

# PL/SQL

## I. Préambule

SGBD Relationnel

**PL/SQL : Procedural Language** (Oracle)

Remarque : Outils pour faire du PL/SQL

LiveSQL ; OEE (oracle express Edition (v10) c'est un exe) ; téléchargement sur oracle ; SQLDeveloper.

Définition : Le PL/SQL est une extension du SQL (instructions SELECT, INSERT, DELETE, UPDATE) auquel on vient adosser du code

Code : structure répétitive (boucles), structure alternative (if....), récursivité...

SQL niveau 1 : on évolue dans un environnement CLIENT / SERVEUR => tous les échanges impliquent une réponse en retour donc des échanges entre le C et le S.

SQL niveau 2 : on évolue dans un environnement CLIENT / SERVEUR avec un BLOC qui ne donne lieu qu'à UN SEUL ECHANGE

Bloc : c'est une suite d'instructions contenues dans un script dans lequel on retrouve du SQL et du code.

## SQL 1 niveau 1

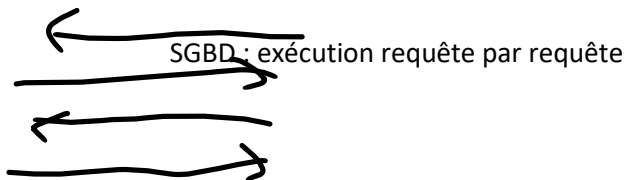
CLIENT

SERVEUR

Select

Update

Insert



**SQL 2 niveau 2** : un seul échange entre le client et le serveur (avec les résultats intermédiaires qui sont calculés côté Serveur et seul le résultat final est renvoyé au Client)

CLIENT

SERVEUR

Declare

Begin

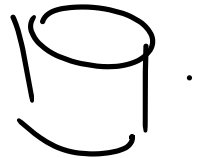
Select Update Insert

Exception

End;



SGDB: Execution GLOBALE



Structure du bloc (script, programme) :

Le bloc est organisé en **sections** : DECLARE, BEGIN, EXCEPTION, END

DECLARE : on déclare les variables, les types, les curseurs.....

BEGIN : section obligatoire qui contient toutes les directives SQL, le code PL/SQL ... On peut faire appel à d'autres blocs depuis ce bloc

EXCEPTION permet de traiter les erreurs retournées par le SGBD et de les faire remonter

END ;

Remarques :

- pas de casse comme en SQL (A....Z, a....z, 0.....9), les symboles, les commentaires avec – ou /\*\*/. Un programme en PL/SQL interprète tous les caractères.

## II. Variables

On peut en PL/SQL manipuler des variables qui sont déclarées => on peut transmettre des valeurs à des programmes et réaliser l'affichage en sortie de résultats.

Il existe différents types de variables :

### -Variable de base :

*Identification d'un nom de la variable typée donnée ;*

*/\*déclaration de la variable test\*/*

test NUMBER (3) ;

*/\*déclaration de la variable test et initialisation à 999\*/*

test NUMBER (3) := 999 ;

*/\*déclaration de la variable x\*/*

x NUMBER (8) ;

*/\*déclaration de la variable x et initialisation à 125\*/*

x NUMBER (3) := 125 ;

Remarque : on peut mettre un v devant la variable pour « expliquer » que c'est une variable par convention

v\_datenaissance DATE ;

v\_trouv BOOLEAN ;

### -Variable %TYPE :

La directive %TYPE permet de déclarer une variable selon la définition **d'une colonne** d'une table (ou d'une VUE existante)

Cf exemple : on se trouve dans la partie DECLARE. On a une BD avec **une table PILOTE** qui a pour **propriété BREVET**

V\_brevet **Pilote.brevet**%TYPE; /\*V\_brevet prend le type de la colonne **brevet** de la table **Pilote**\*/ /\*on utilise un V majuscule ou V minuscule => PAS de Casse\*/

### -Variable %ROWTYPE :

On travaille sur un enregistrement (record), la directive **ROWTYPE** permet de définir un **ensemble de colonnes ou toutes les colonnes d'une table**.

Ce type de variable est très utilisé notamment pour la gestion et l'administration de la base de données

Cf exemple : on se trouve dans la partie DECLARE. On a une BD avec **une table PILOTE** qui a différentes colonnes

Exple1 :

Rty\_pilote **Pilote.%ROWTYPE**; cette variable est liée à tout ou partie des colonnes de la table pilote

Exple2 :

Pour un select :

SELECT \*

INTO Rty\_pilote

FROM Pilote

WHERE brevet = 'PL\_1';

Pour une affectation

Rty\_pilote.brevet := XXXXXXX;

Rty\_pilote.nom := YYYYYY;

Pour un insert

INSERT INTO Pilote /\*on travaille dans la table pilote\*/

VALUES rty\_pilote

### -Variable de substitution :

Avec une variable de substitution on va passer en paramètre d'entrée de bloc **avant le BEGIN** => utilisation du ACCEPT.

**ACCEPT S\_brevet PROMPT 'blablalblalbla' ; /\*S** en rapport avec la variable de **S**ubstitution par convention. Nous n'avons pas à les DECLARER \*/

On va accéder aux valeurs d'une variable dans le code, dans le Begin (on va faire préfixer cette variable du symbole **&**)

Exemple : on utilise les variables avant le DECLARE qui n'ont pas à être déclarées.

Interaction avec l'utilisateur.

Remarque : utilisation d'un package d'affichage via DBMS\_OUTPUT.PUT\_LINE

### -Variable de session :

Une variable de session est une variable globale que l'on retrouve avant le DECLARE.

Il faut faire préfixer le nom de la variable de session du symbole **:** dans le code

On utilise cette variable de session avec les mêmes options que les variables de bases dans le BEGIN

### -Variable tableaux :

Il s'agit du type TABLE => on va manipuler des tableaux DYNAMIQUES (taille sans dimension initiale).

Le tableau est composé d'une clé primaire et d'une colonne (qui a un type de base, %TYPE, %ROWTYPE....) qui stocke chaque élément.

### Déclaration d'un tableau :

*TYPE nomdutableau IS TABLE OF variable de base, %TYPE, %ROWTYPE /\*type de variable du contenues dans le tableau\*/*

*INDEX BY BINARY INTEGER*

Exemple : on va définir le tableau puis on déclare des variables liées à ce tableau dans la partie DECLARE

**Tab\_brevets** **brevet\_tytat** /\*on déclare un tableau **Tab\_brevets** lié à **brevet\_tytat**\*/

Remarque : il existe des fonctions prédéfinies qui effectuent des actions COUNT, EXIST ...

Remarque : on peut utiliser des indices pour affecter des valeurs à ce tableau. Par exemple (0, -1, -2, 7800) => l'accès à ces éléments se fait via un pointeur.

## -Affectation de variables

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

- Variable := affectation ou expression
- DEFAULT exple : v\_trouv BOOLEAN DEFAULT FALSE ;
- Utilisation de la directive **INTO** dans le cadre d'une requête :

SELECT affichage

**INTO** variable

FROM

WHERE

;

Exple :

DECLARE /\*declaration de variable\*/

v\_brevet VARCHAR(6);

BEGIN

SELECT brevet

**INTO** v\_brevet /\*affectation d'une chaine de caractère par une requete  
depuis la propriété brevet dans la table colonne\*/

FROM pilote

WHERE nom = 'toto';

End;

### III. Interaction avec la Base de Données

Il s'agit des mécanismes pour interfacer un script PL/SQL avec la BD

- **Extraire des données**

On utilise le SELECT :

SELECT

INTO

FROM

WHERE

- **Manipuler des données**

Il s'agit : INSERT UPDATE DELETE

- **Paquetage DBMS\_OUTPUT**

Ce paquetage permet d'afficher des résultats.

DBMS\_OUTPUT.PUT\_LINE

Remarque : c'est utile aussi pour afficher des résultats intermédiaires

Exple : affichage de résultats

DBMS\_OUTPUT.PUT\_LINE ('CHAINE DE CARACTERE' || variable || 'AUTRE CHAINE DE CARACTERE' );

## IV. Séquences

Définition : On parle d'objet pour la séquence (objet au sens oracle) (objet au sens système et objet utilisateur).

On effectue une action à un moment donné : l'objectif est de générer une suite liée à des entiers (lien avec PK)

La séquence peut être liée à toutes les actions en SQL.

Code :

*Créer* => CREATE :

```
CREATE SEQUENCE nomsequence
            INCREMENT BY (incrémentation de la séquence=> entier)
            START WITH (entier)
            MAXVALUE
            MINVALUE
```

*Modifier* :

```
ALTER SEQUENCE nomsequence
```

*Supprimer* :

```
DROP SEQUENCE nomsequence
```

*Fonction* :

On associe la séquence à une fonction : **sequence.fonction**

**CURRVAL** : retour de la valeur courante de la séquence

**NEXTVAL** : incrémentation de la séquence

Exemple :

```
CREATE SEQUENCE nomsequence /*nom de la séquence*/
            INCREMENT BY 1 /*indentation non nécessaire*/
```

```
INSERT INTO client /*création d'une ligne supplémentaire via l'insert dans la table
client*/
```

```
VALUES (nomsequence.NEXTVAL,'durant','P12',SYSDATE)
```

```
SELECT nomsequence.CURRVAL /*cela nous retourne un résultat : 1*/
```

```
FROM client
```



## V. SouS-Programmes

Il existe deux types de sous-programmes :

Procédure : pour effectuer une action (seule la procédure peut avoir des paramètres en sortie). (cf document ci-dessous)

Fonction : renvoyer un résultat. (cf document ci-dessous)

Remarque : les sous-programmes sont compilés et on retrouve leur code (ces programmes peuvent être utilisés, partagés dans un cadre multi-utilisateur) dans le dictionnaire de données.

le noyau recompile (lors de l'appel de la fonction ou de la procédure) le programme si un objet cité dans le code a été modifié et le charge en mémoire

En termes d'utilisation : il existe des avantages à utiliser des sous-programmes en terme de sécurité (gestion des objets avec les droits d'accès sur les programmes stockés), d'intégrité (les traitements dépendants sont exécutés dans la même transaction), de performance (simplicité en terme d'utilisation et de maintenance) et de productivité (limitation du nombre d'appels à la base)

Transaction : c'est une suite de code qui permet de faire passer la base de données d'un état cohérent à un autre état cohérent lui aussi. (Atomicité de l'instruction contenue dans la transaction)

Appel : la personne qui a les droits pour gérer les sous-programmes peut les exécuter (privilège pour lancer : EXECUTE). Le mode de fonctionnement est identique que ce soit une procédure ou une fonction.

Supprimer : DROP (DROP FUNCTION nomssprog ; DROP PROCEDURE nomssprog)

### Exemple :

#### Création de la Procédure :

```
CREATE [OR REPLACE] PROCEDURE [schéma.]nomProcédure
    [(paramètre [ IN | OUT | IN OUT ] [NOCOPY] typeSQL
      [{:= | DEFAULT} expression]
    [,paramètre [ IN | OUT | IN OUT ] [NOCOPY] typeSQL
      [{:= | DEFAULT} expression]... ) ] ]
{ IS | AS } corpsduSousProgrammePL/SQL ;
```

Création de la Fonction :

## tions cataloguées

```
CREATE [OR REPLACE ] FUNCTION [schéma.]nomFonction
    [(paramètre [ IN | OUT | IN OUT ] [NOCOPY] typeSQL
        [{:= | DEFAULT} expression]
    [,paramètre [ IN | OUT | IN OUT ] [NOCOPY] typeSQL
        [{:= | DEFAULT} expression]... ) ] ]
RETURN typeSQL
{IS | AS} corpsduSousProgrammePL/SQL ;
```

- il doit se trouver une instruction RETURN dans le code.

## Gestion des paramètres IN, OUT, IN OUT

- `IN` désigne un paramètre d'entrée, `OUT` un paramètre de sortie et `IN OUT` un paramètre d'entrée et de sortie. Il est possible d'initialiser chaque paramètre par une valeur.
- `NOCOPY` permet de transmettre directement le paramètre. On l'utilise pour améliorer les performances lors du passage de volumineux paramètres de sortie comme les `RECORD`, les tables `INDEX-BY` (les paramètres `IN` sont toujours passés en `NOCOPY`).
- *corpsduSousProgrammePL/SQL* contient la déclaration et les instructions de la procédure, toutes deux écrites en PL/SQL.

## Appel de Procédure et de Fonction :

Procédure	Fonction
<pre> SET SERVEROUT ON DECLARE   v_comp      VARCHAR2(4) := 'AF';   v_nom       VARCHAR2(16);   v_heuresVol NUMBER(7,2); BEGIN   PlusExpérimenté(v_comp,                   v_nom, v_heuresVol);   DBMS_OUTPUT.PUT_LINE     ('Nom, heures de vol '    v_nom         ' : '    v_heuresVol); END; / </pre>	<pre> SET SERVEROUT ON DECLARE   v_comp      VARCHAR2(4) := 'AF';   v_heuresVol NUMBER(7,2) := 300;   v_résultat  NUMBER; BEGIN   v_résultat :=     EffectifsHeure(v_comp, v_heuresVol);   DBMS_OUTPUT.PUT_LINE('Pour AF et     300h résultat : '    v_résultat ); END; / </pre>

Nom, heures de vol Gilles Laborde : 2450	Pour AF et 300h résultat : 2
Procédure PL/SQL terminée avec succès.	Procédure PL/SQL terminée avec succès.