Fondements des bases de données

Programmation en PL/SQL Oracle

Marc Plantevit

http://liris.cnrs.fr/~mplantev/doku/doku.php?id=lif10

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)

Outline

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

- Seuls `BEGIN` et `END` sont obligatoires
- Les blocs 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 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`.

Exemples

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

Attention

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

SELECT ...INTO ...

Instruction 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 ;
- 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

- 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`).

Important pour la robustesse du code

Exemple

```
DECLARE
   -- Declaration
   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;
...
   -- Insertion d'un tuple dans la base
   INSERT into emp VALUES v_employe;
END;
```

Vérifier à bien retourner *un seul tuple*
avec la requête `SELECT ...INTO ...`

Outline

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;
```

Test conditionnel (2)

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;
END IF;
```

CASE

```
CASE selection
  WHEN expression1 THEN resultat1
  WHEN expression2 THEN resultat2
   ...
   ELSE resultat
END;
```

`CASE` renvoie une valeur qui vaut `resultat1` ou `resultat2` ou ...ou `resultat` par défaut.

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;
EXIT [WHEN condition];
   instructions;
END LOOP;

WHILE condition LOOP
   instructions;
END LOOP;
```

Exemple

```
LOOP
    monthly_value := daily_value * 31;
    EXIT WHEN monthly_value > 4000;
END LOOP;
```

Obligation d'utiliser la commande **EXIT**
pour éviter une boucle infinie.

FOR

```
FOR variable IN [REVERSE] 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).
- Le mot clef `REVERSE` à l'effet escompté.

Exemple

```
FOR Lcntr IN REVERSE 1..15
LOOP
    LCalc := Lcntr * 31;
END LOOP;
```

Affichage

- Activer le retour écran : `set serveroutput on size 10000`
- Sortie standard : `dbms_output.put_line(chaîne);`
- Concaténation de chaînes : opérateur `||`

Exemple

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

Le caractère / seul sur une ligne déclenche l'évaluation.

Affichage

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;
```

Outline

Les curseurs

Toutes les requêtes SQL sont associées à un curseur :

- Ce curseur représente la zone mémoire utilisée pour analyser et exécuter la requête.
- Le curseur peut être implicite (pas déclaré par l'utilisateur) ou explicite.
- Les curseurs explicites permettent de manipuler l'ensemble des résultats d'une requête.

Les curseurs implicites sont tous nommés `SQL`

```
DECLARE
  nb_lignes integer;
BEGIN
  DELETE FROM emp WHERE dept = 10;
  nb_lignes := SQL%ROWCOUNT;
  ...
END;
```

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 explicites

Pour traiter les *SELECT* qui renvoient plusieurs lignes

- Les curseurs doivent être déclarés explicitement.
- Le code doit les utiliser avec les commandes
  - `OPEN moncurseur`, pour ouvrir le curseur ;
  - `FETCH moncurseur`, pour avancer le curseur à la ligne suivante ;
  - `CLOSE moncurseur`, pour refermer le curseur

Utilisation

- On utilise souvent les curseurs dans une boucle `FOR` qui permet une utilisation *implicite* des instructions `OPEN`, `FETCH` et `CLOSE`.
- Généralement, on sort de la boucle quand l'attribut `NOTFOUND` est vrai.

Les curseurs explicites

Exemple de boucle FOR pour les curseurs

```
DECLARE
CURSOR salaires IS
      SELECT sal
      FROM emp;
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;
```

Les curseurs explicites

Déclaration d'un type 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 IF;
END;
```

Boucle FOR pour un curseur

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

Exemple

```
DECLARE
  CURSOR c_nom_clients IS
  SELECT nom, adresse
  FROM clients;
BEGIN
  FOR le_client IN c_nom_clients LOOP
    dbms_output.put_line(
      'Employe : ' || UPPER(le_client.nom) ||
      ' Ville : '|| le_client.adresse);
  END LOOP;
END;
```

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 si on utilise pas la boucle `FOR` dédiée.

Exemple

```
DECLARE
  CURSOR c(p_dept integer) IS
    SELECT dept, nom
    FROM emp
    WHERE dept = p_dept;
BEGIN
  FOR employe in c(10) LOOP
  dbms_output.put_line(employe.nom);
END LOOP;
FOR employe in c(20) LOOP
  dbms_output.put_line(employe.nom);
END LOOP;
```

Outline

Les exceptions

Une exception est une erreur qui survient durant une exécution, elle est soit :

- prédéfinie par Oracle,
- définie par le programmeur.

Exceptions prédéfinies

`NO_DATA_FOUND` quand `SELECT ...INTO` ne retourne aucune ligne.

`TOO_MANY_ROWS` quand `SELECT ...INTO` retourne plusieurs lignes.

`VALUE_ERROR` érreur numérique.

`ZERO_DIVIDE` division par zéro

`OTHERS` toutes erreurs non interceptées.

Traitement des exceptions

Saisir une exception

- Une exception ne provoque pas nécessairement l'arrêt du programme : elle peut être saisie par une partie `EXCEPTION`.
- Une exception non saisie remonte dans la procédure appelante (où elle peut être saisie).

Exemple

```
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`

Exceptions utilisateur

Exemple

```
DECLARE
   v_salaire numeric(8,2);
   salaire_trop_bas EXCEPTION;
BEGIN
   SELECT sal INTO v_salaire
   FROM emp
   WHERE matr = 50;
   IF v_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;
```

Outline

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.
- On passe plutôt une procédure ou une fonction nommée pour réutiliser le code.

Procédure sans paramètre

Exemple

```
CREATE OR REPLACE PROCEDURE list_nom_clients IS
BEGIN
  DECLARE
    CURSOR c_nom_clients IS
      SELECT nom, adresse
      FROM clients;
  BEGIN
    FOR le_client IN c_nom_clients LOOP
      dbms_output.put_line(
        'Client : ' || UPPER(le_client.nom) ||
        ' Ville : ' || le_client.adresse);
    END LOOP;
  END;
END;
SET SERVEROUTPUT ON;
CALL list_nom_clients();
```

Procédure avec paramètres

Exemple

```
CREATE OR REPLACE PROCEDURE
   list_nom_clients(ville IN varchar2, result OUT number)
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;
SET SERVEROUTPUT ON;
CALL list_nom_clients();
```

`IN`	lecture seule
`OUT`	écriture seule
`IN OUT`	lecture et écriture

Fonctions sans paramètre

Exemple

```
CREATE OR REPLACE FUNCTION nb_clients
  RETURN NUMBER -- Type de retour
  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;
      CLOSE get_nb_clients;
    RETURN i;
  END;
END;
```

Fonctions avec paramètres

Exemple

```
CREATE OR REPLACE FUNCTION euro_to_fr(v_somme IN number)
  RETURN NUMBER
  IS
  BEGIN
    DECLARE
      taux CONSTANT number := 6.55957;
    BEGIN
      RETURN v_somme * taux;
    END;
  END;
END;

SELECT euro_to_fr(15.24)
FROM dual;
```

Seuls les paramètres `IN` (en lecture seule) sont autorisés

Un peu plus ...

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

Outline

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*
- Exécuter des actions lors de certains événements :
  - `AFTER` ou `BEFORE`
  - `INSERT`, `DELETE` ou `UPDATE`
  - `FOR EACH ROW`
    - non (`STATEMENT`) : exécuté *une seule fois* pour la commande.
    - oui (`ROW`) : exécuté *à chaque ligne* concernée.

Syntaxe

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER} {INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name] ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
BEGIN
  --- sql statements
END;
```

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 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 les colonnes `USERNAME`, `DATEMODIF`, `TYPEMODIF`.
- Créer un trigger `AFTER INSERT ON clients`

Accès aux valeurs modifiées

Exemple

```
CREATE or REPLACE TRIGGER logadd
AFTER INSERT ON clients
FOR EACH ROW
BEGIN
  INSERT INTO log_clients values(
     :new.nom,
     :new.adresse,
     ...
     :new.echeance,
     USER,
     SYSDATE,
     'INSERT');
END;
```

Prédicats conditionnels

- Lorsqu'un trigger a plusieurs opérations déclenchantes le corps peut avoir des prédicats conditionnels :
  - `IF INSERTING THEN ... END IF;`
  - `IF UPDATING THEN ... END IF;`
- On peut préciser les colonnes soumises aux opérations déclenchantes

Exemple

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

Important

Interdiction dans le coprs des triggers

- Les commandes de définition de données (LDD)
- les commandes de contrôle de transactions (`ROLLBACK`, `COMMIT`)

Ne doivent pas être utilisées dans le corps d'un trigger.

Tables mutantes et contraignantes

- Une table mutante est une table *en cours de modification* par une opération déclenchante (`UPDATE`, `DELETE`, `INSERT`) 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 valeurs 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 || ' employes dans la base');
  END;

DELETE FROM emp WHERE empno = 7499;
```

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

Dans ce cas là, il ne faut pas utiliser `FOR EACH ROW` pour pouvoir déterminer la valeur du `SELECT COUNT(*) FROM emp;`.

☛ 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;`

Fin du cinquième cours.