

Cours 5 : PL/SQL

<http://deptinfo.unice.fr/~grin/messupports/plsql.pdf>

http://sheikyerbouti.developpez.com/pl_sql/

Pourquoi PL/SQL ?

- PL/SQL = PROCEDURAL LANGUAGE/SQL
- SQL est un langage non procédural
- Les traitements complexes sont parfois difficiles à écrire si on ne peut utiliser des variables et les structures de programmation comme les boucles et les alternatives
- On ressent vite le besoin d'un langage procédural pour lier plusieurs requêtes SQL avec des variables et dans les structures de programmation habituelles

Principales caractéristiques

- Extension de SQL : des requêtes SQL cohabitent avec les structures de contrôle habituelles de la programmation structurée (blocs, alternatives, boucles)
- La syntaxe ressemble au langage Ada ou Pascal
- Un programme est constitué de procédures et de fonctions
- Des variables permettent l'échange d'information entre les requêtes SQL et le reste du programme

Utilisation de PL/SQL

- PL/SQL peut être utilisé pour l'écriture des procédures stockées et des triggers
 - Oracle accepte aussi le langage Java
- Il convient aussi pour écrire des fonctions utilisateurs qui peuvent être utilisées dans les requêtes SQL (en plus des fonctions prédéfinies)
- Il est aussi utilisé dans des outils Oracle
 - Ex : Forms et Report

Normalisation du langage

- Langage spécifique à Oracle
- Tous les SGBD ont un langage procédural
 - TransacSQL pour SQLServer,
 - PL/pgSQL pour Postgresql
 - Procédures stockées pour MySQL depuis 5.0
- Tous les langages L4G des différents SGBDs se ressemblent
- Ressemble au langage normalisé PSM (Persistant Stored Modules)

Utilisation de PL/SQL

Le PL/SQL peut être utilisé sous 3 formes

- un **bloc** de code, exécuté comme une unique commande SQL, via un interpréteur standard (SQLplus ou iSQL*PLus)
- un **fichier** de commande PL/SQL
- un **programme** stocké (procédure, fonction, package ou trigger)

Le langage PL/SQL

Blocs

- Un programme est structuré en blocs d'instructions de 3 types
 - procédures ou bloc anonymes
 - procédures nommées
 - fonctions nommées
- Un bloc peut contenir d'autres blocs
- Dans cette partie, nous allons considérer les bloc anonymes.

Structure d'un bloc anonyme

Seuls BEGIN et END sont obligatoires

```
DECLARE
    -- définition des variables
BEGIN
    -- code du programme
EXCEPTION
    -- code de gestion des
erreurs
END;
```

Comme les instruction SQL, les bloc se terminent par un ;

Déclaration, initialisation des variables

- Identificateurs Oracle :
 - 30 caractères au plus
 - commence par une lettre
 - peut contenir lettres, chiffres, _, \$ et #
 - pas sensible à la casse
- Portée habituelle des langages à blocs
- Doivent être déclarées avant d'être utilisées

Déclaration, initialisation des variables

- Déclaration et initialisation

```
Nom_variable type_variable := valeur;
```

- Initialisation

```
Nom_variable := valeur;
```

- Déclaration multiple interdite

- Exemples :

```
age integer;
```

```
nom varchar(30);
```

```
dateNaissance date;
```

```
ok boolean := true;
```

Initialisation de variables

Plusieurs façons de donner une valeur à une variable

- **Opérateur d'affectation**

n :=

- **Directive INTO** de la requête SELECT

- Exemples :

- `dateNaissance := to_date('10/10/2004','DD/MM/YYYY');`
- `SELECT nom INTO v_nom
FROM emp
WHERE matr = 509;`

Pour éviter les conflits de nommage, préfixer les variables PL/SQL par v_

SELECT ... INTO ...

- **SELECT** expr1, expr2, ... **INTO** var1, var2, ...
- Met des valeurs de la BD dans une ou plusieurs variables var1, var2, ...
- Le select ne doit retourner **qu'une seule ligne**
- Avec Oracle il n'est pas possible d'inclure un select sans « into » dans une procédure
- Pour retourner plusieurs lignes, voir la suite du cours sur les curseurs.

Le type de variables

- **VARCHAR2**

- Longueur maximale : 32767 octets

- **Syntaxe:**

```
Nom_variable VARCHAR2 (30);
```

- **Exemple:**

```
name          VARCHAR2 (30);
```

```
name          VARCHAR2 (30) := 'toto';
```

- **NUMBER(long,dec)**

- Long : longueur maximale

- Dec : longueur de la partie décimale

- **Exemple:**

```
num_telnumber(10);
```

```
toto number(5,2)=142.12;
```

Le type de variables (2)

- DATE

- Par défaut DD-MON-YY (18-DEC-02)

- Fonction TO_DATE

- Exemple:

```
start_date := to_date('29-SEP-2003','DD-MON-  
YYYY');
```

```
start_date := to_date('29-SEP-2003:13:01','DD-MON-  
YYYY:HH24:MI');
```

- BOOLEAN

- TRUE
- FALSE
- NULL

Déclaration %TYPE et %ROWTYPE

- On peut déclarer qu'une variable est du même type qu'une colonne d'une table ou (ou qu'une autre variable)

- Exemple :

```
v_nom emp.nom.%TYPE;
```

- Une variable peut contenir toutes les colonnes d'une ligne d'une table

- Exemple :

```
v_employe emp%ROWTYPE;
```

déclare que la variable `v_employe` contiendra une ligne de la table `emp`

Exemple d'utilisation

```
DECLARE
    v_employe emp%ROWTYPE;
    v_nom emp.nom.%TYPE;

BEGIN
    SELECT * INTO v_employe
    FROM emp
    WHERE matr = 900;
    v_nom := v_employe.nom;
    v_employe.dept := 20;
    ...
    INSERT into emp VALUES v_employe;

END;
```

Commentaires

```
-- Pour une fin de ligne
```

```
/* Pour plusieurs  
   lignes */
```

Les principales commandes

Test conditionnel

- IF-THEN

- **IF** v_date > '11-APR-03' **THEN**
 v_salaire := v_salaire * 1.15;
END IF;

- IF-THEN-ELSE

- **IF** v_date > '11-APR-03' **THEN**
 v_salaire := v_salaire * 1.15;
ELSE
 v_salaire := v_salaire * 1.05;
END IF;

Test conditionnel

- IF-THEN-ELSIF

- **IF** v_nom = 'PAKER' **THEN**
 v_salaire := v_salaire * 1.15;
ELSIF v_nom = 'ASTROFF' **THEN**
 v_salaire := v_salaire * 1.05;
END IF;

Test conditionnel

- CASE

- **CASE** sélecteur

```
WHEN expression1 THEN résultat1
```

```
WHEN expression2 THEN résultat2
```

```
ELSE résultat3
```

```
END;
```

- Exemple

- `val := CASE city`

```
    WHEN 'TORONTO' THEN 'RAPTORS'
```

```
    WHEN 'LOS ANGELES' THEN 'LAKERS'
```

```
    ELSE 'NO TEAM'
```

```
END;
```

Le CASE renvoie une valeur qui vaut
résultat1 ou résultat2 ou
Ce n'est pas une instruction.

Les boucles

- **LOOP**

instructions exécutables;

EXIT [**WHEN** condition];

instructions exécutables;

END LOOP;

- Obligation d'utiliser la commande **EXIT** pour éviter une boucle infinie, facultativement quand une `condition` est vraie.

- **WHILE** condition **LOOP**

instructions exécutables;

END LOOP;

- -- tant que la condition est vraie

Les boucles

- **FOR** variable **IN** debut..fin

LOOP

instructions;

END LOOP;

- La variable de boucle prend successivement les valeurs de début, debut+1, debut+2, ..., jusqu'à la valeur fin.
- On pourra également utiliser un curseur dans la clause **IN** (voir plus loin)

Affichage

- Activer le retour écran sous sqlplus
 - `set serveroutput on size 10000`
- Affichage
 - `dbms_output.put_line(chaine);`
 - Utilise `||` pour faire une concaténation

Exemple

```
set serveroutput on          -- sous SQLPLUS
DECLARE
    i number(2);
BEGIN
    FOR i IN 1..5 LOOP
        dbms_output.put_line('Nombre : ' || i);
    END LOOP;
END;
```

Exemple

```
DECLARE
```

```
    nb integer;
```

```
BEGIN
```

```
    delete from emp where matr in (600, 610);
```

```
    nb := sql%rowcount;           -- curseur sql
```

```
    dbms_output.put_line('nb = ' || nb);
```

```
END;
```

Exemple

```
DECLARE
    compteur number(3);
    i number(3);
BEGIN
    select count(*) into compteur from clients;
    FOR i IN 1..compteur LOOP
        dbms_output.put_line('Nombre : ' || i );
    END LOOP;
END;
```

Les curseurs

Les curseurs

- **Toutes** les requêtes SQL sont associées à un curseur
- Ce curseur représente la zone mémoire utilisée pour *parser (analyser)* et exécuter la requête
- Le **curseur** peut être **implicite** (pas déclaré par l'utilisateur) ou explicite
- Les **curseurs explicites** servent à retourner plusieurs lignes avec un select

Attributs des curseurs

Tous les curseurs ont des attributs que l'utilisateur peut utiliser

- **%ROWCOUNT** : nombre de lignes traitées par le curseur
- **%FOUND** : vrai si au moins une ligne a été traitée par la requête ou le dernier fetch
- **%NOTFOUND** : vrai si aucune ligne n'a été traitée par la requête ou le dernier fetch
- **%ISOPEN** : vrai si le curseur est ouvert (utile seulement pour les curseurs explicites)

Les curseurs implicites

- Les curseurs implicites sont tous nommés SQL
- **Exemple :**

```
DECLARE
```

```
    nb_lignes integer;
```

```
BEGIN
```

```
    delete from emp where dept = 10;
```

```
    nb_lignes := SQL%ROWCOUNT;
```

```
    ...
```

Les curseurs explicites

- Pour traiter les select qui renvoient **plusieurs lignes**
- Ils doivent **être déclarés**
- Le code doit les utiliser explicitement avec les ordres OPEN, FETCH et CLOSE
 - OPEN moncurseur : ouvre le curseur.
 - FETCH moncurseur : avance le curseur à la ligne suivante.
 - OPEN moncurseur : referme le curseur.
- Le plus souvent on les utilise dans une boucle dont on sort quand l'attribut NOTFOUND du curseur est vrai
- On les utilise aussi dans une **boucle FOR** qui permet une utilisation implicite des instructions OPEN, FETCH et CLOSE

Curseur explicite

```
BEGIN
```

```
  open salaires;
```

```
  loop
```

```
    fetch salaires into salaire;
```

```
    exit when salaires%notfound;
```

```
    if salaire is not null then
```

```
      total := total + salaire;
```

```
      DBMS_OUTPUT.put_line(total);
```

```
    end if;
```

```
  end loop;
```

```
  close salaires;
```

Ne pas oublier de fermer le curseur

```
DBMS_OUTPUT.put_line(total);
```

```
END;
```

Type Row associé à un curseur

On peut déclarer un type « row » associé à un curseur

```
DECLARE
```

```
    cursor c is
```

```
        select matr, nome, sal from emp;
```

```
        employe c%ROWTYPE;
```

```
BEGIN
```

```
    open c;
```

```
    fetch c into employe;
```

```
    if employe.sal is null then ...
```

```
END;
```

Boucle FOR pour un curseur

- Elle simplifie la programmation car elle évite d'utiliser explicitement les instruction OPEN, FETCH, CLOSE
- En plus elle déclare implicitement une variable de type ROW associée au curseur

Boucle FOR pour un curseur

```
DECLARE
    nom varchar2(30);
CURSOR c_nom_clients IS
        SELECT nom, adresse FROM clients;

BEGIN
    FOR le_client IN c_nom_clients
    LOOP
        dbms_output.put_line('Employé : ' ||
            UPPER(le_client.nom) || ' Ville : ' ||
            le_client.adresse);
    END LOOP;
END;
```

Préfixer le nom d'un curseur
par c_ pour éviter les
confusions de nommage

Curseurs paramétrés

- Un curseur paramétré peut servir plusieurs fois avec des valeurs des paramètres différentes
- On doit fermer le curseur entre chaque utilisation de paramètres différents (sauf si on utilise « for » qui ferme automatiquement le curseur)

Curseurs paramétrés

```
DECLARE
```

```
  CURSOR c(p_dept integer) is
```

```
  select dept, nome from emp where dept =  
  p_dept;
```

Déclaration du
paramètre du curseur

```
BEGIN
```

```
  FOR employe in c(10) LOOP
```

```
    dbms_output.put_line(employe.nome);
```

```
  END LOOP;
```

```
  FOR employe in c(20) LOOP
```

```
    dbms_output.put_line(employe.nome);
```

```
  END LOOP;
```

```
END;
```

Instantiation du paramètre

Les exceptions

Les Exceptions

- Une exception est une erreur qui survient durant une exécution
- 2 types d'exception :
 - prédéfinie par Oracle
 - définie par le programmeur
- Saisir une exception
 - Une exception ne provoque pas nécessairement l'arrêt du programme si elle est saisie par un bloc (dans la partie « EXCEPTION »)
 - Une exception non saisie remonte dans la procédure appelante (où elle peut être saisie)

Exceptions prédéfinies

- NO_DATA_FOUND
 - Quand Select into ne retourne aucune ligne
- TOO_MANY_ROWS
 - Quand Select into retourne plusieurs lignes
- VALUE_ERROR
 - Erreur numérique
- ZERO_DIVIDE
 - Division par zéro
- OTHERS
 - Toutes erreurs non interceptées

Traitement des exceptions

```
BEGIN
```

```
...
```

```
EXCEPTION
```

```
    WHEN NO_DATA_FOUND THEN
```

```
        . . .
```

```
    WHEN TOO_MANY_ROWS THEN
```

```
        . . .
```

```
    WHEN OTHERS THEN
```

```
        -- optionnel
```

```
        . . .
```

```
END;
```

Exceptions utilisateur

- Elles doivent être déclarées avec le type **EXCEPTION**
- On les lève avec l'instruction **RAISE**

Exemple d'exception utilisateur

```
DECLARE
```

```
    salaire numeric(8,2);
```

```
    salaire_trop_bas EXCEPTION;
```

```
BEGIN
```

```
    select sal into salaire from emp where matr = 50;
```

```
    if salaire < 300 then
```

```
        RAISE salaire_trop_bas;
```

```
    end if;
```

```
EXCEPTION
```

```
    WHEN salaire_trop_bas THEN
```

```
        dbms_output.put_line('Salaire trop bas');
```

```
    WHEN OTHERS THEN
```

```
        dbms_output.put_line(SQLERRM);
```

```
END;
```

Procédures et fonctions

Bloc anonyme ou nommé

- Un bloc anonyme PL/SQL est un bloc « DECLARE – BEGIN – END » comme dans les exemples précédents
- Dans SQL*PLUS on peut exécuter directement un bloc PL/SQL anonyme en tapant sa définition
- Le plus souvent, on crée plutôt une procédure ou une fonction nommée pour réutiliser le code

Procédures sans paramètre

```
create or replace procedure list_nom_clients  
IS  
BEGIN  
    DECLARE  
        nom varchar2(30);  
        CURSOR c_nom_clients IS  
            select nom, adresse from clients;  
    BEGIN  
        FOR le_client IN c_nom_clients LOOP  
            dbms_output.put_line('Employé : '  
                || UPPER(le_client.nom)  
                || ' Ville : '  
                || le_client.adresse);  
        END LOOP;  
    END;  
END;
```

Procédures avec paramètre

```
create or replace procedure list_nom_clients
```

```
  (ville      IN      varchar2,
```

```
  result     OUT     number)
```

```
IS
```

```
BEGIN
```

```
  DECLARE
```

```
    CURSOR c_nb_clients IS
```

```
      select count(*) from clients where  
adresse=ville;
```

```
  BEGIN
```

```
    open c_nb_clients;
```

```
    fetch c_nb_clients INTO result;
```

```
    close c_nb_clients;
```

```
  END;
```

```
END;
```

IN : en lecture seule

OUT : en écriture seule

IN OUT : en lecture/écriture

Récupération des résultats dans SQLPLUS

- Déclarer une variable

```
SQL> variable nb number;
```

Une variable globale
s'utilise avec le préfixe :

- Exécuter la fonction

```
SQL> execute list_nom_clients('paris', :nb)
```

- Visualisation du résultat

```
SQL> print
```

- Description des paramètres

```
SQL> desc nom_procedure
```

Fonctions sans paramètre

```
create or replace function nombre_clients
```

```
return number
```

```
IS
```

```
BEGIN
```

```
    DECLARE
```

```
        i number;
```

```
        CURSOR get_nb_clients IS select count(*) from  
clients;
```

```
BEGIN
```

```
    open get_nb_clients;
```

```
    fetch get_nb_clients INTO i;
```

```
    return i;
```

```
END;
```

```
END;
```

Déclaration du type de
retour de la fonction

Exécution:
select nombre_clients() from dual

Fonctions avec paramètre

Seuls les paramètre IN (en lecture seule)
sont autorisés pour les fonctions

```
create or replace
  function euro_to_fr(somme IN number)
return number
IS
  taux constant number := 6.55957;
BEGIN
  return somme * taux;
END;
```

Procédures et fonctions

- Suppression de procédures ou fonctions
 - `DROP PROCEDURE nom_procedure`
 - `DROP FUNCTION nom_fonction`
- Table système contenant les procédures et fonctions :
`user_source`

Compilation, exécution et utilisation

- Compilation

- Sous SQL*PLUS, il faut taper une dernière ligne contenant « / » pour compiler une procédure ou une fonction

- Exécution

- Sous SQL*PLUS on exécute une procédure PL/SQL avec la commande EXECUTE :
- EXECUTE *nomProcédure(param1, ...)*;

- Utilisation

- Les procédures et fonctions peuvent être utilisées dans d'autres procédures ou fonctions ou dans des blocs PL/SQL anonymes
- Les fonctions peuvent aussi être utilisées dans les requêtes SQL

Triggers (rappels)

Les déclencheurs (trigger)

- Automatiser des actions lors de certains événements du type :
AFTER ou BEFORE et
INSERT, DELETE ou UPDATE

- Syntaxe :

```
CREATE OR REPLACE TRIGGER nom_trigger  
Événement [OF liste colonne] ON nom_table  
WHEN (condition) [FOR EACH ROW]  
Instructions PL/SQL ou SQL
```

Accès aux valeurs modifiées

- Utilisation de **new** et **old**
- Si nous ajoutons un client dont le nom est toto alors nous récupérons ce nom grâce à la variable **:new.nom**
- Dans le cas de suppression ou modification, les anciennes valeurs sont dans la variable **:old.nom**

Exemple

- Archiver le nom de l'utilisateur, la date et l'action effectuée (toutes les informations) dans une table LOG_CLIENTS lors de l'ajout d'un clients dans la table CLIENTS
- Créer la table LOG_CLIENTS avec la même structure que CLIENTS
- Ajouter 3 colonnes USERNAME, DATEMODIF, TPEMODIF

Exemple

create or replace trigger logadd
after insert on clients
for each row

Bloc PL/SQL

```
BEGIN

  insert into log_clients values
    (:new.nom, :new.adresse, :new.reference, :new
     .nom_piece,
     :new.quantite, :new.prix, :new.echeance,
     USER, SYSDATE, 'INSERT' );

END;
```