# Introduction a Oracle PL/SQL

PL/SQL (pour PROCEDURAL LANGUAGE/SQL) est un langage procédural d'Oracle corporation étendant SQL. Il permet de combiner les avantages d'un langage de programmation classique avec les possibilités de manipulation de données offertes par SQL.

**Avantages :**

- **La modularité** : Un bloc peut être nommé pour devenir une procédure ou une fonction catalogué, donc réutilisable. Une fonction ou procédure cataloguée peut être incluse dans un paquetage.
- **La portabilité** : Un programme PL/SQL est indépendant du système d'exploitation qui héberge le serveur Oracle. En changeant de système, les applicatifs n'ont pas à être modifiés.
- **L'intégration avec les données des tables** : On retrouvera avec PL/SQL tous les types de données et instructions disponibles sous SQL.

## 1 Structure d'un programme PL/SQL

La structure de base d'un programme PL/SQL est celle de bloc (possiblement imbriqué).

Il a généralement la forme suivante:

```
DECLARE
/* section de déclaration */
BEGIN
/* corps du bloc de programme
Il s'agit de la seule zone
dont la présence est obligatoire */
EXCEPTION
/* gestion des exceptions */
END;
```

Le corps du programme (entre le BEGIN et le END) contient des instructions PL/SQL (assignements, boucles, appel de procédure) ainsi que des instructions SQL.
Il s'agit de la seule partie qui soit obligatoire. Les deux autres zones, dites zone de déclaration et zone de gestion des exceptions sont facultatives.
Les seuls ordres SQL que l'on peut trouver dans un bloc PL/SQL sont: SELECT, INSERT, UPDATE, DELETE.
Les autres types d'instructions (par exemple CREATE, DROP, ALTER) ne peuvent se trouver qu'a l'extérieur d'un tel bloc. Chaque instruction se termine par un ";".

Le PL/SQL ne se soucie pas de la casse (majuscule vs. minuscule). On peut inclure des commentaires par - - (en début de chaque ligne commentée) ou par /*.... */ (pour délimiter des blocs de commentaires).

## 2 Variables et types

Un nom de variable en PL/SQL comporte au plus 30 caractères. Toutes les variables ont un type On trouve trois sortes de types de variables en PL/SQL.

A savoir :

• un des types utilisés en SQL pour les colonnes de tables.
• un type particulier au PL/SQL.
• un type faisant référence à celui d'une (suite de) colonne(s) d'une table.

Les types autorisés dans PL/SQL sont nombreux.

On retiendra principalement:
- **Pour les types numériques** : REAL, INTEGER, NUMBER (précision de 38 chiffres par défaut), NUMBER(x) (nombres avec x chiffres de précision), NUMBER(x,y) (nombres avec x chiffres de précision dont y après la virgule).
- **Pour les types alphanumériques** : CHAR(x) (chaîne de caractère de longueur fixe x), VARCHAR(x) (chaîne de caractère de longueur variable jusqu'à x), VARCHAR2 (idem que précédent excepté que ce type supporte de plus longues chaînes et que l'on est pas obligé de spécifier sa longueur maximale).
- **Pour les types Dates :** PL/SQL permet aussi de manipuler des dates (type DATE) sous différents formats.

Une déclaration pourrait donc contenir les informations suivantes :

***DECLARE***
***n NUMBER;***
***mot VARCHAR(20)***
***mot2 VARCHAR2***

**Affectation :**

Il existe plusieurs possibilités pour affecter une valeur à une variable :

- L'affectation variable := expression
- Par la directive DEFAULT
- Par la directive INTO d'une requête (SELECT .... INTO variable FROM....)

Exemple :

```
DECLARE
v_brevet VARCHAR2(6);
v_brevet2 VARCHAR2(6);
v_prime NUMBER(5,2);
v_naissance DATE;
v_trouvé BOOLEAN NOT NULL DEFAULT FALSE;

BEGIN
v_brevet := 'PL-1';
v_brevet2 := v_brevet;
v_prime := 500.50;
v_naissance := '04-07-1971';
v_trouvé := TRUE;
SELECT brevet INTO v_brevet FROM pilote WHERE nom = 'Tondeur Hervé';
END;
```

Une autre spécificité du PL/SQL est qu'il permet d'assigner comme type à une variable celui d'un champs d'une table (par l'opérateur **%TYPE**) ou d'une ligne entière (opérateur **%ROWTYPE**).

**%TYPE :** déclare une variable selon la définition d'une colonne d'une table ou d'une vue existante. Elle permet de déclarer une variable conformément à une autre variable précédemment déclarée.

**%ROWTYPE :** Permet de travailler au niveau d'un enregistrement. Ce dernier est composé d'un

ensemble de colonnes. L'enregistrement peut contenir toutes les colonnes d'une table ou seulement certaines.

Dans la déclaration suivante :

***DECLARE***
***nom emp.nom%TYPE;***
***employe emp%ROWTYPE;***

La variable nom est définie comme étant du type de la colonne nom de la table emp (qui doit exister au préalable).
De même, employé est un vecteur du type d'une ligne de la table emp.
A supposer que cette dernière ait trois champs numéro, nom, age de type respectifs NUMBER, VARCHAR, INTEGER, la variable employe disposera de trois composantes : employe.numero, employe.nom, employe.age, de même types que ceux de la table.

Un premier petit programme (noter au passage l'instruction d'affectation "a:=a+b") :

***DECLARE***
***a NUMBER;***
***b NUMBER;***
***BEGIN***
***a:=a+b;***
***END;***

Un deuxième petit programme (incluant une requête SQL) :

***DECLARE***
***a emp.numero%TYPE;***
***BEGIN***
***SELECT numero INTO a FROM emp WHERE noemp='21';***
***END;***

Ce dernier exemple donne comme valeur à la variable a le résultat de la requête (qui doit être du même type). Il est impératif que la requête ne renvoie qu'un et un seul résultat (c'est à dire qu'il n'y ait qu'un seul employé numéro 21). Autrement, une erreur se produit.

**Les variables RECORD :**

Ce sont des structures de données personnalisées. Nb : il peuvent contenir des LOB ou des extensions objets (REF, TABLE ou VARRAY).

**TYPE** NomRecord **IS RECORD**
**(**NomChamp TypeDonées [NOT NULL] {:= | DEFAULT} expression],
**.....);**

Exemple :

```
DECLARE
TYPE AvionAirBus_rec IS RECORD
( nserie CHAR(10), nomAvion CHAR(20),
  usine CHAR(10) := 'Blagnac',
  nbHVol NUMBER(7,2));
```

```
r_UnA320  avionAirBus_rec;

BEGIN
r_UnA320.nserie := 'A1';
r_UnA320.nomAvion := 'A320-200';
r_UnA320.nbHVol := 2500.60;
END;
```

**Les variables tableaux :**

Permettent de définir et manipuler des types tableaux dynamiques, défini sans dimension initiale.
Un type tableau est composé d'une clé primaire (de type BINARY_INTEGER) et d'une colonne (de type scalaire, TYPE, ROWTYPE ou RECORD) pour stocker chaque élément.

**TYPE** nomTypeTableau **IS TABLE OF**
{typeScalaire | variable%TYPE | table.colonne%TYPE} [NOT NULL] | table%ROWTYPE
[INDEX BY BINARY_INTEGER];

Exemple :

```
DECLARE
TYPE brevet_typetab IS TABLE OF VARCHAR(6) INDEX BY BINARY_INTEGER;
TYPE nompilote_typetab IS TABLE OF Pilote.nom%TYPE INDEX BY BINARY_INTEGER;
TYPE pilotes_typetab IS TABLE OF Pilote%ROWTYPE INDEX BY BINARY_INTEGER;

tab_brevets brevet_typetab;
tab_nompilote nompilote_typetab;
tab_pilotes pilotes_typetab;

BEGIN
tab_brevets(1) := 'PL-1';
tab_brevets(2) := 'PL-2';
tab_nompilote(7800) := 'Hervé';
tab_pilotes(0).brevet := 'PL-0';
END;
```

Quelques fonction pour les tableaux :

**EXISTS(x)** Retourne TRUE si le Xieme élément du tableau existe.
**COUNT** Retourne le nombre d'éléments du tableau.
**FIRST/LAST** Retourne le premier/dernier indice du tableau(NULL si vide).
**PRIOR(x)/NEXT(x)** Retourne l'élément avant/après le X ieme élément du tableau.
**DELETE/DELETE(x)/DELETE(x,y)** Supprime un ou plusieurs éléments du tableau.

**Les variables de substitution :**

Il est possible de passer en paramètres d'entrée d'un bloc PL/SQL des variables définies sous SQL*PLUS. On accède aux valeurs d'une telle variable dans le code PL/SQL en faisant préfixer le nom de la variable par un « & ».
La directive ACCEPT permet de faire une lecture de la variable au clavier.

Exemple :

```
ACCEPT s_brevet PROMPT 'Entrer code Brevet : '
ACCEPT s_duréeVol PROMPT 'Entrer durée du Vol : '

DECLARE
v_nom Pilote.nom%TYPE;
v_nbHVol Pilote.nbHVol%TYPE;
BEGIN
SELECT nom, nbHVol INTO v_nom, v_nbHVol FROM Pilote WHERE brevet = '&s_brevet';
v_nbHVol := v_nbHVol + &s_duréeVol;
DBMSOUTPUT.PUT_LINE('Total heures vol : ' || v_nbHVol || ' de ' || v_nom);
END;
```

Les variables de session :

Il est possible de définir des variables de session globales définies sous SQL*PLUS au niveau d'un bloc PL/SQL. La directive à utiliser est VARIABLE, dans le code PL/SQL il faut faire préfixer le nom de la variable par « : ». L'affichage de la variable sous SQL*PLUS est réalisé par la directive PRINT.

Exemple :

```
VARIABLE g_compteur NUMBER;

DECLARE
v_compteur NUMBER(3) := 99;
BEGIN
:g_compteur := v_compteur + 1;
END;

PRINT g_compteur;
```

## 3 Opérateurs

PL/SQL supporte les opérateurs suivants :

• Arithmétique : +, - *, /, ** (exponentielle)
• Concaténation : ||
• Parenthèses (contrôle des priorités entre opérations): ()
• Affectation: :=
• Comparaison : =, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN
• Logique : AND, OR, NOT

## 4 Structures de contrôle

Comme n'importe quel langage procédural, PL/SQL possède un certain nombre de structures de contrôles évoluées comme les branchements conditionnels et les boucles.

### 4.1 Les branchements conditionnels

Syntaxe de la structure IF :

```
IF <condition> THEN
commandes;
```

**[ ELSEIF <condition> THEN**
**commandes; ]**
**[ ELSE**
**commandes; ]**
**END IF;**

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

Syntaxe de la structure CASE :

[<<étiquette>>]
**CASE** <variable>
**WHEN** <expr1> **THEN** instructions1;
**WHEN** <expr2> **THEN** instructions2;
....
[**ELSE** instructionsN+1;]
**END CASE** [étiquette]**;**

Exemple :

**CASE**
**WHEN v_note>=16 THEN v_mention := 'TB';**
**WHEN v_note>=10 THEN v_mention := 'P';**
**ELSE v_mention := 'R';**
**END CASE**

## 4.2 boucles

PL/SQL admet trois sortes de boucles. La première est une boucle potentiellement infinie :

**LOOP**
**commandes;**
**END LOOP;**

Au moins une des instructions du corps de la boucle doit être une instruction de sortie :

**EXIT WHEN <condition>;**

Dès que la condition devient vraie (si elle le devient...), on sort de la boucle.

Le deuxième type de boucle permet de répéter un nombre défini de fois un même traitement :

**FOR <compteur> IN [REVERSE] <limite_inf> .. <limite_sup>**
**commandes;**
**END LOOP;**

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

```
WHILE <condition> LOOP
commandes;
END LOOP;
```

Toutes ces structures de contrôles sont évidemment « imbriquables » les unes dans les autres. Voici un même exemple traité de trois manières différentes suivant le type de boucle choisi.

```
DECLARE
x NUMBER(3):=1;
BEGIN
LOOP
INSERT INTO employe(noemp, nomemp, job, nodept)
VALUES (x, 'TOTO', 'PROGRAMMEUR', 1);
x:=x+1;
EXIT WHEN x>=100
END LOOP;
END;
```

Deuxième exemple :

```
DECLARE
x NUMBER(3);
BEGIN
FOR x IN 1..100
INSERT INTO employe(noemp, nomemp, job, nodept)
VALUES (x, 'TOTO', 'PROGRAMMEUR', 1);
END LOOP;
END;
```

Troisième exemple :

```
DECLARE
x NUMBER(3):=1;
BEGIN
WHILE x<=100 LOOP
INSERT INTO employe(noemp, nomemp, job, nodept)
VALUES (x, 'TOTO', 'PROGRAMMEUR', 1);
x:=x+1;
END LOOP;
END;
```

## 5 Curseurs

Jusqu'à présent, nous avons vu comment récupérer le résultat de requêtes SQL dans des variables PL/SQL lorsque ce résultat ne comporte au plus qu'une seule ligne.
Bien évidemment, cette situation est loin de représenter le cas général : les requêtes renvoient très souvent un nombre important et non prévisible de lignes. Dans la plupart des langages procéduraux au dessus de SQL, on introduit une notion de "curseur" pour récupérer (et exploiter) les résultats de requêtes.
Un curseur est une variable dynamique qui prend pour valeur le résultat d'une requête. La méthode pour utiliser un curseur est invariable. En premier lieu, celui-ci doit être défini (dans la zone de déclaration).

Exemple :

**CURSOR empCur IS SELECT * FROM emp;**

Le curseur de nom empCur est chargé dans cet exemple de récupérer le résultat de la requête qui suit. Il peut alors être ouvert lorsqu'on souhaite l'utiliser (dans le corps d'un bloc) :

**OPEN empCur;**

Pour récupérer les tuples successifs de la requête, on utilise l'instruction :

**FETCH empCur INTO employeVar;**

La première fois, c'est le premier tuple du résultat de la requête qui est affecté comme valeur à la variable employeVar (qui devra être définie de façon adéquate).
A chaque exécution de l'instruction, un nouveau tuple résultat est chargé dans employeVar.
Enfin, lorsque le traitement sur le résultat de la requête est terminé, on ferme le curseur.

**CLOSE empCur;**

Le petit exemple suivant sélectionne l'ensemble des employés dont le salaire ne dépasse pas 6000 francs et les augmente de 500 francs.

```
DECLARE
CURSOR SalCur IS
SELECT * FROM EMP WHERE SAL<6000.00;
employe EMP%ROWTYPE;
BEGIN
OPEN SalCur;
LOOP
FETCH SalCur INTO employe;
UPDATE EMP
SET SAL=6500.00 WHERE EMPNO=employe.empno;
EXIT WHEN SalCur%NOTFOUND;
END LOOP;
END;
```

Lorsqu'on souhaite parcourir un curseur dans une boucle pour effectuer un traitement, on peut simplifier l'utilisation de ceux-ci.

Le programme suivant, parfaitement légal, est équivalent au précédent.

```
DECLARE
CURSOR SalCur IS
SELECT * FROM EMP WHERE SAL<6000.00;
employe EMP%ROWTYPE;
BEGIN
FOR employe IN SalCur
LOOP
UPDATE EMP
SET SAL=6500.00 WHERE EMPNO=employe.empno;
END LOOP;
END;
```

Dans cette boucle, l'ouverture, la fermeture et l'incrémentation du curseur SalCur sont implicites et n'ont pas besoin d'être spécifiées.
Un certain nombre d'informations sur l'état d'un curseur sont exploitables à l'aide d'attributs prédéfinis : l'attribut **%FOUND** renvoie vrai si le dernier FETCH a bien ramené un tuple. L'attribut **%NOTFOUND**, dual du précédent, permet de décider si on est arrivé en fin de curseur. **%ROWCOUNT** renvoie le nombre de lignes ramenés du curseur au moment de l'interrogation. Enfin **%ISOPEN** permet de déterminer si le curseur est ouvert.

**NB : PL/SQL utilise un curseur implicite** pour chaque opération LMD de SQL (insert, update, delete). Ce curseur porte le nom SQL et il est exploitable après avoir éxécuté l'instruction.

SQL%ROWCOUNT, SQL%FOUND et SQL%NOTFOUND peuvent dans ce cas être utilisés.

## 6 La gestion des exceptions

PL/SQL permet de définir dans une zone particulière (de gestion d'exception), l'attitude que le programme doit avoir lorsque certaines erreurs définies ou prédéfinies se produisent.
Un certain nombre d'exceptions sont prédéfinies sous Oracle. Citons, pour les plus fréquentes : NO DATA FOUND (devient vrai dès qu'une requête renvoie un résultat vide), TOO MANY ROWS (requête renvoie plus de lignes qu'escompté), CURSOR ALREADY OPEN (curseur déjà ouvert), INVALID CURSOR (curseur invalide)...

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.

Exemple :

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 ;
```

Syntaxe du bloc de gestion des exceptions :

**EXCEPTION**
**WHEN <exception1> [OR <exception2> OR ...] THEN <instructions>**
**WHEN <exception3> [OR <exception2> OR ...] THEN <instructions>**
**WHEN OTHERS THEN <instructions>**
**END;**

Évidemment, un seul traitement d'exception peut se produire avant la sortie du bloc.

**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

## 7 Procédures et fonctions

Il est possible de créer des procédures et des fonctions comme dans n'importe quel langage de programmation classique. La syntaxe de création d'une procédure est la suivante :

**CREATE [OR REPLACE] PROCEDURE <schéma.nomProcedure>**
**[ (parametre [IN | OUT | IN OUT] [NOCOPY] typeSQL [ {:= | DEFAULT} expression ],**
**……..**
**)]**
**{IS | AS}**
**<zone de déclaration de variables>**
**BEGIN**
**<corps de la procédure>**
**EXCEPTION**
**<traitement des exceptions>**
**END;**

A noter que toute fonction ou procédure crée devient un objet part entière de la base (comme une table ou une vue, par exemple). Elle est souvent appelée "procédure stockée". Elle est donc, entre autres, sensible à la notion de droit : son créateur peut décider par exemple d'en permettre l'utilisation à d'autres utilisateurs.
Elle est aussi appelable depuis n'importe quel bloc PL/SQL (sur le même principe, on peut créer des bibliothèques de fonctions et de procédures appelées "packages").

Il y a trois façon de passer les paramètres dans une procédure : IN (lecture seule), OUT (écriture seule), INOUT (lecture et écriture).
Le mode IN est réservé au paramètres qui ne doivent pas être modifiés par la procédure.
Le mode OUT, pour les paramètres transmis en résultat, le mode INOUT pour les variables dont la valeur peut être modifiée en sortie et consultée par la procédure (penser aux différents passages de paramètres dans les langages de programmation classiques...).

Regardons l'exemple suivant :

```
CREATE TABLE T (num1 INTEGER, num2 INTEGER)
CREATE OR REPLACE PROCEDURE essai(IN x NUMBER, OUT y NUMBER, INOUT z NUMBER)
AS
BEGIN
y:=x*z;
z:=z*z;
INSERT INTO T VALUES(x,z);
END;
```

Cette fonction est appelée dans le bloc suivant :

```
DECLARE
a NUMBER;
b NUMBER;
BEGIN
a:=5;
b:=10;
essai(2,a,b);
a:=a*b;
```

Après l'appel, le couple (a, b) vaut (20, 100) et c'est aussi le tuple qui est inséré dans T. Puis, a prend la valeur 20 100 = 2000.

Lors de l'appel de la procédure, les arguments correspondants aux paramètres passés en mode OUT ou INOUT ne peuvent être des constantes (puisqu'ils doivent être modifiables).
Cette distinction des paramètres en fonction de l'usage qui va en être fait dans la procédure est très pratique et facilite grandement le débogage.
On peut aussi créer des fonctions. Le principe est le même, l'en-tète devient :

```
CREATE [OR REPLACE] FUNCTION <schéma.nomfonction>
[ (parametre [IN | OUT | IN OUT] [NOCOPY] typeSQL [ {:= | DEFAULT} expression ],
   ..............
  )]
RETURN typeSQL
{IS | AS}
<zone de declaration de variables>
BEGIN
<corps de la procedure>
EXCEPTION
<traitement des exceptions>
END;
```

Une instruction RETURN <expression> devra se trouver dans le corps pour spécifier quel résultat est renvoyé.

Les procédures et fonctions PL/SQL supportent assez bien la surcharge (i.e. Coexistence de procédures de même nom ayant des listes de paramètres différentes).
C'est le système qui, au moment de l'appel, inférera, en fonction du nombre d'arguments et de leur types, quelle est la bonne procédure à appeler.

On peut détruire une procédure ou une fonction par (les procédures et fonctions sont des objets stockés. On peut dont les détruire par des instructions similaires aux instructions de destructions de tables...)

**DROP PROCEDURE <nom de procedure>**

## 8 Les déclencheurs (“triggers”)

Les déclencheurs ou ”triggers” sont des séquences d’actions définis par le programmeur qui se déclenchent, non pas sur un appel, mais directement quand un événement particulier (spécifié lors de la définition du trigger) sur une ou plusieurs tables se produit.

Un trigger sera un objet stocké (comme une table ou une procédure)

La syntaxe est la suivante:

**CREATE [OR REPLACE] TRIGGER <nom>**
**{BEFORE|AFTER} {INSERT|DELETE|UPDATE} ON <nom de table>**
**[FOR EACH ROW [WHEN (<condition>)]]**
**<corps du trigger>**

Un trigger se déclenche avant ou après (BEFORE|AFTER) une insertion, destruction ou mise à jour (INSERT|DELETE|UPDATE ) sur une table (à noter que l’on peut exprimer des conditions plus complexes avec le OR: INSERT OR DELETE ...).

L’option FOR EACH ROW [WHEN (<condition>)]] fait s’exécuter le trigger à chaque modification d’une ligne de la table spécifiée (on dit que le trigger est de ”niveau ligne”). En l’absence de cette option, le trigger est exécuté une seule fois (”niveau table”).

Soit l’exemple suivant :

```
CREATE TABLE T1 (num1 INTEGER, num2 INTEGER);
CREATE TABLE T2 (num3 INTEGER, num4 INTEGER);
CREATE TRIGGER inverse
AFTER INSERT ON T1
FOR EACH ROW WHEN (NEW.num1 <=3)
BEGIN
INSERT INTO T2 VALUES(:NEW.num2, :NEW.num1);
END inverse;
```

Ce trigger va, en cas d’insertion d’un tuple dans T1 dont la première coordonnée est inférieure à 3, insérer le tuple inverse dans T2. Les préfixes NEW et OLD (en cas de UPDATE ou de DELETE) vont permettre de faire référence aux valeurs des colonnes après et avant les modifications dans la table. Ils sont utilisés sous la forme NEW.num1 dans la condition du trigger et sous la forme :NEW.num1 dans le corps.

## Les déclencheurs sur événements 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

### Les attributs

**ora_client_ip_adress**
Adresse IP du poste client qui se connecte
**ora_database_name**
Nom de la base de données
**ora_des_encrypted_password**
Description codée du mot de passe de l'utilisateur créé ou modifé
**ora_dict_obj_name**
Nom de l'objet visé par l'opération DDL
**ora_dict_obj_name_list**
Liste de tous les noms d'objets modifiés
**ora_dict_obj_owner**
Propriétaire de l'objet visé par l'opération DDL
**ora_dict_obj_owner_list**
Liste de tous les propriétaires d'objets modifiés
**ora_dict_obj_type**
Type de l'objet visé par l'opération DDL
**ora_grantee**
Liste des utilisateurs disposant du privilège
**ora_instance_num**
Numéro de l'instance
**ora_is_alter_column**
Vrai si la colonne en paramètre a été modifiée
**ora_is_creating_nested_table**
Création ou non d'une table de fusion
**ora_is_drop_column**
Modification ou non de la colonne en paramètre
**ora_is_servererror**
Vrai si le numéro erreur passé en paramètre se trouve dans la pile des erreurs
**ora_login_user**
Nom de la connexion
**ora_privileges**
Liste des privilèges accordés ou retirés par un utilisateur
**ora_revokee**
Liste des utilisateurs à qui le privilège a été retiré
**ora_server_error**
Numéro d'erreur dans la pile dont la position est passée en paramètre
**ora_sysevent**
Nom de l'évènement système qui a activé le déclencheur
**ora_with_grant_option**
Vrai si le privilège a été accordé avec option d'administration

### Les évènements système

**CREATE TRIGGER nom_déclencheur {BEFORE|AFTER} évènement_système ON{DATABASE|SCHEMA} bloc PL/SQL**

**STARTUP**
Evènement déclenché lors de l'ouverture de l'instance (AFTER seulement)
**SHUTDOWN**
Evènement déclenché avant le processus d'arrêt de l'instance (non déclenché en cas d'arrêt brutal du serveur) (BEFORE seulement)
**SERVERERROR**
Evènement déclenché lors d'une erreur Oracle (sauf ORA-1034, ORA-1403, ORA-1422, ORA-1423 et ORA-4030) (AFTER seulement)

**Les évènements utilisateur**
**CREATE TRIGGER nom_déclencheur {BEFORE|AFTER} évènement_utilisateur ON{DATABASE|SCHEMA} bloc PL/SQL**

**LOGON**
Après une connexion (AFTER seulement)
**LOGOFF**
Avant une déconnexion (BEFORE seulement)
**CREATE**
Lors de la création d'un objet
**ALTER**
Lors de la modification d'un objet
**DROP**
Lors de la suppression d'un objet
**ANALYZE**
Lors de l'analise d'un objet
**ASSOCIATE STATISTICS**
Lors de l'association d'une statistique
**AUDIT**
Lors de la mise en place d'un audit
**NOAUDIT**
Lors de l'annulation d'un audit
**COMMENT**
Lors de l'insertion d'un commentaire
**DDL**
Lors de l'exécution des ordres DDL (sauf ALTER DATABASE, CREATE CONTROLFILE et CREATE DATABASE)
**DISSOCIATE STATISTICS**
Lors de la dissociation d'une statistique
**GRANT**
Lors de l'exécution d'une commande GRANT
**RENAME**
Lors de l'exécution d'une commande RENAME
**REVOKE**
Lors de l'exécution d'une commande REVOKE
**TRUNCATE**
Lors d'une troncature de table

Un trigger peut être activé ou désactivé par les instructions :

**ALTER TRIGGER <nom> {ENABLE|DISABLE};**

et détruit par :

**DROP TRIGGER <nom>.**

**Maintenance des déclencheurs**
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 événementiel ou non

La colonne TRIGGERING_EVENT permet de connaître l'événement concerné par le déclencheur

La colonne TRIGGER_BODY contient le code du bloc PL/SQL .

## 9 Quelques remarques

### 9.1 Affichage
PL/SQL n'est pas un langage avec des fonctionnalités d'entrées sorties évoluées (ce n'est pas son but!). Toutefois, on peut imprimer des messages et des valeurs de variables de plusieurs manières différentes. Le plus pratique est de faire appel à un package prédéfini : DBMS output.

En premier lieu, taper au niveau de la ligne de commande :

**SET SERVEROUTPUT ON**

Pour afficher, on utilise alors la commande suivante :

**DBMS_OUTPUT.PUT_LINE('Au revoir' || nom || ' a bientôt');**
Si on suppose que nom est une variable dont la valeur est "toto", l'instruction affichera :
Au revoir toto a bientôt.

### 9.2 Débugger et gestion des exceptions
Pour chaque objet crée (procédure, fonction, trigger...), en cas d'erreur de compilation, on peut Voir ces erreurs en tapant :

**SHOW ERRORS PROCEDURE nomprocedure;**