contenir d'autres blocs, puisque un bloc peut toujours être écrit en lieu et place d'une instruction.

Exemple :

```
DECLARE
  déclarations ;
  ..
BEGIN
  instruction_executable ;
  ..
    DECLARE
      déclarations ;
    BEGIN
      instruction_executable ;
      ....
    EXCEPTION
      gestion_d_exception ;
    END ;
..instruction_executable ;
  ...
EXCEPTION
  gestion_d_exception ;
END ;
```

Partie déclarations:

PL/SQL est un langage à typage fort et par conséquent toute entité de programmation doit être déclarée.

Déclaration de constantes et variables

```
declaration_de_variable ::=
nom_variable [CONSTANT]
     {type | table.attribut%Type | table%Rowtype}
     [NOT NULL][{:= |DEFAULT}expression_PL/SQL];
```

Exemple:

```
     DECLARE
          -- variable pour le nom d'un client courant
     nom_client     Varchar(15) ;
          -- constante rabais égale à 0.1
     rabais    CONSTANT Number(3,2) := 0.1;  DECLARE
          -- variable pour le nom d'un client courant
     nom_client     Varchar(15) ;
          -- constante rabais égale à 0.1
     rabais    CONSTANT Number(3,2) := 0.1;
```

Les Types de données

PL/SQL offre deux variétés de types de données prédéfinies, les types scalaires et composés.

Types scalaires

Liste de types et de sous-types prédéfinie en PL/SQL.

TYPE	SOUS-TYPE	
NUMBER	DECIMAL DOUBLE_PRECISION FLOAT INTEGER NUMERIC REAL SMALLINT	
BINARY_INTEGER	NATURAL POSITIVE POSITIVE	-2^{31} -1 à 2^{31} -1
CHAR(n)	CHARACTER STRING	
VARCHAR2(n)	VARCHAR	
DATE		
BOOLEAN		True ou False

Types composés

Définition d'un type enregistrement.

```
TYPE nom_type IS RECORD
     (attribut {type_attribut | table.attribut%Type}
     [NOT NULL] ,...)
```

Exemple :

```
DECLARE
    TYPE ty_commande IS RECORD
          (num_comm      Number(5,0) NOT NULL     := 0,
           num_cli       client.numero%Type,
           date_commande Date NOT NULL       := Sysdate);
...
```

Une fois le type enregistrement défini, il est possible de déclarer des variables ou des paramètres formels de procédure et de fonction comme étant de ce type.

Exemple:

```
DECLARE
    commande_courante ty_commande;
```

Pour atteindre un champ de la variable enregistrement commande_courante, on utilise la notation pointée. Par exemple pour affecter la date du jour au champ date_commande on écrira l'instruction suivante:

```
commande_courante.date_commande := Sysdate;
```

Définition d'un type tableau.

```
TYPE nom_type IS TABLE OF
     {type_attibut | table.attribut%Type} [NOT NULL]
     INDEX BY Binary_Integer ;
```

Exemple :

```
DECLARE
    TYPE ty_prix_commande IS TABLE OF Number(5.2)
          INDEX BY Binary_Integer;

          prix_commande ty_prix_commande;
```

Pour atteindre une case du tableau nombre_commande on utilisera la notation:
prix_commande(i)
L'indice du tableau, i est une variable déclarée de type Binary_Integer.
Actuellement, le tableau ne peut être que d'une dimension

Types basés sur une relation

Pour manipuler des variables de même type que les attributs d'une relation ou une structure correspondant au schéma de la relation d'une table ou d'une vue, PL/SQL offre deux attributs **%Type** et **%Rowtype**. Ainsi la déclaration suivante:

```
DECLARE
    TYPE type_commande IS RECORD
    (num_comm commande.num_comm%Type NOT NULL := 0,
     num_cli commande.num_cli%Type NOT NULL := 0,
     date_comm commande.date_commande%Type := Sysdate);
```

définit un type enregistrement dont les champs sont de même type que les colonnes de la table commande.

Il est encore tout à fait possible de passer outre la définition du type et de déclarer une variable enregistrement dont le type est la déscription d'une ligne de la table commande.

```
DECLARE
    commande_courante    commande%Rowtype;
```

Conversions de types

Les fonctions de conversions de types sont les suivantes

	Char	Date	Number	Raw	Rowid
Char		To_Date	To_Number	HexToRaw	CharToRowid
Date	To_Char				
Number	To_Char	To_Date			
Raw	RawToHex				
Rowid	RowidToChar				

Visibilité des variables

Les règles de portée des variables sont assez classiques.

a) Une variable est connue du bloc dans lequel elle est déclarée, et de tous les blocs internes à ce bloc.

b) Si une variable est redéclarée(avec le même nom, mais un type et une valeur initiale éventuellement différents) dans un sous bloc, c'est cette nouvelle déclaration qui prime sur la précédente.

```
DECLARE
    a  CHAR;
    b  REAL;
BEGIN
    -- Variables accessibles ici a (CHAR), b


    DECLARE
        a  INTEGER;
        c  REAL;
    BEGIN
        -- Variables accessibles ici a(INTEGER), b, c
    END;

    DECLARE
        d  REAL;
    BEGIN
        -- Variables accessibles ici a(CHAR), b, d
    END;

    -- Variables accessibles ici a(CHAR), b
END;
```

Le préfixe du nom de bloc devant la variable, permet de forcer la référence sur une variable externe au bloc :

```
<<bloc_princ>>
DECLARE
    birthdate  DATE;
BEGIN
    ...
    DECLARE
        birthdate  DATE;
    BEGIN
        ...
        IF birthdate = bloc_princ.birthdate THEN

            ...
        END IF;
    END;
END outer;
```

Partie des commandes exécutables

Les commandes du langage PL/SQL recouvre le LMD de SQL (INSERT,UPDATE,DELETE,SELECT) les commandes de gestion des transactions du LCD (COMMIT, ROLLBACK), mais ne supporte pas le LDD de SQL (CREATE,ALTER,DROP) ainsi que les instructions de gestion de la sécurité (GRANT, REVOKE) en standard.

Affectation de données provenant de la base à des variables PL/SQL.

PL/SQL permet l'affectation de données tirées de tables Oracle à l'aide d'une requête SQL à des structures de donnée d'un programme PL/SQL. Cependant cette affectation dépendra du résultat retourné par la requête. Dans le cas où le résultat n'est constitué que d'une ligne d'une table, une variable enregistrement sera une structure adaptée, mais dans le cas ou le résultat est une relation, c'est à dire un ensemble de tuple une structure de curseur est nécessaire.

```
commande_select ::=
     SELECT liste_selection INTO {liste_variable|nom_enreg}
     FROM liste_de_tables
     [condition_de_recherche] ;
```

Affectation dans le cas où le résultat est une ligne d'une table.

Exemple:

```
DECLARE
    commande_courante    commande%Rowtype;
BEGIN
    SELECT num_comm, num_cli, date_commande
          INTO commande_courante.num_comm,
               commande_courante.num_cli
               commande_courante.date_commande
          FROM commande
          WHERE num_comm = 453;
```

Attention : Si la requête ramène 0 ou plusieurs valeurs, une erreur est provoquée. Cette commande est donc à utiliser dans des cas bien précis.

Structure conditionnelle IF

```
IF condition THEN sequence_de_commandes
    [ELSIF condition THEN sequence_de_commandes,...]
    [ELSE sequence_de_commandes]
END IF ;
```

La structure conditionnelle la plus simple est l'instruction **IF**; sous sa forme élémentaire, on peut l'écrire:

```
IF expression_booléenne THEN

    suite_d_instructions ;

END IF;
```

La suite d'instructions ne sera exécutée que dans le cas où l'expression booléenne est vraie.

Branche ELSE

```
IF expression_booléenne THEN

    suite_d_instructions_1 ;

ELSE

    suite_d_instructions_2 ;

END IF ;
```

La suite d'instructions 2 ne serra exécuté que lorsque la condition booléenne est fausse. Les suites d'instructions 1 et 2 sont donc exclusives.

Branche ELSIF

La branche elsif permet d'enchaîner les conditions sans augmenter le niveau d'imbrication:

```
IF conditon_1 THEN
    suite_d_instructions_1.;
ELSIF condition_2 THEN
    suite_d_instructions_2.;
ELSE
    suite_d_instructions_3.;
END IF;
```

Structure de boucle

Instruction loop

La boucle est toujours signalée par le mot clé **LOOP** et se termine par **END LOOP;**

```
LOOP

suite_d_instructions
END LOOP;
```

Boucle d'itération for

Il s'agit ici de répéter un certain nombre de fois, connu à l'exécution, un ensemble d'instructions. Une variable, dite variable de boucle, permet de compter le nombre de cycles. Ce comptage s'effectue en parcourant un type énuméré quelconque, une sous-type INTEGER par exemple.

```
FOR compteur IN [REVERSE] borne_inf..borne_sup LOOP
    suite_d_instructions
END LOOP;
```

Boucle d'itération while

La boucle tourne tant qu'une condition est vraie. Cette condition est testé avant chaque tour.

```
WHILE condition LOOP

    suite_d_instructions
END LOOP;
```

Instruction de sortie exit

On peut sortir de n'importe quel type de boucle par l'instruction exit. Cette instruction peut se trouver à n'importe quel niveau d'imbrication par rapport au niveau de la boucle.

```
LOOP
    suite_d_instructions
    EXIT WHEN condition
END LOOP;
```

Exemple :

```
<<boucle_princ>>
    LOOP
          ....
          LOOP
               suite_d_instructions
               EXIT boucle_princ WHEN condition
          END LOOP;
          suite_d_instructions
    END LOOP <<boucle_princ>>;
```

Branchement

La structure de PL/SQL est telle que l'instruction GOTO peut parfois simplifier la programmation.

L'instruction GOTO effectue un branchement inconditionnel a une étiquette.
L'étiquette (label) doit être unique, et précéder une instruction exécutable, ou un bloc PL/

Exemple :

```
BEGIN
    ...
    GOTO insert_row;
    ...
    <<insert_row>>
    INSERT INTO emp VALUES ...
END;
```

L'utilisation de l'instruction GOTO peut engendrer un programmation compliquée et non structurée, difficile à concevoir et surtout à maintenir. Cette instruction est à utiliser avec bon sens.

Instruction nulle

L'instruction NULL spécifie explicitement l'inaction.
Exemple :

```
IF rating > 90 THEN
        INSERT INTO bonus VALUES(emp_id);
    ELSE
        NULL;
    END IF;
```

Curseur Définition

Pour pouvoir exécuter une requête, le système de gestion de base de donnée alloue dynamiquement et gère une zone mémoire appelée contexte. Le contexte contient les informations nécessaires l'exécution de la requête SQL. A savoir, le résultat de l'interprétation de la requête, l'adresse des variables hôtes , le statut d'exécution de la requête, etc..
Cette zone peut être manipulée par l'intermédiaire d'un identificateur appelé un curseur. Il s'agit en fait d'un pointeur sur le contexte.

On peut donc voir un curseur est une structure de données mémoire, capable de stoker le résultat d'une requête sur une relation.
Cette structure est donc particulièrement bien adaptée à la manipulation de données en mémoire lorsque le nombre de lignes ramenées par un SELECT est inconnu, ce qui est généralement le cas dans la pratique.

Le concept de curseur est fondamental en base de données, on le trouve également en programmation avec langage hôte avec un pré-compilateur. Dans le présent chapitre, nous nous limiterons à l'étude des curseurs avec le langage PL/SQL, mais les mécanismes sont identiques avec d'autres langages.

Traitement du curseur

A chaque ordre Pour traiter une commande SQL, PL/SQL ouvre une zone de contexte pour exécuter la commande et stocker les données. Le curseur permet de nommer cette zone de contexte, d'accéder aux données et éventuellement contrôler le traitement.
PL/SQL utilise deux type de curseurs :

Les curseurs statiques dont l'instruction est connue lors de la compilation et les pointeurs de curseurs qui sont dynamiques. Pour les curseurs statiques, on distingue encore les curseurs explicites ou implicites.

- Les curseurs explicites pour traiter les requêtes multi-lignes
- Les curseurs implicites utilisés pour les autres commandes SQL

Curseurs statiques explicites

Pour chaque requête SELECT on associe un curseur nommé et on y référence dans la suite du programme pour traiter les lignes les unes après les autres.
On peut donc maintenir autant de curseurs ouvert simultanément que les ressources système le permettent et économiser les phases d'analyse du noyau.

Pour utiliser un curseur, il faut :
Le déclarer dans la partie déclarative, l'ouvrir dans la partie commandes, ensuite défiler l'extraction des données. Une fois le traitement terminé, il faut refermer le curseur pour libérer les ressources.

Déclaration du curseur

La déclaration d'un curseur consiste à le nommer et à lui associer une requête. La syntaxe est la suivante :

```
CURSOR nom_curseur [(nom_param type[,nom_param type]...)]
IS query ;
```

nom_param est un paramètre du curseur qui peut être utilisé seulement en entrée de la commande SELECT. Il doit être du type Char, Number, Date ou Boolean.
Quand un paramètre à le même nom qu'un attribut, le nom réfère l'attribut en priorité.
Pour référencer le paramètre, on prefixe avec le nom du curseur.

Exemple de déclaration

```
CURSOR cur_article (prix_unitaire Number) IS
   SELECT idarticle, prix_unitaire
   FROM article
   WHERE prix_unitaire > cur_article.prix_unitaire ;
```

Ouverture du curseur

L'ouverture du curseur se fait par la commande OPEN. Le curseur se positionne sur la première ligne de la relation qui satisfait la condition de recherche.

```
ouverture du curseur ::=
     OPEN nom_curseur [(param_entree [,param_entree]...)];
```

Les arguments spécifiés dans la déclaration sont appelés paramètres formels, les paramètres réels sont ceux définis au niveau de l'ouverture du curseur.
Chaque paramètre réel est associé à un et un seul paramètre formel.

Association par position

Chaque paramètre formel occupant la position i dans la liste des paramètres formels est associé à un paramètre réel occupant la même position dans la liste des paramètres réels.

Exemple d'ouverture:

```
DECLARE

  CURSOR nom_curseur (par_form1 Char,par_form2 Number ) IS ...
..
BEGIN
  OPEN nom_curseur ( par_reel1, par_reel2 ) ;
```

par_form1 est associé à par_reel1 et par_form2 à par_reel2.

Association par nom

L'ordre des paramètre peur être arbitraire, et la correspondance est assurée par la syntaxe
param_formel=>param_reel

Exemple d'ouverture

```
OPEN nom_curseur (par_form2=>par_reel2,par_form1=>par_reel1);
```

Défilement de curseur

Lors de son ouverture, le curseur pointe sur la première ligne, et pour trouver la ligne suivante c'est la commande FETCH qui doit être utilisée. Elle permet également de stocker les valeurs contenues dans le curseur dans des variables déclarées.

```
défilement d'un curseur ::=
     FETCH nom_curseur INTO {nom_variable [,nom_variable]...
                             | nom_enregis};
```

Fermeture du curseur.

Pour libérer les ressources du curseur une fois le traitement fini, on utilise la commande CLOSE.

```
fermeture d'un curseur ::=
          CLOSE nom_curseur ;
```

Attributs du curseurs

PL/SQL fournit des attributs pour le curseur qui sont :

%Isopen

Retourne vrai si le curseur est ouvert
Exemple :

```
IF nom_curseur%Isopen THEN
    CLOSE nom_curseur;
ELSE
    OPEN nom_curseur;
END IF;
```

%Rowcount

Compte le nombre de ligne déjà parcouru avec la commande fetch dans un curseur.

%Notfound, %Found

%Notfound génère un Vrai quand la commande FETCH échoue à cause de la non disponibilité de données.(fin du curseur)

Exemple de traitement classique:

```
OPEN c1;
    LOOP
        FETCH c1 INTO mon_record;
        EXIT WHEN c1%NotFound;
        -- traitement des données de mon_record
    END LOOP;
```

Curseur statiques implicites

La base de donnée ouvre une zone de contexte pour le traitement de chaque ordre SQL. Si la commande n'est pas une sélection, on peut accéder aux informations de la zone contexte par la syntaxe SQL%Attribut.

Exemple de curseur implicite :

```
UPDATE stock SET qty = qty - 1 WHERE stock_no = stock_id;
  IF SQL%NotFound THEN
 -- Traitement de l'anomalie!
  END IF;
```

Boucle pour curseur

Il est possible de simplifier la programmation en utilisant un cursor FOR loop, plutôt que les instructions OPEN, FETCH, et CLOSE

Cette méthode permet d'ouvrir le curseur, de défiler les données et de fermer le curseur à la fin du traitement d'une façon implicite.

```
boucle FOR curseur ::=
FOR nom_enreg IN {nom_curseur [(param [,param]...]
| (commande_select)} LOOP
Sequence_de_commandes
END LOOP ;
```

La variable nom_enreg est déclarée implicitement dans la boucle FOR. Le système assure une correspondance par nom entre les champs de la structure et ceux de la liste de sélection de la requête.

Exemple de boucle pour curseur

```
DECLARE
  salary_total  Real := 0.0;
  CURSOR c1 IS SELECT ename, sal, hiredate, deptno
                FROM emp;
        ...
 BEGIN
   FOR emp_rec IN c1 LOOP
      ...
      salary_total :=  salary_total + emp_rec.sal;

   END LOOP;
   ...
 END;
```

Le passage de paramètres dans un curseur FOR est possible :

Exemple de boucle pour curseur N°2:

```
DECLARE
    CURSOR c1 (dnum NUMBER) IS
        SELECT sal, comm FROM emp WHERE deptno = dnum;
    ...
BEGIN
    FOR emp_rec IN c1(20) LOOP

        ...
    END LOOP;
    ...
END;
```

Ouverture du curseur pour mise a jour

Il est possible de mettre a jour les données directement dans le curseur.

Il faut déclarer le curseur avec une clause FOR UPDATE pour la gestion multi - utilisateur.(exclusif row locks)

Exemple de déclaration de curseur pour mise à jour:

```
DECLARE
   CURSOR c1 IS SELECT empno, sal FROM emp
     WHERE job = 'SALESMAN' AND comm > sal
     FOR UPDATE;
```

Les verrous seront posés lors de l'exécution de la commande OPEN et relâchés sur le COMMIT marquant la fin de la transaction.

Lors de la manipulation des données dans le curseur, il faut utiliser la clause WHERE CURRENT OF pour référencer le curseur actif.

Exemple de mise à jour de données à partir d'un curseur:

```
DECLARE
    CURSOR c1 IS SELECT empno, job, sal
    FROM emp FOR UPDATE;
    ...
BEGIN
    OPEN c1;
    LOOP

        FETCH c1 INTO ...
        ...
        UPDATE emp SET sal = new_sal
          WHERE CURRENT OF c1;
    END LOOP;
    ...
```

Lorsque les données sont manipulées sur une requête multi-table, il est possible de verrouiller que certaines tables en utilisant la clause FOR UPDATE OF.

Pointeurs de curseur

Les pointeurs de curseurs ont la particularité d'être attachés une requête lors de l'exécution et non pas lors de la compilation du bloc PL/SQL. Il est donc possible de faire des curseurs dynamique, en construisant la requête SQL par programmation.

Création d'un type de curseur

Pour créer un pointeur de curseur, un type de curseur doit être déclaré avec sa structure par une clause RETURN, puis un objet de ce type peut être déclaré. La définition du query sera affectée lors de l'ouverture du curseur.

```
DECLARE
  TYPE ty_curseur1 IS REF CURSOR RETURN Varchar ;
  Mon_curseur1 ty_curseur1 ;
```

Ouverture du curseur

Lors de l'ouverture du curseur, un query lui est affecté. La chaîne contenant la requête SQL peut être construite.

```
BEGIN
  ch_SQL := 'SELECT col1 FROM table1' ;
  OPEN Mon_curseur1 FOR ch_SQL ;
```

Pointeur de curseur fortement ou faiblement typé

Lors de la déclaration du type pour le pointeur de curseur, le type de retour est optionnel. Si il est présent, le compilateur va vérifier les types si cela est possible. Ce pointeur de curseur dit à typage fort, présente l'avantage d'être contrôlé lors de la compilation.
L'absence de type de retour, si il comporte l'inconvénient de ne pas être contrôlé, permet une programmation très souple puisque la structure du curseur est totalement dynamique. Ce second type de pointeur de curseur dit à typage faible nécessite une programmation très propre car si lors du FETCH les variables hotes ne sont pas correctement typées par rapport au curseur, l'exception *ROWTYPE_MISMATCH* sera activée.

Gestion des exceptions

PL/SQL possède un mécanisme de gestion d'erreurs qui permet d'identifier et de traiter des erreurs survenant dans un programme de façon propre et séparée de la séquence du programme. L'utilisateur peut à sa convenance utiliser des exceptions déjà prédéfinies pour lui, ou définir ses propres exceptions.

Exception internes

les exceptions internes se déclenchent quand une instruction PL/SQL viole une règle du SGBD, ou dépasse une limite du système d'exploitation.

Les erreurs du SGBD sont numérotées, mais PL/SQL ne gère que des erreurs nommées, ainsi il a redéfini quelques erreurs du SGBD, et les gère comme des exceptions.

CURSOR_ALREADY_OPEN	vous essayez d'ouvrir un curseur déjà ouvert
DUP_VAL_ON_INDEX	vous essayez d'insérer une copie d'une valeur dans une colonne à valeurs uniques
INVALID_CURSOR	vous vous référez à un nom de cursor non valide ou vous essayez d'y effectuer des opérations non conformes
INVALID_NUMBER	vous essayez d'utiliser quelque chose d'autre qu'un nombre là où un est requis
LOGIN_DENIED	votre connexion a été refusée
NO_DATA_FOUND	aucune donnée n'a été sélectionnée par une requête **select into**
NOT_LOGGED_ON	vous n'êtes pas connecté à Oracle 7
PROGRAM_ERROR	une erreur interne au programme PL/SQL est survenue
STORAGE_ERROR	une erreur de place mémoire est survenue
TIMEOUT_ON_RESOURCE	le temps d'attente lors de la demande d'une ressource Oracle 7 (client/serveur) a été dépassé
TOO_MANY_ROWS	votre **select into** a retourné plus qu'une ligne
TRANSACTION_BACKED_OUT	votre transaction a été annulée par un serveur Oracle (client/serveur)
VALUE_ERROR	une erreur arithmétique ou de conversion ou de troncation ou de contrainte est survenue
ZERO_DIVIDE	vous essayez de diviser par 0

Pour gérer les exceptions, il faut définir l'action a exécuter lorsque l'exception est activée dans la partie exception.

Exemple

```
EXCEPTION
  WHEN err1 THEN
  -- Gérer l'exception de err1
  WHEN err2 THEN
  -- Gérer l'exception de err2
  WHEN err3 OR err4 THEN
  -- Gérer les exceptions err3 et err4
  WHEN OTHERS THEN
  -- Gérer toutes les autres exceptions
END ;
```

Pour les erreurs non nommées par PL/SQL, on dispose de la directive de compilation Pragma

```
PRAGMA EXCEPTION_INIT (nom_exception, numero_erreur_oracle) ;
```

Exemple :
-1031 est le numéro d'erreur interne d'Oracle, pour la violation de privilèges.

```
DECLARE
    privile_insuffisant Exception ;
    PRAGMA EXCEPTION_INIT ( privilege_insuffisant, -1031 ) ;
    ...
BEGIN
    ...
EXCEPTION
    WHEN privilege_insuffisant THEN
   -- Gere l'erreur -1031
```

Définition d'exceptions par l'utilisateur

L'utilisateur peut nommer ses propres exceptions dans la partie **DECLARE** d'un programme PL/SQL. Une exception utilisateur se définit en utilisant le type **Exception**.

Exemple

```
DECLARE
   mon_exception Exception ;
```

La commande RAISE, permet l'activation de l'exception externe.
Elle arrête l'exécution normale du bloc et transfert le contrôle au gestionnaire d'exception.

La procédure prédéfinie Raise_Application_Error stoppe l'exécution du bloc et affiche une erreur a l'utilisateur.

```
Raise_application_error(num_err, message);
```

ou num_err est un entier négatif dans les valeurs sont comprises entre -20000 .. -20999. message est une chaîne de 2048 bytes.

Lorsque Raise_application_error est appelé, le SGBD fait un ROLLBACK et retourne le numéro et le message d'erreur a l'application

Exemple :

```
DECLARE
  stock_a_zero  Exception ;

BEGIN
  IF qty_stock = 0 THEN
    RAISE stock_a_zero ;
  END IF ;

EXCEPTION
  WHEN stock_a_zero THEN
    Raise_application_error ( -20001, 'Le stock est vide') ;
END ;
```

Propagation des exceptions

Lorsqu'une exception est activée, et que PL/SQL ne peut pas trouver d'action dans les exceptions du bloc courant, elle est propagée jusqu'à ce qu'elle soit traitée.

Exemple:

```
DECLARE
    ...
BEGIN
    ...
    ---------------- Début du sous-bloc --------------
    DECLARE
        mon_exception  Exception;
    BEGIN
        ...
        IF ... THEN
            RAISE mon_exception;
        END IF;
        ...

    END;
------------------- Fin de sous-bloc -------------------
    ...
EXCEPTION
    ...
    WHEN OTHERS THEN
       ROLLBACK;
END;
```

```
DECLARE
    ...
BEGIN
    ...
    ---------------- Début du sous-bloc --------------
    DECLARE
        mon_exception  Exception;
    BEGIN
        ...
        IF ... THEN
            RAISE mon_exception;
        END IF;
        ...

    END;
------------------- Fin de sous-bloc -------------------
```

```
...
EXCEPTION
   ...
   WHEN OTHERS THEN
       ROLLBACK;     END;
```

Une seule exception peut être activée en même temps.

Gestion des transactions

Dans un bloc PL/SQL, l'utilisateur peut utiliser les commandes standard de gestion de transaction pour cordonner ses traitements. En plus des deux commandes COMMIT et ROLLBACK, l'utilisateur dispose de la commande SAVEPOINT qui lui permet de découper une transaction en sous-transactions. Ainsi l'utilisateur peut annuler seulement une partie de la transaction, contrairement au mécanisme classique qui nécessite l'annulation de la totalité de la transaction.

Syntaxe :

```
COMMIT ;
ROLLBACK [WORK] [TO SAVEPOINT] savepoint ;
SAVEPOINT savepoint ;

Exemple :
...
BEGIN
     ..
     INSERT INTO temp VALUES ...
     SAVEPOINT a ;
     ...
     INSERT INTO temp VALUES
     ...
     SAVEPOINT b ;
     INSERT INTO temp VALUES
     ...

     IF condition THEN
       ROLLBACK TO SAVEPOINT b ;
     ELSEIF condition_2
       ROLLBACK TO SAVEPOINT a ;
     ELSE
       COMMIT;
     END IF ;
     ...
```

Unités de programmation

Une unité de programmation est une unité de traitement qui peut contenir des objets PL/SQL. Elles sont appelés procédures ou fonctions. (Une fonction est une procédure qui retourne une valeur.) Elles sont crées comme des objets de la base et appartiennent a un schéma. Elle sont donc soumis au mécanisme de sécurité et de confidentialité (droit EXECUTE).
On peut donc appeler des procédures stockées sur la BD, a partir de plusieurs applications différentes.
Un package est une « encapsulation » de procédures, fonctions, curseurs et variables au sein d'une unité nommée explicitement.

Les procédures et les fonctions

La notion de procédure a été conçue dans l'esprit de grouper un ensemble de commande avec les instructions procédurales, pour constituer une unité de traitement logique analogue à une commande SQL présentée au moteur.

Les procédures et les fonctions sont utilisées pour augmenter considérablement la production. Elles servent également à gérer la sécurité et l'intégrité des données ainsi qu' à augmenter les performances.
Du point de vue de la sécurité, l'utilisateur peut autoriser l'accès à certaines tables seulement à travers les procédures. Les usagers bénéficiant du privilège d'accès aux tables à travers des procédures développées n'auront aucune autorisation d'accès à ces mêmes tables en dehors du cadre ces procédures.
Au niveau de l'intégrité, les procédures développées et testées assure la même fonctionnalité indépendamment de la partie appelante. Autrement dit, la recompilation d'une procédure en cas de correction n'exige pas la recompilation de l'ensemble du corps l'application.

Les performances sont assurées par les facteurs suivants:

- Réduction du trafic sur le réseau.(Remote Process Control)
- Compilation des procédures cataloguées

(Le moteur ne recompile pas les procédures a l'exécution).

- Exécution immédiate de la procédure si elle est dans la mémoire du SGBD (réduction des I/O du disque).
- Partage de l'exécution d'une procédure par plusieurs utilisateurs (notions des modules partagée).

Création de procédures et de fonctions

La commande qui permet de créer une procédure est la suivante:

```
    CREATE [OR REPLACE] PROCEDURE
     [schema.]nom_fonction
[(<liste_d_arguments>)]
     {IS|AS}
     bloc_PL/SQL ;
```

Pour une fonction :

```
    CREATE [OR REPLACE] FUNCTION
     [schema.]nom_fonction
[(<liste_d_arguments>)]
     RETURN type
     {IS|AS}
     bloc_PL/SQL ;

liste_d_arguments ::=
   nom_argument {IN | OUT | IN OUT} type
```

L'option OR REPLACE permet de spécifier au système le remplacement de la procédure ou de la fonction si elle existe déjà dans la base.

L'utilisateur peut précéder le nom de la procédure (fonction) par celui d'un schéma s'il n'est pas dans cet environnement, à condition d'avoir les privilèges de création de procédures dans le schéma.

Le mot clé IN indique que la variable est passée en entrée(passage par valeur).

Le mot clé OUT indique que la variable est renseignée par la procédure puis renvoyée à l'appelant.
Le mot clé IN OUT est une combinaison des deux modes précédents. La variable est passée en entrée, renseignée par la procédure puis renvoyée à l'appelant (équivalant au passage de paramètres par référence dans les langages de programmation).

Le mot RETURN permet de spécifier le type de la donnée de retour de la fonction.

Exemples:

```
CREATE PROCEDURE baisse_prix (id IN NUMBER,
                              taux IN NUMBER)
IS
BEGIN
         UPDATE article SET prixunit = prixunit * (1+taux )
           WHERE id_article=id;
EXCEPTION
         WHEN No_Data_Found THEN
           Raise_Application_Error (-20010,
                'Article Invalide:'I);
       END ;

CREATE FUNCTION choix_numero(nature IN CHAR )
   RETURN Number
IS

valeur Number;
BEGIN
  IF UPPER(nature) = 'C' THEN
     SELECT seq_cl.Nextval INTO valeur
       FROM Dual;
    ELSIF UPPER(nature) = 'A' THEN
      valeur := 0 ;
    ELSE
      valeur := -1;
  END IF;

 RETURN ( valeur);
END;
```

En cas d'erreur de compilation, l'utilisateur doit la corriger sur le fichier et la soumettre au moteur avec l'option OR REPLACE

Modification d'une procédure (fonction)

Si le schéma de la base évolue (suppression ou modification de tables), il faut recompiler les procédures existantes pour qu'elles tiennent compte de ces modifications.

La commande est la suivante:

```
ALTER {FUNCTION | PROCEDURE }
  [schéma.]nom_unite COMPILE;
```

Cette commande recompile uniquement les procédures cataloguées standard. Il faut avoir les privilèges nécessaires pour réaliser cette recompilation.

Exemple:

```
ALTER PROCEDURE baisse_prix COMPILE;
ALTER FUNCTION numéro COMPILE;
```

Suppression d'une procédure (fonction)

```
DROP {FUNCTION | PROCEDURE} [schema.]nom_unite
```

L'appel d'une fonction est une expression.

Introduction

Pour des développement importants, la maintenance de procédures stockées peut rapidement devenir fastidieuse. Il est préférable d'utiliser les concepts de compilation séparée et de librairies qui ont déjà fait leurs preuves en programmation classiques.
Dérivé de Ada, PL/SQL dispose de ces mécanismes efficaces, et met à disposition des développeurs la notion de « package »

Les packages

Les packages permettent d'encapsuler des procédures, des fonctions, des curseurs et des variables comme une unité nommée dans la base de données. Ils apportent un certain nombre d'avantages par rapport aux procédures et aux fonctions isolées.

Structuration de développement

Les packages offrent un meilleur moyen de structuration et d'organisation du processus de développement. Le mécanisme de gestion de privilèges devient plus facile par rapport aux procédures (fonctions) cataloguées. En effet, l'attribution de privilèges d'utilisation des composantes d'un package se fait par une seule commande.

Développement de composants

Le package offrent un meilleur mécanisme de gestion de la sécurité. L'utilisateur peut spécifier au cours de la création d'un package des composantes d'un package publiques et des composantes privées. La séparation des déclarations des composantes d'un package de leur corps permet une meilleure flexibilité au développeur.

Gestion des performances

Les performances peuvent être améliorées en utilisant les packages plutôt que les procédures cataloguées. Le moteur charge en mémoire le package entier quand une de ses procédures est appelée. Une fois le package en mémoire, le moteur n'a plus besoin d'effectuer de lectures (I/O disque) pour exécuter les procédures de ce même package.

Constituants d'un package

Un package est constituées de deux parties distincts :
-Les spécifications du package (PACKAGE)
-Le corps du package (PACKAGE BODY)

Spécifications du packages

Les spécifications d'un package consistent à déclarer les procédures, fonctions, constantes, variables, types et exceptions publiques.
En autres termes, il s'agit de déclaration des objets qui pourront être utilisés par des objets externes au package.

```
specification_pl/sql::=
            declaration_de_variable |
            declaration_d_enregistrement |
            declaration_de_curseur |
            declaration_d_execption |
            declaration_de_tableau_pl/sql |
            declaration_de_fonction |
            declaration_de_procédure...
```

Corps du package

Le corps d'un package implémente les procédures (fonctions), les curseurs et les exceptions qui sont déclarés dans les spécifications de la procédure.
Il peut également définir d'autres objets de même type non déclarés dans les spécifications. Ces objets sont alors privés et ne peuvent en aucun cas être accédés en dehors du corps du package. La commande qui permet de créer le corps d'une procédure est la suivante:

```
corps_pl/sql::=
            declaration_de_variable |
            declaration_d_enregistrement |
            corps_de_curseur |
            declaration_d_execption |
            declaration_de_tableau_pl/sql |
            corps_de_fonction |
            corps_de_procédure...
```

L'identificateur de la spécification et celui du corps du package doivent être les mêmes, ce qui permets au système de les associés.

Exemple de package

L'exemple est un package composé de deux fonctions qui calculent le chiffre d'affaires global et par client pour un vendeur donné et une procédure d'augmentation annuelle des salaires de vendeurs.

```
CREATE PACKAGE ges_vendeur IS
     TYPE venEnreg IS RECORD
          (idven vendeur.idvendeur%Type,
          qual vendeur.qualification%Type,
          salaire vendeur.nom%TYPE):
     TYPE LstVendeur IS TABLE OF vendeur.nom%Type
          INDEX BY BINARY_INTERGER;

     INVAL_VENDEUR Exception;

     FUNCTION chiffre_affaire (Idvendeur Number) RETURN
                                                  Number;
     FUNCTION chiffre_affaire (Idvendeur Number;Idclient
                                                  Number )
          RETURN Number;

     PROCEDURE augmante_ann;

END ges_vendeur,

CREATE PACKAGE BODY ges_vendeur IS
/*fct permettant le calcul du chiff. d'aff. d'un vendeur*/

FUNCTION chiffre_affaire(Idvendeur Number)
     RETURN Number IS
     ca Number DEFAULT 0;
BEGIN
     SELECT Nvl (Sum/a.montant),)INTO ca
          FROM COMMANDE a
          WHERE a.idvendeur = Idvendeur;
     RETURN (CA);
END chiffre_affaires;

/*Fct permettant le calcul du ch. d'aff du vendeur pour un
client donne*/

FUNTION chiffre_affaire(Idvendeur Number,Idclient Number)
     RETURN Number IS
     ca Number DEFAUT 0;
BEGIN
     SELECT Nvl(Sum(a.montant),0) INTO ca
          FROM COMMANDE a
          WHERE a.idvendeur = Idvendeur AND a.idclient =
                                                  Idclient;
     RETURN(ca);
END chiffre affaires;
```

```
/*Procédure permettant de réaliser une augmentation de 6%
pour tout les employés*/

/*Ensuite et en fonction du chiffre d'affaires réaliser
une seconde augmentation*/

PROCEDURE augmente_ann IS
     CURSOR bonus IS
          SELECT commision
               FROM vendeur
               WHERE qualification = '002'
               FOR UPDATE OF salaire,
     mntcom  vendeur.commission%Type;

BEGIN
     IF(To_Char(Sysdate,'DDMM')='0501') THEN
          UPDATE vendeur SET salaire = salaire*1.06;
          OPEN bonus
          LOOP
               FETCH bonus INTO mntcom;
               EXIT WHEN bonus%NotFound;
               IF(mntcom-500000)THEN
                    facteur:=.05
               ESLIF (mntcom-200000)THEN
                    facteur:=.03;
               ELSE facteur := 0;
               END IF;
               UPDATE vendeur
                    SET salaire = salaire*(1+facteur)
                    WHERE CURRENT OF bonus;
          END LOOP;
          CLOSE bonus;
     END IF;
END;
BEGIN
     INSERT INTO audit_vendeur VALUES Sysdate, User,
                                      'Gestion Vendeur');
     nbr_engages:=0;
END ges_vendeur;
```

Visibilités des objets

Les objets déclarés dans les spécifications du package sont rendus publics et peuvent être accédés à l'intérieur du package. Ainsi , les variables, les curseurs et les exceptions publics peuvent être utilisés par des procédures et des commandes qui n'appartiennent pas au package, à condition d'avoir le privilège EXECUTE sur ce package. Les objets privés sont déclarés par le corps du package et n'auront comme étendue que le corps du package. Les variables d'une procédure ne sont accessibles que par la procédure elle-même.

Il est important de signaler l'étendue (durée de vie) des variables, constantes et curseurs entre les procédures cataloguées et les packages. Les variables appartenant à une procédure standard ou du package ont une durée de vie limitée à l'exécution de la procédure. Une fois la procédure terminée, les variables et les constantes sont perdues.
Par contre, les mêmes objets déclarés au niveau des spécifications d 'un package ou de son corps persistent le long de la session et après le premier appel à ce package.

On va donc trouver trois sortes de variables dans un package

Les variables publiques

Elle sont déclarées dans les spécifications du package.
Elles sont visibles par tous les objets du packages, ainsi que ceux externes au package, tout en respectant la gestion des droits classiques (GRANT)

Les variables privées, globales.

Elles sont déclarée uniquement dans le corps du package. Elle sont visibles par tous les objets du package, mais uniquement par ceux-ci

Les variables locales

Déclarées dans la procédure ou la fonction, elles ne sont visibles que de celles-ci

Initialisation de variables

Quand une session commence, les variables et les curseurs sont initialisés à la valeur nulle, à moins qu'une initialisation ait été explicitement effectuée sur ces objets. Pour réaliser cette initialisation explicité, un package peut contenir dans son corps exécuté seulement lors du premier appel à ce package. Ce code constitue un bloc séparé par les mots clés BEGIN et END.

Signature

Il est possible de nommer plusieurs procédures de la même façon. Cette possibilité permet de définir une procédure à plusieurs points d'entrée (nombre d'arguments et type de données différents).

Exemple de signatures

```
CREATE OR REPLACE PACKAGE demo IS

PROCEDURE proc_1 ( param_1 IN Varchar ) ;

PROCEDURE proc_1 ( param_1 IN Varchar , param_2 Boolean ) ;

PROCEDURE proc_1 ( param_1 IN Number ) ;

END ;

CREATE OR REPLACE PACKAGE BODY demo IS

PROCEDURE proc_1 ( param_1 IN Varchar )
IS
  ……
  …….
END proc_1 ;

PROCEDURE proc_1 ( param_1 IN Varchar , param_2 Boolean ) ;
IS
  ……
  …….
END proc_1 ;

PROCEDURE proc_1 ( param_1 IN Number ) ;
IS
  ……
  …….
END proc_1 ;
END ;
```

Création d'un package

Création d'un package se fait en deux étapes ordonnées
1) Création des spécifications du package (PACKAGE)
2) Création du corps package (PACKAGE BODY)

Lors de la création du corps du package, le système va vérifier les signatures des objets décrits dans les spécifications. Ci celles ci ne correspondent pas la compilation ne peut être faite.

Syntaxe :

```
CREATE [OR REPLACE] PACKAGE [schema.]nom_package
         {IS | AS} specification_pl/sql

specification_pl/sql::=
             declaration_de_variable |
             declaration_d_enregistrement |
             declaration_de_curseur |
             declaration_d_execption |
             declaration_de_tableau_pl/sql |
             declaration_de_fonction |
             declaration_de_procédure...
```

La création du corps
d'une procédure est la suivante:

```
CREATE [OR REPLACE] PACKAGE BODY [schema.]nom_package
      {IS | AS} corps_pl/sql

corps_pl/sql::=
             declaration_de_variable |
             declaration_d_enregistrement |
             corps_de_curseur |
             declaration_d_execption |
             declaration_de_tableau_pl/sql |
             corps_de_fonction |
             corps_de_procédure...
```

L' identificateur de la spécification et du corps du package doivent être les mêmes.

Modification d'un package

La modification d'un package concerne sa version compilée. En d'autres termes, il est important de recompiler le package afin que le noyau tienne compte de l'évolution de la base et que l'on puisse modifier ainsi sa méthode d'accès et son plan d'exécution.

Cette recompilation se fait par la commande suivante:

```
ALTER PACKAGE [schema.]package
     COMPILE [ PACKAGE | BODY ];
```

Pour les modification de l'implémentation des objets, on utilisera la commande CREATE OR REPLACE, qui recréée un objet package.
Il n'est pas nécessaire de sauvegarder les sources des package dans des fichiers pour les reprendre en vue d'une modification de leur contenu, car la source est stockée dans la base de données.

Exemple: Recompiler le package ges_vendeur.

```
ALTER PACKAGE ges_vendeur COMPILE PACKAGE;
```

Suppression d'un package

La suppression d'un package se fait par la commande DROP comme suit:

```
DROP PACKAGE [schéma.]package;
```

Vues du dictionnaire

Les vues du dictionnaire référençant les procédures et les packages sont :

USER_SOURCE
USER_OBJECT_SIZE
USER_ERRORS

Dépendances d'objets

Un package qui contient des erreurs de programmation existe dans la base de données mais n'est pas compilé. Il a un status INVALID.
De même, lors de son execution, si il essaie d'atteindre un objet inexistant (supprimé), il sera décompillé par le noyau.

Il est donc important de connaître les dépendances entre objets de programmation, afin de les modifier ou de les détruire.
Une analyse d'impact simple peut être faite en traquant les dépendances directs sur l'objet analysé en interrogeant la vue du dictionnaire USER_DEPEDENCIES

Packages fournis

La base de données dispose de package dans le dictionnaire. Ces objets, appartiennent au schéma SYS, peuvent être utilisés par les développeurs.
La liste exhaustive de ces objets est différente pour chaque version de la base de données, mais on peut signaler quelques grands classiques.

DBMS_OUTPUT	Informations de sorties des procédures stockées
DBMS_DDL	Procédures et fonctions d'accès au LDD
DBMS_UTILITY	Analyse d'objets d'un schéma
DBMS_SESSION	Modification de la session utilisateur
DBMS_SHARED_POOL	Accès au SQL partagé
DBMS_TRANSACTION	Contrôle logique des transactions
DBMS_MAIL	Lien avec Oracle*Mail
DBMS_PIPE	Envoi de messages du serveur (PIPE UNIX)
DBMS_ALERT	Signalisation d'événement sur la database
DBMS_LOCK	Appel par l'application de lock

Ces packages existant dans la base de données peuvent être considérés comme des librairies enrichissants le langage PL/SQL.

Droits d'accès et modules PL/SQL stockés

Pour pouvoir exécuter un module PL/SQL stocké il faut en être le propriétaire, ou disposer du droit EXECUTE sur cet objet.
Si l'on dispose des droits d'exécution, il y a deux cas de figures définit lors de la compilation de la procédure, pour les droits de manipulation des objets utilisé par les modules PL/SQL .
Les droits ne peuvent être gérés a travers les rôles et doivent être donnée directement à l'utilisateur

Si la clause AUTHID est définie avec DEFINER, le module est exécuté avec les droits de son propriétaire. Par contre c'est avec les droits de l'utilisateur qui exécute le module si elle est définie avec CURRENT USER.

Dans le cas de l'utilisation de AUTHID DEFINER (valeur par défaut) les droits sont contrôlés lors de la compilation.

```
CREATE PROCEDURE ma_proc (param1 IN Varchar)
AUTHID DEFINER
IS
…
```