Bases de Données
2ème année

Rosine CICCHETI, Lotfi LAKHAL, Sebastien NEDJAR

Rosine CICCHETI
Lotfi LAKHAL
Sebastien NEDJAR

Bases de données
2ème année

Une application doit utiliser une base de données si :
— il y a un volume de données conséquent
— les données doivent être soumises à de nombreuses contraintes d'intégrité
— il y a plusieurs utilisateurs
— il y a besoin de la notion de transaction

Le langage PL/SQL (Procedural Language/Structured Query Language) est un langage hôte qui accueille des instructions SQL.

Première partie

Structure d'un bloc PL/SQL

Il existe différents types de blocs :
— procédures
— fonctions
— packages
— blocs anonymes

Un programme PL/SQL a la structure suivante :

```
DECLARE
       <liste_declarations>
BEGIN
       <liste_instructions>
EXCEPTION
       <gestion_exceptions>
END;
```

Rosine CICCHETI
Lotfi LAKHAL
Sebastien NEDJAR

Deuxième partie

Déclarations

On déclare des variables, des types, des exceptions développeurs et des curseurs.

1 Variables scalaires

Ce sont des variables simples dont le type est un des types proposés par Oracle : *VARCHAR2*(n), *DATE*, *NUMBER*(m,n) [1], *CHAR*(n). On a aussi le type *BOOLEAN* qui peut prendre trois valeurs : *TRUE*, *FALSE* et *NULL*.

Une variable scalaire est déclarée ainsi : `<nomvariable> <nomtype> [NOT NULL] [DEFAULT <valeur_default>]`

Au démarrage d'un programme, toutes les variables sont initialisées à *NULL*, sauf celles qui ont une valeur par défaut. Les valeurs par défaut peuvent être des constantes, le résultat de calculs ou de fonction de calcul horizontal SQL (*UPPER*, *LOWER*, *LENGTH*,...), ou le contenu d'une autre variable. On peut aussi faire des déclarations par référence : déclarer une variable en lui donnant le même type qu'une autre variable ou qu'un attribut de la base.

Exemple

```
effectif NUMBER(3,0); -- Nombre entier
diplome  VARCHAR2(30) DEFAULT 'DUT'; -- Chaine de caractère ayant 'DUT' comme valeur par défaut
nb       NUMBER(4,2)  NOT NULL DEFAULT 10; -- Nombre décimal ayant 10 comme valeur par défaut
nb1      NUMBER(4,2)  DEFAULT nb; -- Nombre décimal prenant la valeur de nb
nb2      nb%TYPE      DEFAULT nb*0.5; -- Variable du même type que nb, prenant comme valeur nb*0.5
date_t   DATE         DEFAULT SYSDATE; -- Date, prenant comme valeur la date du jour (SYSDATE)
```

Exemple de déclaration par référence

```
nb    NUMBER(4,2) DEFAULT 10;
nb1   nb%TYPE;
nom   ETUDIANT.NOM_ET TYPE;
ville ETUDIANT.VILLE%TYPE;
```

Les déclarations par référence permettent une cohérence des déclarations des variables comparables, et une réduction de la maintenance des applications. Une variable déclarée par référence « hérite » de la clause *NOT NULL*, mais pas de la valeur par défaut.

2 Variables composées

On peut utiliser des enregistrements ou des tableaux à une dimension.

2.1 Les enregistrements

Déclaration d'un type enregistrement

```
TYPE <nom_type> IS RECORD (<nom_champ1> <type1> [NOT NULL] [DEFAULT <valeur_default1>]
                           [, <nom_champ2> <type2> [NOT NULL] [DEFAULT <valeur_default2>]]
<nom_variable> <nom_type>
```

1. m chiffres, dont n décimales ; par exemple 545.27 est un *NUMBER*(5,2)

On peut imbriquer les enregistrements.

Pour manipuler les champs des variables enregistrement, on utilise `<nom_variable>.<nom_champ>`. Pour les variables enregistrement simples (non imbriquées), on peut faire des déclarations par référence avec `<nom_variable> <nom_type>%ROWTYPE`, par exemple `<un_etudiant> ETUDIANT%ROWTYPE`

2.2 Les variables sculptures

Les tableaux sont à une dimension et les élements sont scalaires.

Déclaration d'un type tableau

```
TYPE <nom_type_tableau> IS TABLE OF <nom_type | nom_var%TYPE>
     INDEX BY BINARY_INTEGER ;
<nom_variable> <nom_type_tableau>
```

Pour manipuler les élements du tableau, on utilise `<nom_variable_tableau>(index)`.

En PL/SQL, les tableaux sont non denses (indices non consécutifs) et non bornés (taille dynamique).

On dispose des primitives suivantes :

— `<nom_variable>.EXISTS(n)` renvoie vrai s'il existe un élément d'ordre n, faux sinon
— `<nom_variable>.COUNT` renvoie le nombre d'éléments existant dans le tableau
—
— `<nom_variable>.FIRST` renvoie l'indice du premier élément du tableau
—
— `<nom_variable>.LAST` renvoie l'indice du dernier élément du tableau

`tab.PRIOR(tab.FIRST)` et `tab.NEXT(tab.LAST)` renvoient *NULL*.

2.3 Constantes

`<nom_constante> CONSTANT <nom_type>:=<valeur>`

2.4 Exception

Seules les exceptions utilisateurs sont déclarées avec `<nom_exception> EXCEPTION`.

3 Instructions

3.1 Affectation

3.1.1 Affectation classique

On utilise := pour donner une valeur à une variable.

Exemple On suppose qu'on a déclaré les variables et types nécessaires.

```
nb:=100;
nb1:=nb/2;
adressse.ville:='Marseille';
notes(3):=20;
```

3.1.2 Affectation par requêtes

Pour cette affectation, la requête doit rendre au plus un résultat.

Exemple

```
DECLARE
       Effectif NUMBER(3,0);
       un_etudiant ETUDIANT%ROWTYPE;
BEGIN
      SELECT COUNT(*) INTO Effectif;
      FROM   ETUDIANT;

      SELECT * INTO un_etudiant
      FROM   ETUDIANT
      WHERE  NUM_ET=210;END
```

La première affectation ne peut pas déclencher d'exception système (`Effectif=0` si la relation `ETUDIANT` est vide). La seconde peut déclencher l'exception système `NO_DATA_FOUND`. On peut spécifier plusieurs variables de réception pour une affectation.

3.2 Instructions conditionnelles

```
IF <condition> THEN
  [BEGIN]
  <liste_instructions>
  [END]
ELSIF <condition2> THEN
     <liste_instructions>
ELSE
    <liste_instructions>
END IF
```

`<condition>` et `<condition2>` peuvent être des conditions liées par `AND`, `OR`, `NOT`, et on peut utiliser les prédicats SQL ou leur forme négative.

3.3 Itérations

3.3.1 Boucle *FOR*

```
FOR <variable_de_parcours> IN <borne_inf>..<borne_sup>
BEGIN
   <liste_instructions>
END LOOP;
```

`<variable_de_parcours>` ne doit pas être initialisée.

3.3.2 Boucle *WHILE*

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

Exemple : parcours d'un tableau

```
numero := Notes.FIRST
WHILE  numero IS NOT NULL LOOP
       <liste_instructions>
       numero:=Notes.NEXT(numero)
END LOOP;
```

3.3.3 Boucles répeter/jusqu'à (*EXIT WHEN*

```
LOOP
    <instructions>
    EXIT WHEN <condition_sortie>
END LOOP;
```

Exemple : parcours d'un tableau

```
numero:=NOTES.FIRST;
LOOP
    <instructions>
    numero:=Notes.NEXT(numero);
    EXIT WHEN numero IS NULL
END LOOP;
```

3.4 Autres instructions

— pour lever une exception utilisateur déclarée : *RAISE* <nom_exception>
— pour « ne rien faire » (terminer le programme proprement) : *NULL*

4 Les curseurs

Dans l'univers des bases de données, on manipule les tuples sous forme d'ensemble. Dans l'univers de la programmation, on manipule les tuples enregistrement par enregistrement. Pour lier les deux, on utilise des curseurs. On peut voir un curseur comme le résultat d'une requête. On le déclare par *CURSOR* <nom_curseur> *IS* <requete>;

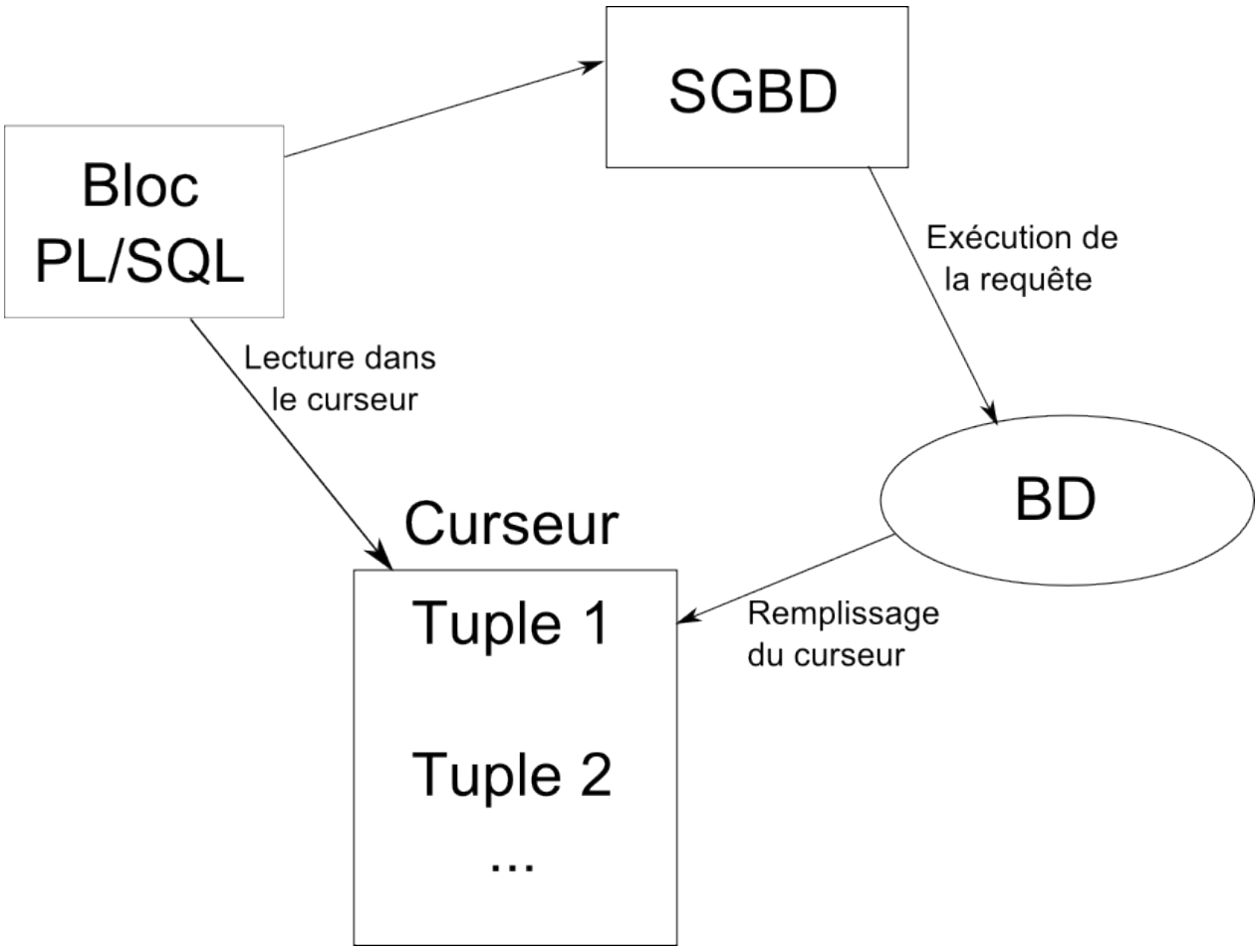

FIGURE 1 – Représentation d'un curseur

Exemple On a besoin de récupérer, dans un bloc PL/SQL, la liste des étudiants de 2ème année. Dans la base, on a la relation ETUDIANT(NUM_ET, NOM_ET, PRENOM_ET, ..., ANNEE).

```
DECLARE
        CURSOR Et2 IN SELECT NUM_ET, PRENOM_ET
                      FROM ETUDIANT
                      WHERE ANNEE=2
                      ORDER BY NOM_ET, PRENOM_ET
```

4.1 Ordres de déclaration des curseurs

— *OPEN* <nom_curseur>;
La requête de définition du curseur est exécutée et le curseur est rempli.
— *CLOSE* «nom_curseur>;
La zone mémoire nécessaire est liberée.
— *FETCH* <nom_curseur> *INTO* <nom_variable>;
Retourne un enregistrement dans <nom_variable>.

<nom_variable> peut être :
— une liste de variables scalaires
— une variable enregistrement qui peut avoir la même structure que le curseur

Exemple : on déclare Et2

```
nom ETUDIANT_NOM%TYPE;
prenom ETUDIANT_PRENOM%TYPE;
-- OU
etudiant Et2%ROWTYPE;
FETCH Et2 INTO nom, prenom;
-- OU
FETCH Et2 INTO etudiant;

nom:=UPPER(nom)
prenom:=PRENOM(prenom)
-- OU
etudiant.NOM_ET:=UPPER(etudiant.NOM_ET)
```

4.2 Propriétés des curseurs

- <nom_curseur>%*FOUND* (ou *NOTFOUND*) rend *TRUE* si le dernier *FETCH* a renvoyé un résultat
- <nom_curseur>%*ISOPEN* rend vrai si le curseur est ouvert
- <nom_curseur>%*ROWCOUNT* rend le nombre d'enregistrements retournées par les *FETCH*. Cette propriété vaut 0 à l'ouverture du curseur.

Exemple : on utilise le curseur Et2 et la variable enregistrement

```
BEGIN
     OPEN  Et2;
     FETCH Et2 INTO etudiant;
     WHILE Et2%FOUND LOOP
          <instructions>
          FETCH Et2 INTO etudiant;
     END LOOP;
     CLOSE Et2;
END;
```

Exemple : avec une boucle *EXIT WHEN*

```
BEGIN
     OPEN  Et2;
     LOOP
          FETCH Et2 INTO etudiant;
          <instructions>
          EXIT WHEN Et2%NOTFOUND;
     END LOOP;
     CLOSE Et2;
END;
```

Exemple Dans une base, on a la relation ETUDIANT(NUM_ET,...,DEPT#). On vient de créer la relation EFF_DEPT(NUM_DEPT,EFF) qui est vide. On va écrire un bloc permettant de remplir cette relation à partir des données de ETUDIANT.

```
DECLARE
       CURSOR E_D IS SELECT COUNT(*)
              FROM   ETUDIANT
              GROUP BY DEPT;
       E      ED%ROWTYPE;
BEGIN
       OPEN E_D;
       FETCH E_D INTO E;
       WHILE E_D%FOUND LOOP
              INSERT INTO  EFF_DEPT(NUM_DEPT,EFF)
                           VALUES(E.DEPT, E.EFF);
              FETCH E_D INTO E;
       END LOOP;
       CLOSE E_D;END;
```

On suppose qu'on a EFF_DEPT(NUM_DEPT, EFF, EFF_AN1, EFF_AN2). Ces effectifs peuvent être calculés à partir de la relation *ETUDIANT*.

```
DECLARE
       CURSOR  ET1_D IS SELECT DEPT, COUNT(*) F1
              FROM      ETUDIANT
              WHERE     ANNEE=1
              GROUP BY DEPT;
       E1     ET1_D%ROWTYPE
BEGIN
       OPEN ET1_D;
       FETCH ET1_D INTO E1;
       WHILE  ET1_D%FOUND LOOP
              UPDATE EFF_DEPT
                     SET        EFF_AN1=E1.F1
                     WHERE      NUMDEP=E1.DEPT;
              FETCH ET1_D INTO E1;
       END LOOP;CLOSE ET1_D;END;
```

4.3 Curseurs paramétrés

On peut paramétrer les constantes de sélection d'un curseur.

Exemple

```
CURSOR EFF_AN(An ETUDIANT, AN%TYPE) IS SELECT DEPT, COUNT(*) Nb
       FROM                          ETUDIANT
       WHERE                         ANNEE=An
       GROUP BY DEPT;
E      EFF_AN%ROWTYPE;
OPEN   EF_AN(1);
```

4.4 Parcours automatique d'un curseur

Les particularités du parcours automatique sont qu'il n'y a pas de variable de parcours, pas de *OPEN/CLOSE*, et pas de *FETCH*.

```
FOR <nom_variable> IN <nom_curseur>
LOOP
     <instructions>
END LOOP;
```

Exemple

```
DECLARE
       CURSOR CM IS
BEGIN
       FOR vCM IN CM LOOP
           <instructions>
       END LOOP;
END
```

5 Les exceptions

Elles sont utilisateur ou système, anonymes ou nomées.

5.1 Exceptions système nommées

Une dizaine d'exception est nommée par exemple : *NO_DATA_FOUND*, *TOO_MANY_ROWS*, *ZERO_DIVIDE*, *INVALID_CURSOR*, ...

5.2 Exceptions système anonymes

C'est le cas de la majorité des exceptions Oracle. Elles ont un code (négatif). Les fonctions *sqlcode* et *sqlerm* renvoient respectivement le code et le message de l'erreur.

5.3 Exceptions utilisateur anonymes

Elles sont réservées aux triggers et blocs stockés. On ne les déclare pas dans *DECLARE*. On les déclenche avec :

```
IF <condition> THEN RAISE_APPLICATION_ERROR(<code>, <message>); END IF;
```

<code> est dans l'intervalle $[-20999; -20000]$.

5.4 Traitement des exceptions

Toutes les exceptions, sauf les exceptions utilisateur anonymes, doivent être traitées dans la partie *EXCEPTION*.

```
EXCEPTION
     WHEN <nom_exception> THEN <traitement>;
     WHEN <exception1> OR <exception2> THEN <traitement>;
     WHEN OTHERS THEN
          IF sqlcode = <code> THEN
          END IF;
```

6 Les différents types de blocs

Les blocs, procédures, fonctions et blocs anonymes peuvent être imbriquéess les uns das les autre.

6.1 Procédures

```
PROCEDURE <nom_procedure> (<nom_parametre1> <mode_parametre1>  <type_parametre1>
                          [,<nom_parametre2> <mode_parametre2> <type_parametre2>...]) IS
         <declarations_locales>
         BEGIN
              <instructions>
END <nom_procedure>;
```

<mode_parametre> peut être *IN*, *OUT* ou *IN OUT*.

Exemple : fonction qui permet de formatter le nom et prénom

```
PROCEDURE  Formatter (nom IN OUT ETUDIANT.NOM_ET%ROWTYPE,
                      prenom IN OUT ETUDIANT%ROWTYPE) IS
          nom:=UPPER(nom);
          prenom:=INITCAP(prenom);
END Formatter;
```

6.2 Fonction

```
FUNCTION <nom_fonction> (<nom_parametre1> <mode_parametre1> <type_paramatre1>
                         [<nom_parametre2> <mode_parametre2> <type_parametre2>...])
                         RETURN <type_retour> IS
         <instructions>
         RETURN(<valeur_retour>);
END <nom_fonction>;
```

6.3 Procédures stockées et fonctions stockées

La procédure ou fonction est stockée comme un objet Oracle et décrit dans des tables système (jusqu'à son code). On peut alors l'appeler depuis n'importe quel autre bloc.

```
CREATE[OR REPLACE] PROCEDURE | FUNCTION <nom_bloc>(<parametres>) [RETURN <type_retour>] IS ...
      <instructions>
END <nom_bloc>;
```

6.4 Blocs anonymes

FIGURE 2 – Portée des variables

Exemple de l'intérêt des blocs imbriqués On suppose qu'on doit faire un *INSERT* dans une relation 1, puis dans une relation 2.

Hypothèse :le 1er *INSERT* lève une exception ; toutes les instructions suivantes (dont le 2ème *INSERT*) ne sont pas évaluées. En utilisant des blocs imbriquées, on peut poursuivre l'exécution.

```
DECLARE
       <declarations>
BEGIN
     <instructions>
     DECLARE
            <declarations>
     BEGIN
          <instructions>
          INSERT ...
     EXCEPTION
     END
     INSERT ...
END
```

7 Les triggers (déclencheurs)

Les triggers sont aussi appelés « règles ECA » (Évènement/Condition/Action).

E Quand un évènement survient sur la base de données
C Si une condition est vérifiée
A Alors l'action est exécutée

Intérêts

— le même trigger peut être déclenché par plusieurs programmes, ce qui rend le code plus modulaire
— on peut automatiquement déclencher la vérification des contraintes dynamiques, des actions ou des alertes (seuil atteint,...)
— permet de propager automatiquement une mise à jour des données

7.1 Partie Évènement

Un évènement est la détection par le système d'un ordre SQL (à l'exception de *SELECT*) par un utilisateur (humain ou programme).

Les ordres SQL peuvent être des ordres :

— du LMD : *INSERT, DELETE, UPDATE*
— du LDD : *CREATE, ALTER*
— du LCD : *GRANT, REVOQUE*

7.2 Chronologie d'un trigger

Détection de l'ordre SQL déclencheur du trigger	Exécution du trigger de type *BEFORE*	Exécution de l'ordre SQL	Exécution du trigger de type *AFTER*
|t1	|t2	|t3	|t4

FIGURE 3 – Chronologie d'un trigger

Remarque

- un trigger permettant le contrôle ou la mise en forme de données est forcément de type *BEFORE* car il doit vérifier/formatter les données avant leur insertion
- un trigger effectuant la propagation d'une mise à jour est forcément de type *AFTER* car il faut être sûr que l'ordre SQL déclencheur a bien été exécuté

7.3 Granularité des triggers LMD

Un trigger LMD peut être orienté ensemble (exécuté une seule fois pour tous les tuples concernés par l'ordre SQL déclencheur), ou orienté tuple (exécuté pour chaque tuple concerné).

		$\Downarrow n+1 \Leftarrow$	
Ordre SQL LMD de l'utilisateur	Lecture des tuples concernés par l'ordre SQL	Exécution du trigger de type *BEFORE* pour le tuple n	Exécution de l'ordre SQL pour le tuple n
|t1	|t2	|t3	|t4

FIGURE 4 – Chronologie d'un trigger orienté tuple de type *BEFORE*

Rosine CICCHETI
Lotfi LAKHAL
Sebastien NEDJAR

Index

Table des matières

Liste des tableaux

Table des figures

Liste des mots-clefs PL/SQL