PL/SQL

Marc Plantevit

marc.plantevit@liris.cnrs.fr

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.

Utilisation de PL/SQL (suite)

Le PL/SQL peut être utilisé sous 3 formes :

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

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.
- Considérons d'abord les blocs anonymes.

Structure d'un bloc anonyme

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

Seuls BEGIN et END sont obligatoires

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

Déclaration, initalisation 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 et initialisation
  - `Nom_variable type_variable := valeur ;`
- Initialisation
  - `Nom_variable := valeur ;`
- Déclarations multiples interdites.
- Exemples :
  - `age integer ;`
  - `nom varchar(30) ;`
  - `dateNaissance date ;`
  - `ok boolean := true ;`

Plusieurs façons d'affecter une valeur à une variable

- **Opérateur d'affectation** `n :=`.
- **Directive INTO** de la requête SELECT.

Exemple

- `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.

Types de Variables

VARCHAR2

- Longueur maximale : 32767 octets ;
- Syntaxe : Nom_variable VARCHAR2(30) ;
- Exemples :
  name VARCHAR2(30) ;
  name VARCHAR2(30) := 'toto' ;

NUMBER(long,dec)

- Long : longueur maximale ;
- Dec : longueur de la partie décimale ;
- Exemples :
  num_telnumber(10) ;
  toto number(5,2)=142.12 ;

DATE

- Par défaut DD-MON-YY (18-DEC-09) ;
- Fonction TO_DATE ;
- Exemples :
  start_date := to_date('29-SEP-2003','DD-MONYYYY') ;
  start_date :=
  to_date('29-SEP-2003 :13 :01','DD-MONYYYY :HH24 :MI') ;

BOOLEAN

- TRUE
- FALSE
- NULL

Déclaration %TYPE et %ROWTYPE

`v_nom emp.nom.%TYPE ;`

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

`v_employe emp%ROWTYPE ;`

Une variable peut contenir toutes les colonnes d'un tuple d'une table (la variable `v_employe` contiendra une ligne de la table `emp`.

Exemple

```
DECLARE
-- Déclaration
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 ;
...
/* On insère un tuple dans la base*/
INSERT into emp VALUES v_employe ;
END ;
```

Vérifiez à bien retourner un seul tuple avec la requête SELECT ... INTO ...

Test Conditionnel

IF-THEN

```
IF v_date > '01-JAN-08' THEN
v_salaire := v_salaire * 1.15 ;
END IF ;
```

IF-THEN-ELSE

```
IF v_date > '01-JAN-08' THEN
v_salaire := v_salaire * 1.15 ;
ELSE
v_salaire := v_salaire * 1.05 ;
END IF ;
```

IF-THEN-ELSIF

```
IF v_nom = 'PARKER' THEN
v_salaire := v_salaire * 1.15 ;
ELSIF v_nom = 'SMITH' THEN
v_salaire := v_salaire * 1.05 ;
ENDIF ;
```

CASE

```
CASE sélecteur
WHEN expression1 THEN résultat1
WHEN expression2 THEN résultat2
ELSE résultat3
END ;
```

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

Exemple

```
 val := CASE city
WHEN 'TORONTO' THEN 'RAPTORS'
WHEN 'LOS ANGELES' THEN 'LAKERS'
WHEN 'SAN ANTONIO' THEN 'SPURS'
ELSE 'NO TEAM'
END ;
```


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 ...

FOR

- FOR variable IN debut .. fin
  LOOP
  instructions ;
  END LOOP ;
- La variable de boucle prend successivement les valeurs de *debut*, *debut* + 1, *debut* + 2, ..., jusqu'à la valeur *fin*.
- On pourra également utiliser un curseur dans la clause IN (dans quelques slides).

Affichage

Activer le retour écran :
`set serveroutput on size 10000`

Affichage

- dbms_output.put_line(chaîne) ;
- Utilise || pour faire une concaténation.

Exemple

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

Si '/' seul sur une ligne : fin d'une définition (déclenche l'évaluation).

Exemple bis

```
 DECLARE
compteur number(3) ;
i number(3) ;
BEGIN
select count(*) into compteur from EtudiantLIF10 ;
FOR i IN 1..compteur LOOP
dbms_output.put_line('Nombre : L3IF ' || i ) ;
END LOOP ;
END ;
```


Overview

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 ;
...
END ;
```


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.

Exemple Curseurs Explicites

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

Ne pas oublier de fermer le curseur.

Type Row associé à un curseur

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

```
 DECLARE
CURSOR c IS SELECT matr, nom, sal FROM emp ;
employe c%ROWTYPE ;
BEGIN
open c ;
fetch c into employe ;
IF employe.sal IS NOT 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).

Exemple

```
 DECLARE
-- Déclaration du paramètre du curseur.
CURSOR c(p_dept integer) IS select dept, nome from
emp where dept = p_dept ;
BEGIN
-- Instantiation du paramètre avec le dept 10.
FOR employe in c(10) LOOP
dbms_output.put_line(employe.nome) ;
END LOOP ;
-- Instantiation du paramètre avec le dept 20.
FOR employe in c(20) LOOP
dbms_output.put_line(employe.nome) ;
END LOOP ;
END ;
```

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**.

```
 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 ;
```


Bloc anonyme ou nommé

- Un bloc anonyme PL/SQL est un bloc « **DECLARE –BEGIN –END** » comme dans les exemples précédents.
- On peut exécuter directement un bloc PL/SQL anonyme en tapant sa définition.
- Le plus souvent, on passe 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 ;
```



```
CALL list_nom_clients() ;
```

Procédures avec paramètres

```
 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

Un peu plus ...

- Déclarer une variable :
  `variable nb number ;`
- Exécuter la fonction :
  `execute list_nom_clients('paris', :nb) ;` (Une variable globale s'utilise avec le préfixe :)
- Visualisation du résultat :
  `print ;`
- Description des paramètres :
  `desc nom_procedure`

Fonctions sans paramètre

```
 CREATE OR REPLACE FUNCTION nombre_clients
-- Déclaration du type de retour de la fonction
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 ;
```


- Exécution :

```
select nombre_clients() from dual ;
```

Fonctions avec paramètres

- Seuls les paramètres 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 ;
/
show errors ;

-- appel :
select euro_to_fr(10) from dual ;
```


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` ;
- 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.

Les déclencheurs (triggers)

- Les contraintes prédéfinies ne sont pas toujours suffisantes.
Ex. : *tout nouveau prix d'un produit doit avoir une date de début supérieure à celle des autres prix pour ce produit (pas de prix antidatés).*
- Automatiser des actions lors de certains événements du type : AFTER ou BEFORE et INSERT, DELETE ou UPDATE
- Syntaxe :

```
CREATE OR REPLACE TRIGGER nom_trigger
Evénement [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**.
- Dansle 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, TYPEMODIF.

```
 create or replace trigger logadd
after insert on clients
for each row
BEGIN
  insert into log_clients values
( :new.nom, :new.adresse, :new.reference, :new.nom_piece, :new.qua
USER,SYSDATE,'INSERT') ;
END ;
```


Prédicats Conditionnels

- Lorsqu'un trigger a plusieurs opérations déclenchantes (ON INSERT OR DELETE OR UPDATE OF EMP), le corps du trigger peut avoir des prédicats conditionnels :

```
 IF INSERTING THEN ... END IF ;
IF UPDATING THEN ... END IF ;
```

- On peut préciser les colonnes soumises aux opérations déclenchantes :

```
 CREATE TRIGGER ...
... UPDATE OF sal, commission ON EMP ...
BEGIN
  ...
   IF UPDATING('SAL') THEN
     ...
  END IF
END ;
```


- Les commandes de définition de données (LDD) et les commandes de contrôle de transactions ne sont pas permises dans le corps d'un trigger. Ainsi, les commandes ROLLBACK, COMMIT, ..., ne doivent pas être utilisées dans le corps d'un trigger.

Tables Mutantes et Tables Contraignantes

- Une table **mutante** est une table en cours de modification par une opération déclenchante (UPDATE, DELETE, INSERTION) ou l'effet de DELETE CASCADE provenant de cette opération.
- Une table **contraignante** est une table qu'une opération déclenchante doit lire, soit directement via une commande SQL (UPDATE SET ... WHERE) ou indirectement pour une contrainte d'intégrité référentielle.
- Les commandes SQL dans le corps d'un trigger ne peuvent pas :
  - Lire (par une requête) ou modifier un table mutante d'une opération déclenchante.
  - Changer des valuers sur les colonnes de clés (PRIMARY, FOREIGN, UNIQUE) d'une table contraignante.
- Ces restrictions permettent d'éviter la consultation d'une table dans un état transitoire et donc incohérent.

Exemple d'erreur

```
 CREATE OR REPLACE TRIGGER emp_count
AFTER DELETE ON emp
FOR EACH ROW
DECLARE
n INTEGER ;
BEGIN
  SELECT COUNT(*) INTO n FROM emp ;
DBMS_OUTPUT.PUT_LINE('On a '|| n || 'employés dans la
base') ;
```

DELETE FROM emp WHERE empno = 7499 ;

On a l'erreur suivante :

```
ORA-04091 : table SCOTT.EMP is mutating,
trigger/function may not see it.
```


Le dictionnaire de données a des vues sur les triggers : `USER_TRIGGERS, ALL_TRIGGERS, DBA_TRIGGERS`.

Exemple : `SELECT trigger_type, triggering_event, table_name FROM user_triggers ;`