MEMENTO PL/SQL

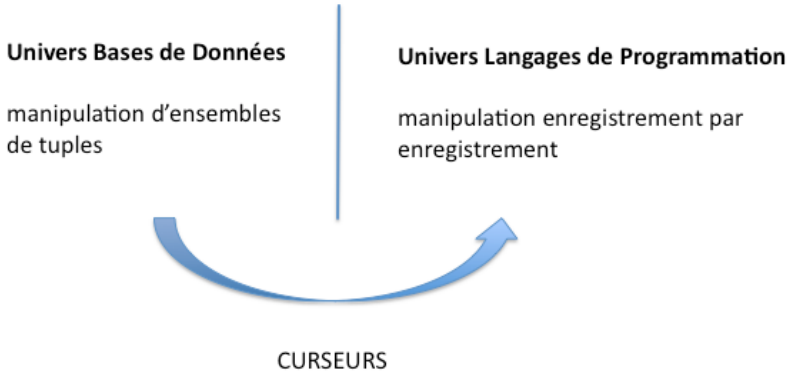
Licence L3

1. Introduction au Langage PL/SQL

- Langage procédural, inspiré de PASCAL, proposé par ORACLE pour la définition de traitements sur les données d'une BD. Ces traitements peuvent se faire sous deux formes distinctes :
  - des procédures correspondant à des blocs PL/SQL ;
  - des triggers qui sont des traitements spécifiques associés aux relations.

- Un programme PL/SQL peut inclure certains ordres SQL comme n'importe quel langage hôte. Ces ordres sont, en fait, limités à ceux de l'aspect LMD de SQL (i.e. requêtes d'interrogation et de mise à jour des données, gestion de transactions).

- L'idée, lorsqu'on écrit un programme PL/SQL, est d'utiliser au maximum les possibilités des langages relationnels (ici SQL) et de les compléter, lorsqu'elles sont insuffisantes, par celles des langages de programmation classiques.
  Le problème majeur est qu'un tel programme travaille dans deux "univers" très différents, comme l'illustre la figure suivante :



Pour assurer les échanges entre les deux mondes, il est nécessaire de disposer d'un outil permettant de "récupérer" des données de la base de manière ensembliste (en fait via une requête) et de travailler sur ces données de manière procédurale. Cet outil est un curseur.

2. Structure d'un bloc PL/SQL

Un traitement PL/SQL est décrit sous forme d'un bloc (unité d'interprétation du moteur PL/SQL), avec la syntaxe suivante :

[**DECLARE**
    Liste déclarations de variables, constantes, curseurs, exceptions]
[**BEGIN**]

```
        Liste des instructions
    [EXCEPTION
        Gestion des exceptions]
    [END] ;
```

Remarques :

- toute instruction de n'importe quelle section se termine par **;**
- La partie déclarative n'est obligatoire que s'il y a des variables, constantes, curseurs, exceptions utilisés de manière locale au bloc.
- **BEGIN** et **END** ne sont obligatoires que s'il y a une partie déclarative.
- Les commentaires sont encadrés par /* et */ ou -- et -- (pour 1 seule ligne).
- Il est possible d'imbriquer des blocs PL/SQL.

Exemple :

```
DECLARE
 var_salaire NUMBER(6);
 var_emp_id NUMBER(6) = 8207;
BEGIN
 SELECT salaire
 INTO var_salaire
 FROM employe
 WHERE emp_id = var_emp_id;
 dbms_output.put_line(var_salaire);
 dbms_output.put_line('L'employe '
        || var_emp_id || ' a une salaire de   ' || var_salaire);
END;
/
```

Le programme ci-dessus récupère le salaire de l'employé 8207 et l'affiche sur l'écran. Le caractère « / » à la fin de la déclaration indique d'exécuter le bloc PL/SQL.

3. Partie déclarative d'un bloc PL/SQL

Vous ne pouvez pas en PL/SQL définir de types structurés. Vous disposez simplement des types de base prédéfinis et d'un constructeur : tuple !

Les variables

Variables locales : elles sont définies dans la partie déclarative du bloc sous la forme : *nom_variable type* ;
où type peut être :

- un des types prédéfinis d'Oracle, i.e. ceux que vous utilisez pour définir les attributs (**VARCHAR2(n)**, **NUMBER(x,y)**, ...) ;
- le type **BOOLEAN** ;

Exemples :

```
DECLARE
salaire  NUMBER (6);
/*"Salaire" est une variable de type NUMBER et de longueur 6*/
```

```
DECLARE
salaire NUMBER(4);
dept VARCHAR2(10) NOT NULL := “INFORMATIQUE”;
/*“Dept” est une variable de type VARCHAR2 qui n’est pas NULL et qui est initialisé à
INFORMATIQUEr 6*/
```

Mais il est également possible de réutiliser le type d'un attribut ou d'une variable ou de faire référence à la combinaison des types d'un tuple (i.e. la structure d'une relation) par :

- *nom_variable* nomrelation.nom_attribut**%TYPE**;
- *nom_variable1* nom_variable2**%TYPE**;
- *nom_var_tuple* nomrelation**%ROWTYPE**;

L'obligation d'avoir une valeur pour une variable se fait en interdisant les valeurs nulles, avec **NOT NULL**, dans ce cas, la variable doit être initialisée lors de sa déclaration.

Exemples :

- nb_et **NUMBER(4,0)** ;
- nb **NUMBER** := 0 ;
- effectif nb_et**%TYPE** ;
- numero pilote.num_pil**%TYPE** ;
- un_pilote pilote**%ROWTYPE** ;

Les autres variables : elles sont toutes préfixées par :

- Variables globales à l'application : Elles permettent le passage de données entre programmes PL/SQL. Elles ne sont pas déclarées mais simplement utilisées avec la syntaxe suivante :
  - **:global.nom_variable**
- Champs d'un masque SQLFORMS (pas vu cette année): ces variables sont utilisées dans les triggers, avec la syntaxe : **:nom_bloc.nom_variable**

Les constantes et exceptions

Elles sont déclarées de la manière suivante :

nom_constante **CONSTANT** type **:=** valeur ;

ou nom_exception **EXCEPTION** ;

Exemple :

effectif_max **CONSTANT NUMBER**(2,0) **:=**40 ;

Exemple :

Cet exemple illustre qu’une constante ne peut pas être modifiée

```
DECLARE
l_string  VARCHAR2(20);
l_number  NUMBER(10);
l_con_string  CONSTANT VARCHAR2(20) := Ceci est une constante.';
BEGIN
l_string := 'Variable';
l_number := 1;
l_con_string := ‘va échouer’;
END;
```

```
/

l_con_string := Va échouer;
*
ERROR at line 10:
ORA-06550: line 10, column 3:
PLS-00363: expression 'L_CON_STRING' cannot be used as an
assignment target
ORA-06550: line 10, column 3:
PL/SQL: Statement ignored
SQL>
```

4. Instructions PL/SQL

Affectations

Il existe deux possibilités pour réaliser des affectations en PL/SQL :

- Classique : l'affectation est symbolisée par **:=**
- Par récupération de données de la BD extraites par une requête d'interrogation SQL : l'affectation est spécifiée dans le bloc SQL par une clause **INTO** intercalée entre le **SELECT** et le **FROM**.

Exemples :

- Récupération du nombre de vols stockés dans la base :

```
DECLARE
        nb_vol    NUMBER(4,0) ;
        ...
BEGIN
        SELECT COUNT(*) INTO nb_vol FROM vol ;
...
END ;
```

- Récupération d'un tuple de la relation VOL de la base :

```
DECLARE
        un_vol    vol%ROWTYPE ;
...
BEGIN
        SELECT * INTO un_vol FROM vol
        WHERE num_vol='AF523';
...
END ;
```

Avec cette deuxième possibilité d'affectation, la requête SQL ne doit retourner qu'une unique valeur ou qu'un unique tuple. Si ce n'est pas le cas, il faut utiliser un curseur.

Instruction conditionnelle

En PL/SQL, les instructions conditionnelles ont une syntaxe classique, comparable à celle d'ADA :

IF <condition> **THEN [BEGIN]** <instructions> **[END]**
[ELSIF <condition> **THEN [BEGIN]** <instructions> **[END]]**
[ELSE [BEGIN] <instructions> **[END]]**
END IF ;

La condition peut utiliser les opérateurs classiques (<, >, =,...) ainsi que **IS NULL** et **IS NOT NULL**. Les conditions peuvent être liées par **AND** et **OR**.

Exemples :

- Ce bloc augmente le salaire de l'employé 120 d'un bonus en fonction du nombre de ventes effectuées. La base de données est mise à jour.

```
DECLARE
  ventes  NUMBER(8,2) := 12100;
  quota  NUMBER(8,2) := 10000;
  bonus  NUMBER(6,2);
  emp_id NUMBER(6) := 120;
BEGIN
  IF ventes > (quota + 200) THEN
    bonus := (ventes - quota)/4;
  ELSE
    bonus := 50;
  END IF;
  UPDATE employes SET salaire = salaire + bonus WHERE employe_id = emp_id;
END;
/
```

- Ce bloc augmente le salaire de l'employé 120 en fonction de sa catégorie (jobid)

```
DECLARE
   jobid      employes.job_id%TYPE;
   empid      employes.employe_id%TYPE := 120;
   sal_augmentation  NUMBER(3,2);
BEGIN
  SELECT job_id INTO jobid from employes WHERE employe_id = empid;
  IF jobid = 'PROFESSEUR' THEN sal_augmentation := .09;
  ELSIF jobid = 'MAITRE CONFERENCE' THEN sal_augmentation := .08;
  ELSIF jobid = 'ATER' THEN sal_augmentation := .07;
  ELSE sal_augmentation := 0;
  END IF;
END;
/
```

Itérations

Là aussi, les instructions d'itération sont tout à fait classiques :

- Boucle FOR :

  FOR <compteur> **IN** <borne_inf> .. <borne_sup>
  LOOP<liste_instructions>**END LOOP ;**
  /* il est inutile de déclarer <compteur> */
- Boucle WHILE :

  WHILE <condition> **LOOP** <liste_instructions> **END LOOP** ;

- La dernière possibilité d'itération est de définir une boucle pour laquelle vous spécifiez vous même la sortie par une instruction **EXIT** !!!
  Cette possibilité, contraire à tout ce qu'on vous a enseigné en programmation, peut s'avérer extrêmement utile lors de la gestion de curseur :

  LOOP <liste_instructions> **END LOOP ;**

Exemples :

```
DECLARE
        NUM     NUMBER(2) := 0
BEGIN
        FOR num IN 0..10
        LOOP
                DBMS_OUTPUT.put_line(TO_CHAR(num)) ;
        END LOOP ;
END ;
/
```

```
DECLARE
        NUM     NUMBER(2) := 0
BEGIN
        LOOP
                INSERT INTO resultat values(NUM)
                NUM := NUM+1 ;
                EXIT WHEN NUM > 10 ;
        END LOOP ;
END ;
/
```

```
DECLARE
        RESTE   NUMBER := 7324 ;
BEGIN
        WHILE RESTE >=9 LOOP
                RESTE := RESTE-9 ;
        END LOOP ;
        INSERT INTO resultat values(reste,'reste division 7324 par 9') ;
END ;
/
```

Branchements

- **GOTO** <étiquette> ;
  où <étiquette> est spécifiée dans le bloc sous la forme : **<<** étiquette **>>**
- **EXIT WHEN**<condition> ;
  pour sortir d'une boucle !!!

Exemple :

```
DECLARE
  p      VARCHAR2(30);
  n      PLS_INTEGER := 37;
BEGIN
  FOR j in 2..ROUND(SQRT(n)) LOOP
    IF n MOD j = 0 THEN -- test nombre premier
      p := ' n'est pas un nombre premier'; -- pas un nombre premier
      GOTO print_now;
    END IF;
  END LOOP;
  p := ' est un nombre premier;
<<print_now>>
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);
END;
/
```

5. Gestion des exceptions

Il existe deux types d'exception :

- Exceptions définies par l'utilisateur dans la partie déclarative du bloc. Elles sont déclenchées dans le corps du bloc, si une condition est remplie, par :
  IF <condition>**THEN RAISE**<nom_exception>**; END IF ;**
- Exceptions prédéfinies, gérées par ORACLE, correspondant à des erreurs internes.

Quelques exemples :

- **NO_DATA_FOUND** : déclenchée si une requête ne rend aucun résultat ;
- **ZERO_DIVIDE** : déclenchée s'il y a tentative de division par 0 ;
- **DUP_VAL_ON_INDEX** : déclenchée lors d'une tentative d'insertion d'une valeur dupliquée pour un attribut sur lequel est défini un index primaire ;
- **INVALID_NUMBER** : déclenchée si une incompatibilité pour un type numérique est détectée.
- **INVALID_CURSOR** déclenchée par exemple dans le cas d'accès à un curseur non ouvert.

Le traitement des exceptions se fait dans la partie **EXCEPTION** du bloc PL/SQL par :

WHEN <nom_exception> **THEN** [**BEGIN**] <liste_instructions> [**END**] **;**

Exemple :

```
DECLARE
```

```
        nb_vols NUMBER(2,0) ;
        impossible EXCEPTION ;
        numero vol.num_vol%TYPE ;
        ...
BEGIN
        SELECT COUNT(*) INTO nb_vols FROM vol ;
        ...
        IF numero > 10000 THEN RAISE impossible ;
        ...
EXCEPTION
        WHEN impossible THEN numero := 0 ;
        WHEN OTHERS THEN numero := 100 ;
END ;
```

La clause **WHEN OTHERS** permet de traiter les autres exceptions.

6. Ordres SQL valides en PL/SQL

Les ordres SQL qui peuvent être utilisés dans un bloc PL/SQL sont les suivants :

- Toute requête d'interrogation : **SELECT** ... **FROM** .....
- Les opérations de mise à jour des données : **INSERT**, **UPDATE**, **DELETE**.
- Les ordres de gestion de transaction : **COMMIT**, **ROLLBACK**, **SAVEPOINT**.
    - **COMMIT** et **ROLLBACK** permettent de définir une transaction. En cas d'incident, les ordres de mise à jour des données pourront alors être "défaits" par **ROLLBACK** via une exception.
    - **SAVEPOINT** <nom_point_sauve> et **ROLLBACK TO** <nom_point-sauve>

  peuvent être utilisés pour ne défaire qu'une partie de la transaction.

7. Les curseurs

Le rôle des curseurs est d'établir la transition entre l'univers BD et celui des langages procéduraux classiques. Plus précisément, un curseur permet de "récupérer" un ensemble de tuples, résultat d'une requête d'interrogation sur la base de données et de le manipuler de manière procédurale, i.e. en balayant un par un les tuples/enregistrements.

Définition :

- Un curseur est défini dans la partie déclarative d'un bloc PL/SQL par une requête d'interrogation en SQL (sa structure correspond aux attributs du SELECT), en suivant la syntaxe suivante :

  CURSOR <nom_curseur> **IS** <requête_SQL> **;**

Exemple :

```
CURSOR C_pilote IS
        SELECT num_pil, nom_pil
```

FROM pilote
ORDER BY num_pil, nom_pil ;

- Instructions de gestion des curseurs :

  - **OPEN** <nom_curseur> ;
    exécute la requête de définition du curseur et alloue la place mémoire nécessaire. Le curseur peut alors être perçu comme une suite d'enregistrements.
  - **CLOSE** <nom_curseur> ;
    désactive le curseur et libère la place mémoire. Le curseur est alors perçu comme un ensemble indéfini.
  - **FETCH** <nom_curseur> **INTO** <liste_variables> ;
    ramène le prochain enregistrement du curseur et renseigne les différentes variables réceptrices.

- Informations caractéristiques des curseurs :

Ces informations correspondent à des propriétés booléennes prédéfinies des curseurs.

  - <nom_curseur**>%NOTFOUND**
    est à vrai si l'ordre FETCH ne retourne aucun enregistrement.
  - <nom_curseur>**%FOUND**
    est à vrai si l'ordre FETCH retourne un enregistrement.
  - <nom_curseur>**%ISOPEN**
    est à vrai si le curseur est ouvert.
  - <nom_curseur>**%ROWCOUNT**
    retourne le nombre de tuples qui ont été accédés via le curseur (0 avant le 1er **fetch**, puis 1, puis 2 ...).

- La manipulation de l'enregistrement courant d'un curseur est possible en utilisant simplement le nom du curseur.

Exemples :

- Un curseur qui affiche le nom des pilotes Parisiens

```
DECLARE
CURSOR MesPilotesParisiens IS
        SELECT * FROM pilote
        WHERE adr= 'Paris';
mon_pilote pilote%ROWTYPE;
BEGIN
        OPEN MesPilotesParisiens;
        LOOP
                FETCH MesPilotesParisiens INTO mon_pilote;
        DBMS_OUTPUT.PUT_LINE(mon_pilote.plnom);
                EXIT WHEN MesPilotesParisiens%NOTFOUND;
        END LOOP;
        CLOSE MesPilotesParisiens;
END;
```

- Quelques utilisations des propriétés booléennes (%ISOPEN, %FOUND, %NOTFOUND)

```
IF NOT lecurseur%ISOPEN THEN
OPEN lecurseur;
END IF;

OPEN lecurseur;
LOOP
        FETCH lecurseur INTO variable1,variable2;
        EXIT WHEN NOT lecurseur%FOUND;
END LOOP;
CLOSE lecurseur;

OPEN lecurseur;
LOOP
        FETCH lecurseur INTO variable1,variable2;
        EXIT WHEN lecurseur%NOTFOUND;
END LOOP;
CLOSE lecurseur;
```

Si l'ordre **SELECT** de définition du curseur comporte un calcul (horizontal ou vertical), il faut absolument attribuer un alias à ce calcul pour pouvoir le manipuler ultérieurement.

Exemple :

```
CURSOR comptage IS
        SELECT ville_dep, COUNT(*) nb_arrivees
        FROM vol
        GROUP BY ville ;
```

Le nombre de vols desservant chaque ville peut alors être manipulé par comptage.nb_arrivees

- Gestion du parcours d'un curseur

Un curseur peut être géré soit automatiquement par le système, soit par l'utilisateur lui-même dans son programme PL/SQL.

- Gestion automatique : Evidemment cette possibilité est préférable à la suivante, puisque le système se charge de balayer les enregistrements du curseur. Pour cela, l'instruction de parcours à spécifier est la suivante :

  FOR<nom_variable>**IN**<nom_curseur> **LOOP**<liste_instructions>**END LOOP**;

Remarques :

- Il est inutile de déclarer <nom_variable>.
- Les ordres FETCH sont inutiles.
- L'ouverture et la fermeture du curseur sont implicites. Le système s'en charge et vous ne devez pas utiliser OPEN et CLOSE.
- Dans <liste_instructions>, les données manipulées sont référencées par :

  <nom_variable>.<nom_attribut>

Exemple :

FORC1 **IN** comptage **LOOP**
IF C1.nb_arrivees < 10 **THEN** ...
END LOOP;

- Gestion par l'utilisateur
  - Dans ce cas de figure, l'utilisateur doit effectuer :
  - l'ouverture et la fermeture du curseur ;
  - gérer le passage d'un enregistrement à un autre ;
  - contrôler l'arrivée au dernier enregistrement du curseur ;
  - déclarer les variables réceptrices.

Exemple :

```
DECLARE
NUMERO ETUDIANT.NUM_ET%TYPE ; MOY NUMBER(4) ;
CURSOR COMPT AGE IS
        SELECT NUM_ET, AVG(NOTE_TEST) Moyenne FROM NOTATION
        GROUP BY NUM_ET ;
BEGIN
OPEN COMPTAGE ;
LOOP
        FETCH COMPTAGE INTO NUMERO, MOY ;
        EXIT WHEN COMPTAGE%NOTFOUND
        IF NUMERO < 10 THEN ...
END LOOP ;
CLOSE COMPTAGE ;
END ;
```

Remarque : Il est possible dans cet exemple d'utiliser une variable réceptrice de même structure que le curseur (déclarée par : var COMPTAGE**%ROWTYPE**) à la place des variables NUMERO, MOY.

8. Procédures et fonctions stockées

Les procédures et fonctions sont des programmes PL/SQL qui peuvent être stockés dans la base de données et être ainsi ré-utilisables et partageables (C.f. autorisations).

Définition

- Les procédures :

```
CREATE[OR REPLACE] PROCEDURE nom_procedure
        /* Déclaration des paramètres */
(var_entree IN type, var_sortie OUT type, var_entrée_sortie IN OUT type) IS
        /* Déclaration des variables locales*/
var_locale type;
BEGIN
<liste_instructions>
[EXCEPTION ...]
END ;
```

- Les fonctions permettant de retourner un résultat :

```
CREATE [OR REPLACE] FUNCTION nom_fonction
        /* Déclaration des paramètres */
 (var_entree IN type, ...)
RETURN type IS
        /* Déclaration des variables locales*/
var_locale type;
BEGIN
<liste_instructions>
RETURN (var_locale) ;
[EXCEPTION ...]
END ;
```

Utilisation et suppression

Après avoir stocké la procédure ou la fonction sous SQLPlus (start ou @ nom_fichier), elles peuvent être appelées :

- Au sein d'un programme PL/SQL

  Procédure : nom_procedure (param) ;
  Fonction : Variable := nom_fonction(param) ;

- Au niveau de SQL/PLUS par :

  Procédure : **EXECUTE** nom_procedure (param) ;
  Fonction : **SELECT** att1, att2, nom_fonction (param)
  FROM

Pour supprimer les procédures ou fonctions stockées :
DROP PROCEDURE nom_procedure ;
Ou
DROP FUNCTION nom_fonction

Exemples :

- *Fonction qui retourne le nombre de vol*

```
CREATE OR REPLACE FUNCTION nb_vol ( num IN INTEGER)
RETURN INTEGER IS
nb INTEGER ;
BEGIN
        SELECT COUNT(num_vol) INTO nb
        FROM vol
        WHERE num_pil = num ;
        RETURN (nb) ;
END ;
```

- Procédure qui recupère le nom d'un pilote par rapport à son numéro et appel de la procédure dans un bloc PL/SQL

```
CREATE OR REPLACE PROCEDURE nom_pil (
        numero IN pilote.plnum%type,
        nom OUT pilote.plnom%type) IS
BEGIN
        SELECT plnom INTO nom FROM pilote WHERE numero = plnum ;
END;

DECLARE
        LeNomPilote VARCHAR2(100) ;
BEGIN
nom_pil(100,leNomPilote) ; -- appel de la procedure
dbms_output.put_line(leNomPilote) ;
END;
```

La fonction nb_vol retourne le nombre de vols du pilote passé en paramètre. Son utilisation peut être réalisée par une requête SQL nous permettant d'obtenir pour chaque pilote le nombre de vols qu'il assure :

SELECT num_pil, nb_vol (num_pil) 'Nombre de Vol' **FROM** pilote ;

9. Les entrées/sorties

Pour pouvoir **afficher du texte à l'écran** utilisation du package DBMS_OUTPUT : au niveau du prompt SQLPlus, exécuter l'instruction suivante :

set serveroutput on

Dans le corps de vos programmes, l'instruction pour afficher du texte ou le contenu d'une variable est :

dbms_output.put_line('texte' || variable) ;

où || est le caractère permettant la concaténation de chaînes.

dbms_output.enable (autorise l'affichage)
dbms_output.disable (interdit l'affichage)
dbms_output.put_line (affiche la chaine et passe à la ligne)
dbms_output.new_line (passe à la ligne)

La commande **show errors** permet de visualiser le détail des erreurs en cours.

SQLPlus permet l'utilisation de variables de substitution afin que des valeurs soient saisies par l'utilisateur à l'exécution : utilisation du **&**
Attention le texte affiché à l'utilisateur sera celui associé au caractère & (pas de blanc ni d'accent)

SQL> SELECT COUNT(num_vol) **FROM** vol **WHERE** num_pil = **&**numero_du_pilote ;
Enter value for numero_du_pilote: