Le Langage PL/SQL de Oracle (Brève Introduction)

Najib Tounsi
Ecole Mohammadia d'Ingénieurs, Rabat

Année 2016/2017
1ère année Génie Informatique
http://www.emi.ma/~ntounsi/COURS/DB/Polys/SQL/PLSQL/PLSQL.html
Voir aussi PLSQL par l'exemple dans les travaux pratiques

Sommaire

Introduction

- *PL/SQL* est un langage procédural (*Procedural Language/SQL*) qui permet en plus de SQL d'avoir les mécanismes des langages algorithmiques.
- L'unité de programmation PL/SQL est le *bloc*.
- Un bloc contient des déclaration de variables ou de constantes et des instructions (qui peuvent être des commandes SQL).
- Des variables peuvent être utilisées pour mémoriser des résultats de requêtes.

Introduction (suite)

- Parmi les instructions d'un bloc PL/SQL on a:
  - commandes SQL donc,
  - instructions de boucle (**LOOP**),
  - instructions conditionnelles (**IF-THEN- ELSE**),
  - traitement des exceptions,
  - appels à d'autres blocs PL/SQL.
- Un bloc PL/SQL peut-être une fonction (ou une procédure).
- Voir Procédures et Fonction PLSQL.

Introduction (suite)

- Les blocs PL/SQL qui définissent des fonctions ou procédures, peuvent être groupés en *packages*.
- Un *package* est comme un module, il a une partie interface et une partie implémentation.
- PL/SQL permet de traiter le résultat d'une requête *tuple par tuple*.
- La notion de ***CURSOR*** sert à cet effet.

Structure des blocs PL/SQL

- Un bloc est l'unité de programme en PL/SQL.
- Un bloc a un nom quand il définit une fonction, une procédure ou un package.
- Les blocs peuvent être imbriqués.
- Un Bloc a:
  - une partie déclarations (optionnelle),
  - une partie instructions,
  - et une partie (optionnelle) de traitement d'exceptions.

Structure des blocs PL/SQL (suite)

- Structure d'un bloc PL/SQL
  NB. `[...]` signifie optionnel, `<...>` pour partie programmeur

```
[<Enête de bloc>]
[DECLARE
      <Constantes>
      <Variables>
      <Cursors>
      <Exceptions utilisateurs>]
BEGIN
      <Instruction PL/SQL>
      [EXCEPTION <Traitement d'exception>]
END;
```

- Entête de bloc: si fonction, procédure ou package. Sinon, bloc anonyme.
-

Programme "Hello world"

```
SQL>   SET SERVEROUTPUT ON
SQL>    BEGIN
2    dbms_output.putline('Bonjour');
3    END;
4    /

Bonjour
```

- Un programme contient au moins `BEGIN <une insctruction> END;`
- `dbms_output.putline` permet d'écrire sur la console SQLPlus.
- La sortie écran ne marche que si le serveur d'impression est ouvert. Faire `SET SERVEROUTPUT ON` sous SQLPlus pour basculer sur le mode sortie console.
- La ligne 4 commence par un `/`. C'est la demande d'exécution du programme tapée (donc la fin de saisie).

Déclarations de variables

- Exemple de déclarations

```
DECLARE
     dateEmbauche date;   /* initialisation implicite avec null */
     nom varchar2(80) := 'Benali';
     trouve boolean;     /* initialisation implicite avec null aussi*/
     incrSalaire constant number(3,2) := 1.5;    /* constante */
      . . .
BEGIN . . . END;
```

- *null* est valeur par défaut, si pas d'initialisation (sauf si `NOT NULL` spécifié)
- Types usuels de ORACLE.
- Mais les variables les plus intéressantes en PLSQL sont celles qui vont contenir des données de la base. Variable de même type qu'un attribut (colonne) ou de même type qu'un schéma (ligne).

Déclaration de variable par référence aux données de la base

Variable de type colonne

```
DECLARE
     maxSal employee.salary%TYPE;
```

- *maxSal* est une variable PLSQL de même type que la colonne *salary* de la table *employee*.
- Utile pour garantir la compatibilité des affectations (e.g. dans la clause `INTO`, voir plus bas)

Variable de type ligne

```
DECLARE
    employeeRec employee%ROWTYPE;
```

- *employeeRec* est une structure de tuple de même type que le schéma de la relation *employee*.
- Usage avec `SELECT *` ... (voir plus bas)

Déclarations de zone *cursor*

- Le mot CURSOR sert à déclarer une zone mémoire qui recevra le résultat d'un *SELECT* pour un parcours tuple par tuple.

```
DECLARE
     CURSOR empCursor IS
     SELECT * FROM employee WHERE dept = 123;
```

- ou bien avec paramètres formels

```
DECLARE
    CURSOR empCursor (dno number ) IS
    SELECT * FROM employee WHERE dept = dno;
```

  - Le paramètre *dno* sera passé lors de *OPEN*.

```
OPEN empCursor (123);
```

- Si un CURSOR est utilisé pour mettre à jour un tuple de relation, on le signale avec *FOR update* en fin de déclaration

```
DECLARE
    CURSOR empCursor IS
    SELECT * FROM employee WHERE dept = 123
       FOR UPDATE OF salary;
```

- ...

Instructions

- PL/SQL offre la plupart des constructions des langages de programmation: affectation de variables, structures de contrôle de boucle (*LOOP*) et de teste (*if-then- ELSE*), appels de procédure et de fonction, etc.

```
DECLARE
     quantite integer := 0;
      ...
BEGIN
     quantite := quantite + 5;
        ...
END
```

- PL/SQL ne permet pas les commandes SQL de langage de définition comme la création de table, la modification d'attribut etc.
- PL/SQL permet tous les autres types de commandes SQL (*insert*, *delete*, *update*, *commit* ...)
- La commande ***SELECT*** (sauf si imbriqué) est **toujours** utilisé avec ***into***, pour affecter les données retrouvées aux variables PL/SQL.

Exemple d'interrogation *Select* mono-tuple

- PL/SQL permet d'affecter chaque tuple résultat d'un `SELECT` à une structure ou à une liste de variable (cf. *SELECT ... INTO ...*)

```
DECLARE
   employeeRec employee%ROWTYPE;
   maxSal employee.salary%TYPE;
BEGIN
   SELECT * INTO   employeeRec
   FROM employee WHERE  enum='E7';
   dbms_output.putline(employeeRec.ename||' '||employeeRec.Salary);
END;
```

- Ici c'est un `SELECT` *mono-tuple*! (la clause `WHERE` porte sur la clé *enum*)
- C'est pour que le résultat soit affecté avec la clause INTO à une variable PLSQL. (Pour plusieurs tuples il faut une variable `CURSOR`).
- Laquelle variable, *employeeRec*, est de type `ROWTYPE` dans cet exemple. Contient un tuple donc.
- La requête spécifie `*` dans la clause `SELECT`.
- remarquer la notation `employeeRec.ename` (*variableTuple.attribut*) pour accéder aux différents composants qui sont donc ceux déclarés dans le schéma de la relation.

Exemple d'interrogation *Select* mono-tuple (suite)

- Forme plus légère de la même requête: on va accéder à des composants bien définis.

```
BEGIN
    SELECT ename, salary INTO    employeeRec.ename, employeeRec.Salary
    FROM employee
    WHERE   enum = 'E8';

    dbms_output.putline(employeeRec.ename||' '||employeeRec.Salary);
END;
```

- Le résultat est récupéré dans les champs de *employeeRec*, mais on aurait pu utiliser des variables simples, comme dans l'exemple suivant.

```
BEGIN
    SELECT max(salary) INTO   maxSal
    FROM employee;

    dbms_output.putline('Salaire Maximum: '|| maxsal);
END;
```

- Ici on a un résultat de calcul qui est affecté à la variable PLSQL *maxSal*.

Structure de contrôle *if-then-else*

- Sémantique analogue aux autres langages

```
IF <condition> THEN   <séquence d'instructions>
[ELSIF   <condition> THEN   <séquence d'instructions> ]
. . .
[ELSE   <séquence d'instructions> ]
END IF ;
```

- Usage de `ELSIF` pour suite de tests et `END IF` pour finir le `IF`.

Structures de contrôles (boucle *LOOP*)

- Boucle *WHILE*

```
[<label>]
WHILE <condition> LOOP
   <séquence  d'instructions>;
END LOOP  [<label>] ;
```

- Exemple

```
DECLARE
    i number;
BEGIN
   i:=0;
   WHILE  i<8 LOOP
      dbms_output.put_line(i);
      i:= i+1;
   END LOOP ;
END;
```

- On peut nommer une boucle pour, en cas de boucles imbriquées, s'y référer avec *EXIT* par exemple.

Structures de contrôles (boucle *LOOP*)

- Boucle *for*

```
[<label name>]
FOR <index> IN  [reverse] <lower bound>..<upper bound> LOOP
    <séquence  d'instructions> ;
END LOOP  [<label name>] ;
```

- L'index est déclaré implicitement.
- Exemple:

```
BEGIN
  FOR i IN  4..7 LOOP
     dbms_output.put_line(i);
  END LOOP ;
END;
```

Structures de contrôles (boucle *LOOP*)

- Une autre forme de boucle (infinie) est

```
LOOP  ... END LOOP ;
```

- Arrêt avec

```
EXIT WHEN ...
```

- ... survenue d'une exception.
- Utilisée surtout avec *CURSOR*.

Structures de contrôles (boucle *LOOP* avec *cursor*)

- La forme *LOOP ... END LOOP* , est utilisée avec un *cursor*.
- Calcul du salaire maximum en comparant les salaires de tous les employés.

```
DECLARE
 CURSOR   empCursor IS
           SELECT * FROM EMPLOYEE;
    employeeRec employee%ROWTYPE;
    maxSal employee.SALary%TYPE := 0;
BEGIN
   OPEN empCursor;
   LOOP
            /* Accès à chacun des tuples */
         FETCH empCursor INTO   employeeRec;
         EXIT WHEN empCursor%NOTFOUND;
              /* traitement du tuple */
 IF   maxSal < employeeRec.salary THEN
                   maxSal := employeeRec.salary;
         END IF;
              /* fin traitement tuple */

 END LOOP ;
      dbms_output.putline('Salaire Maximum: '|| maxsal);
      CLOSE empCursor;
END;
```

- Noter le INTO dans `FETCH` (au lieu de `select`. Pourquoi?)
- `%NOTFOUND` est un attribut boolean du CURSOR *empCursor*.
- Après *`OPEN`*, `empCursor%NOTFOUND` est évaluée à *null*. Après un `FETCH` elle est évaluée à *false* si un tuple est retrouvé, à *true* sinon.
- La boucle finit donc dès que la condition `EXIT WHEN` est vérifiée (aucun tuple retrouvé par `FETCH`).
- Un CURSOR ouvert doit être fermé avec `CLOSE empCursor;`.
- NB. La requête SQL est exécutée lors de `OPEN`...
- Il existe aussi les attributs `%FOUND`, `%ISOPEN`, `%NOTFOUND`, et `%ROWCOUNT` (le nombre de lignes déjà retrouvés par *FETCH*).
- Exemple: `EXIT WHEN (empCursor%ROWCOUNT > 5) OR (empCursor%NOTFOUND).`

Même exemple avec WHILE ...

Usage se %FOUND, pour boucler. Mêmes déclarations.

```
BEGIN
   OPEN empCursor;
   FETCH empCursor INTO   employeeRec;

   WHILE  empCursor%FOUND LOOP

             /* traitement du tuple */
            IF maxSal < employeeRec.salary THEN
                   maxSal := employeeRec.salary;
            END IF;
             /* fin traitement tuple */

            FETCH empCursor INTO   employeeRec;
   END LOOP ;

   dbms_output.putline('Salaire Maximum: '|| maxsal);
   CLOSE empCursor;
END;
```

Boucle pour chaque: *FOR ... in...*

- Une autre forme plus simple de parcourir un *cursor*: ***itérateur abstrait***.
- Syntaxe: ***FOR*** *variableTuple IN CURSOR* ***LOOP*** *...* ***END LOOP***
- Exemple:

```
BEGIN
      FOR employeeRec IN  empCursor LOOP
                /* traitement du tuple */
 IF   maxSal < employeeRec.salary THEN
                   maxSal := employeeRec.salary;
           END IF;
          /* fin traitement tuple */
        END LOOP ;

      dbms_output.putline('Salaire Maximum: '|| maxsal);
END;
```

- La boucle est exécutée **pour chaque** tuple dans le *cursor*
  - La variable de contrôle `employeeRec` est implicitement déclarée du type du cursor.
  - Cette boucle exécute automatiquement un `FETCH` à chaque itération (A chaque itération, un seul tuple est retrouvé.)
  - Ce `FOR` exécute aussi automatiquement un `OPEN` avant d'entrer en boucle et un `CLOSE` en fin de boucle.
  - La boucle se termine automatiquement (sans `EXIT`) dès qu'aucun tuple n'est trouvé.

Variante de boucle `FOR ...in...`

- Forme plus abstraite.
- Requête directe, CURSOR implicite

```
BEGIN
  FOR untel IN (SELECT * FROM employee )
  LOOP
      dbms_output.putline('Num  = ' || untel.enum ||
                          ', Nom = ' || untel.ename ||
                     ', Salaire = '|| untel.salary);
  END LOOP;
END;
```

- A chaque itération, un tuple est retrouvé. La variable `untel`, implicitement déclarée, reçoit le résultat accessible par `untel.enum` *etc*.
- Cas d'expression `SELECT` calculée.

```
FOR salVar IN  (SELECT salary * 1.07 nouveau FROM employee) LOOP
     /*    ...       */
END LOOP ;
```

- *nouveau* est un alias pour l'expression calculée du `SELECT`. Résultat accessible par `salVar.nouveau`.

Cursor donné lors de `OPEN`.

Usage du mot clé `TYPE` avec `REF CURSOR`.

```
Declare
    -- le curseur
       TYPE EmpCurTyp IS REF CURSOR;
       empCur   EmpCurTyp;
    -- les variables
       nom  employee.ename%TYPE ;
Begin
      OPEN empCur FOR 'SELECT ename FROM employee' ;
      FETCH empCur INTO nom ;     -- On FETCH le premier...
          dbms_output.putline( nom ) ;
      CLOSE empCur;
End ;
```

Cursor avec mise à jour

- Les commandes SQL *update* et *delete* peuvent être utilisés avec un CURSOR (déclaré avec la clause *with update of*)
- Elles affectent alors seulement le tuple courant de *FETCH*.
- La clause *WHERE current of cursor*, est alors ajoutée à la commande

```
DECLARE CURSOR   empCur is SELECT Salary FROM employee
    WHERE  DEPT = 'D1' FOR UPDATE OF salary;
BEGIN
     FOR empRec IN  empCur LOOP
         UPDATE employee
         SET salary = empRec.salary * 1.05
                      WHERE  current of empCur ;
      END LOOP ;
      COMMIT;
END;
```

- pour augmenter de 5% les salaires des employés du département D1

Traitement d'Exceptions

- Une erreur ou avertissement PL+SQL peut survenir en cours d'exécution et soulever une exception.
- Une exception peut être predéfinie par le système ou déclarée par l'utilisateur.
- Le traitement d'une exception se fait par la règle *WHEN*

```
WHEN <nom d'exception> THEN   <séquence  d'instructions>;
```

- où la séquence d'instructions est exécuté quand l'exception donnée est soulevée.
- Exemple: requête SELECT monotuple avec
  si un seul tuple retrouvé OK, sinon exception.

```
DECLARE
    employeeRec employee%ROWTYPE;
BEGIN
-- chercher les noms et salaires d'employés d'un département donné

    SELECT ename, salary INTO    employeeRec.ename, employeeRec.Salary
    FROM EMPLOYEE
    WHERE   dept = '&dnum';   -- à lire avant --
    dbms_output.putline(employeeRec.ename||' '||employeeRec.Salary);

EXCEPTION
   WHEN TOO_MANY_ROWS THEN
       dbms_output.putline('Trop d employés.');
END;
```

- la variable SQLPlus *&dnum* est sensée contenir un numéro de département, à lire au préalable.
- Il y a risque que plusieurs employés travailent dans ce département. D'où le cas d'exception à prévoir.

Traitement d'Exceptions (suite)

- Les exeptions systèmes sont automatiquement soulevées lors de l'apparition de l'erreur ou de l'avertissement correspondant.
- Exemples d'exceptions système.

```
CURSOR_ALREADY_OPEN: tentative d'ouverture de CURSOR   déjà ouvert
INVALID_CURSOR: par exemple FETCH sur un CURSOR   déjà fermé
NO_DATA_FOUND: aucun tuple retourné (SELECT INTO   ou FETCH)
TOO_MANY_ROWS: SELECT INTO   retourne plus d'un tuple
...
ZERO_DIVIDE: tentative de division par zéro
```

- Exemple d'usage

```
WHEN NO DATA FOUND THEN   rollback;
```

- Les exception utilisateurs sont soulevées par *raise*

```
RAISE <nom d'exception>
```

Exemple complet

- Augmenter de 5% les salaires des employés du département '123' sans toutefois dépasser 4000 (rajouter l'employé dans une table RICHE)

```
DECLARE
     sal employee.salary%TYPE;
     num employee.enum%TYPE;
     tropGrand exception;
 CURSOR   empCurseur is SELECT enum , salary
                        FROM EMPLOYEE
                        WHERE  DEPT = '&dept'
                        FOR UPDATE OF SALary;
BEGIN
    OPEN empCurseur;
    LOOP
        FETCH empCurseur INTO   num, sal;
        EXIT WHEN empCurseur%NOTFOUND;
        IF sal * 1.05 > 4000 THEN RAISE tropGrand;
        ELSE
             UPDATE EMPLOYEE SAT salary = sal * 1.05
             WHERE  CURRENT OF empCurseur ;
             dbms_output.put_line (num || ' mis a jour.');
        END IF   ;
      END LOOP ;
EXCEPTION
      WHEN NO_DATA_FOUND THEN
            dbms_output.putline('pas trouve');
            rollback;
      WHEN tropgrand THEN
            INSERT INTO   veterans VALUES (num, sal);
            dbms_output.putline(num||' '||Sal || ' Trop grand');
      COMMIT;
END;
```

- Les traitements d'exceptions sont définis à la fin du bloc instructions par la clause `exception`.
- La variable `tropGrand` est déclarée de type `exception`, pour être utilisée dans `raise`.

En savoir plus

- http://en.wikipedia.org/wiki/PL/SQL
- www.mathcs.emory.edu/~cheung/Courses/377/Others/tutorial.pdf
- http://www.plsql-tutorial.com/
- http://www.plsqltutorial.com/
- http://www.java2s.com/Tutorial/Oracle/CatalogOracle.htm
- http://infolab.stanford.edu/~ullman/fcdb/oracle/or-plsql.html (1998)