M2106

Programmation et administration des bases de données

COURS 2 :

PL/SQL

Février 14, 2017

BELFODIL Adnene – BENDIMERAD Anes

* Cours extraits principalement des cours de Guillaume Cabanac, Fouad Dahak, Yacine Bouhabel, Amrouche Karima, Richard Grin

SOMMAIRe

- Introduction
- Éléments de base
- Curseurs
- Exceptions
- Procédures et fonction
- Déclencheurs (Triggers)

SOMMAIRe

Pourquoi PL/SQL

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

- Besoin important d'un langage procédural pour lier plusieurs requêtes SQL avec des variables et dans des structures de programmation habituelles.

1. Introduction

Définition – PL/SQL (Procédural Language SQL)

PL/SQL est une extension SQL, c'est un langage procédurale qui inclut des éléments habituels de programmation (e.g., conditions et boucles, procédures et fonctions).

PL/SQL est un langage spécifique à Oracle. Il existe d'autres langages pour les autres SGBD (e.g., TransacSQL pour SQLServer).

Blocs PL/SQL

Un programme est structuré en blocs d'instructions de 3 types :

- Blocs anonymes
- Procédures nommées
- Fonctions nommées

Un bloc peut contenir d'autres blocs, et il doit contenir en moins une instruction (il peut donc contenir l'instruction null)

Structure d'un bloc

```
DECLARE
      -- définition des variables
BEGIN
      -- code du programme (instruction SQL ou PL/SQL)
EXCEPTION
      -- code de gestion des erreurs
END;
```

SOMMAIRe

- Introduction
- Éléments de base
- Curseurs
- Exceptions
- Procédures et fonction
- Déclencheurs (Triggers)

2.1. Déclaration

Structure d'une déclaration

```
nom-variable type-variable [ := valeur ] ;
```

Exemples

```
age number(10);
nom varchar(30) := 'Dupont';
date_naissance date;
ok boolean := true;
```

2.1. Déclaration

L'identificateur `nom-variable` respecte les contraintes suivantes:

- 30 caractères au plus
- Il doit commencer par une lettre
- Il peut contenir lettres, chiffres, _, $, et #
- Il n'est pas sensible à la casse.
- Portée habituelle des langages à blocs
- Il doit être déclaré avant d'être utilisé

2.1. Déclaration

`type-variable` peut être :

- Type simple SQL : `VARCHAR2, NUMBER, DATE, BOOLEAN`
- `%TYPE` : Type d'une colonne d'une table. Exemple :

```
v_nom employe.nom%TYPE
```

- `%ROWTYPE` : Type d'une ligne entière d'une table. Exemple :

```
v_employe employe%ROWTYPE
```

2.2.1 Commandes principales - affectations

Affectation d'une valeur à une variable

- Opérateur d'affectation « := » . Exemple :

```
v_nom := 'Rogers';
```

- Directive INTO de la requête SELECT. Exemple :

```
SELECT nom INTO v_nom FROM employe where dept=50;
```

→ Cette commande récupère le résultat du `SELECT` et l'affecte à la variable `v_nom`

2.2.1 Commandes principales - affectations

Exemple d'utilisation

```
DECLARE
      v_emp emp%ROWTYPE;
BEGIN
      SELECT * INTO v_emp FROM emp WHERE dept=50;
      v_emp.dept=20;
      INSERT INTO emp VALUES v_emp;
END;
```

2.2.2 Commandes principales - conditions

Syntaxe IF-THEN

```
IF condition THEN
   instructions…
END IF;
```

Syntaxe IF-THEN-ELSIF

```
IF condition_1 THEN
   instructions…
ELSIF condition_2 THEN
   instructions…
END IF;
```

Syntaxe IF-THEN-ELSE

```
IF condition THEN
   instructions…
ELSE
   instructions…
END IF;
```

Syntaxe CASE

```
CASE selecteur
   WHEN expression1 THEN resultat1
   WHEN expression2 THEN resultat2
   ELSE resultat3
END;
```

2.2.2 Commandes principales - conditions

Exemple IF-THEN

```
IF v_date>'11-APR-03' THEN
   v_salaire:= v_salaire*1.15;
END IF;
```

Exemple IF-THEN-ELSIF

```
IF v_nom='Paker' THEN
 v_salaire:= v_salaire*1.15;
ELSIFv_nom='ASTROFF'
 v_salaire:= v_salaire*1.05;
END IF;
```

Exemple IF-THEN-ELSE

```
IF v_date>'11-APR-03' THEN
   v_salaire:= v_salaire*1.15;
ELSE
   v_salaire:= v_salaire*1.05;
END IF;
```

Exemple CASE

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

2.2.2 Commandes principales - boucles

Syntaxe LOOP

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

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

Syntaxe WHILE

```
WHILE condition LOOP
   instructions…
END LOOP;
```

Tant que `condition` est vraie, les `instructions` sont répétées

Syntaxe FOR

```
FOR variable IN debut..fin  LOOP
    instructions…
END LOOP;
```

`variable` prend les valeurs de `debut`, `debut+1`,…, jusqu'à `fin`. Ne pas déclarer `variable`, elle est déclarée implicitement.

2.2.2 Commandes principales - boucles

Exemple LOOP

```
v_cpt:= 1;
LOOP
   update emp set salaire:=salaire*2
   where emp_id=v_cpt;
   v_cpt:= v_cpt+1;
   EXIT WHEN v_cpt>10;
END LOOP
```

Exemple WHILE

```
v_cpt:= 1;
WHILE v_cpt<=10 LOOP
   update emp set salaire:=salaire*2
   where emp_id=v_cpt;
   v_cpt:= v_cpt+1;
END LOOP
```

Exemple FOR

```
FOR v_cpt in 1..10 LOOP
   update emp set salaire:=salaire*2
   where emp_id=v_cpt;
END LOOP
```

2.2.2 Commandes principales - affichage

Syntaxe d'affichage

```
DBMS_OUTPUT.PUT_LINE(chaine);
```

Remarques

- Avant d'utiliser cette instruction, il faut initialiser le buffer de sortie : `SET SERVEROUTPUT ON SIZE buffersize.`
- Utiliser || pour faire des concaténations.

Exemple d'utilisation

```
DECLARE
   v_nom varchar2(25);
BEGIN
   SELECT nom INTO v_nom FROM emp
   WHERE id=50;
   DBMS_OUTPUT.PUT_LINE('le nom est
   : ' || v_nom);
END;
```

2.2.2 Commandes principales - commentaires

Syntaxe d'un commentaire

```
-- Pour une fin de ligne
/* Pour plusieurs lignes */
```

SOMMAIRe

Pourquoi les curseurs

En PL/SQL, on a parfois besoin de récupérer plusieurs lignes correspondant à une contrainte particulière. Cependant, la directive `SELECT … INTO` permet de récupérer une seule ligne à la fois.

- Besoin d'un outil particulier permettant la manipulation des requêtes correspondant à plusieurs lignes au même temps : Les curseurs.

Définition des curseurs

Un curseur est un pointeur vers la zone de mémoire privée allouée par le serveur Oracle. Il est utilisé pour gérer l'ensemble des résultats d'une requête. Il existe deux types :

- Implicite : créé et géré en interne par le serveur afin de traiter les instructions SQL.
- Explicite : déclaré explicitement par le programmeur.

Attributs d'un curseurs

Tous les curseurs ont des attributs que le programmeur peut utiliser :

- `%ROWCOUNT` : nombre de lignes traitées par le curseurs.
- `%FOUND` : vrai si au moins une ligne a été traitée par la requête.
- `%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).

3.1. Curseurs implicites

Les curseurs implicites sont tous nommés SQL. Exemple :

```
DECLARE
   nb_lignes NUMBER(10);
BEGIN
   DELETE FROM emp WHERE dept=10;
   nb_lignes := SQL%ROWCOUNT;
END;
```

3.2. Curseurs explicites

Les curseurs explicites servent à traiter les SELECT qui renvoient plusieurs lignes. Ces curseurs sont utilisés en 4 étapes :

1. Déclaration (avec les autres variables dans la zone DECLARE)
2. Ouverture (OPEN)
3. Avancement ligne par ligne (FETCH)
4. Fermeture (CLOSE)

3.2. Curseurs explicites

Syntaxe de déclaration

```
CURSOR nom_du_cuseur IS
        un énoncé SELECT;
```

Remarques

Ne pas inclure la clause INTO dans la déclaration du curseur.

Exemple de déclaration

```
DECLARE
   CURSOR c1 IS
      SELECT ref, nom, qte
      FROM Article
      WHERE qte<500;
BEGIN
   …
END;
```

3.2. Curseurs explicites

Syntaxe d'OPEN

```
OPEN nom_du_curseur;
```

Syntaxe de FETCH

```
FETCH nom_du_curseur
   INTO [variable1, [variable2,…];
```

Remarques

- La commande `FETCH` recherche les informations de la ligne en cours et les met dans les variables : `variable1`, `variable2`…
- On peut utiliser les attributs des curseurs (e.g., `%NOTFOUND`) pour tester le résultat du `FETCH`

3.2. Curseurs explicites

Syntaxe de CLOSE

```
CLOSE nom_du_curseur;
```

Remarques

- Le curseur doit être fermé après la fin du traitement des lignes. Il peut être rouvert si nécessaire.
- On ne peut pas rechercher des informations dans un curseur si ce dernier est fermé.

3.2. Curseurs explicites

Exemple avec des variables définies par %TYPE

```
DECLARE
   CURSOR c_emp IS SELECT id, nom FROM emp where dept=30;
    v_id emp.id%TYPE;
   v_nom emp.nom%TYPE;
BEGIN
   OPEN c_emp;
   LOOP
      FETCH c_emp into v_id, v_nom;
      EXIT WHEN c_emp%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(v_id || ‘ ‘ || v_nom);
   END LOOP;
   CLOSE c_emp;
END;
```

3.2. Curseurs explicites

Exemple avec des variables définies par %ROWTYPE

```
DECLARE
   CURSOR c_emp IS SELECT id, nom FROM emp where dept=30;
    v_emp c_emp%ROWTYPE;
BEGIN
   OPEN c_emp;
   LOOP
      FETCH c_emp into v_emp;
      EXIT WHEN c_emp%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(v_emp.v_id || ‘ ‘ || v_emp.v_nom);
   END LOOP;
   CLOSE c_emp;
END;
```

3.2. Curseurs explicites

Utilisation de la boucle FOR

- La boucle FOR simplifie la programmation car elle évite d'utiliser explicitement les instructions OPEN, FETCH, CLOSE.
- En plus, elle déclare implicitement une variable de type ROW associé au curseur (ci-dessous, le type de 'variable' est curs%ROWTYPE).

Syntaxe de la boucle FOR

```
FOR variable IN curs LOOP
   …
END LOOP;
```

Exemple avec une boucle FOR

```
DECLARE
   CURSOR c_client IS
      SELECT nom, adresse FROM clients;
BEGIN
   FOR v_client in c_client LOOP
      DBMS_OUTPUT.PUT_LINE('Nom : '||UPPER(c_client.nom) || ' Ville :' ||
      c_client.addresse);
   END LOOP;
END;
```

3.2. Curseurs explicites

Les curseurs paramétrés

- Un curseur paramétré peut servir plusieurs fois avec des valeurs de paramètres différentes.
- On doit fermer le curseur entre chaque utilisation de paramètres différents (sauf si on utilise la boucle `FOR`, `car elle` ferme le curseur automatiquement).

3.2. Curseurs explicites

Exemple avec un curseur paramétré

```
DECLARE
   CURSOR c_emp (p_dept NUMBER(10)) IS
      SELECT dept, nom FROM emp where dept = p_dept;
BEGIN
   FOR v_emp in c_emp(10) LOOP
      DBMS_OUTPUT.PUT_LINE('Nom : '||UPPER(emp.nom));
   END LOOP;
   FOR v_emp in c_emp(20) LOOP
      DBMS_OUTPUT.PUT_LINE('Nom : '||UPPER(emp.nom));
   END LOOP;
END;
```

SOMMAIRe

4. Exceptions

Définition

C’est une erreur qui survient durant une exécution. Il en existe 2 types :

- Prédéfinie par Oracle (déclenchée implicitement).
- Définie par le programmeur (déclenchée explicitement).

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.

Syntaxe de la clause d'exception

```
EXCEPTION
  WHEN exception1 [OR exception2…] THEN
    instructions…
  [WHEN exception3 [OR exception4…] THEN
    instructions…]
  [WHEN OTHERS THEN
    instructions…]
```

4.1. Exceptions prédéfinies

Noms de quelques exceptions prédéfinies

- `NO_DATA_FOUND` : quand le `SELECT … INTO` ne retourne aucune ligne.
- `TOO_MANY_ROWS` : quand le SELECT … `INTO` retourne plusieurs ligne.
- `VALUE_ERROR` : erreur numérique (e.g., erreur dans la conversion d'une chaine de caractères à un nombre).
- `ZERO_DEVIDE` : Division par zéro.
- `OTHERS` : Toutes les erreurs non interceptées.

4.1. Exceptions prédéfinies

Exemple

```
DECLARE
   …
BEGIN
   …
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('no data found by the query');
   WHEN TOO_MANY_ROWS THEN
      DBMS_OUTPUT.PUT_LINE('too many rows returned by the query');
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('other exception');
END;
```

4.1. Exceptions prédéfinies

Exemple de génération et de traitement d'une exception

```
DECLARE
   v_nom VARCHAR2(15);
BEGIN
   SELECT nom INTO v_nom FROM emp WHERE prenom='John';
   DBMS_OUTPUT.PUT_LINE('Le nom de John est : ' || v_nom);
EXCEPTION
   WHEN TOO_MANY_ROWS THEN
      DBMS_OUTPUT.PUT_LINE('too many rows returned by the query');
END;
```

4.2. Exceptions utilisateur

Définition

C’est une exception définie explicitement par le programmeur. Son utilisation se fait en 3 étapes :

1. Déclaration : nommer l’exception dans la section `DECLARE`.
2. Activation : déclencher l’exception dans la section exécutable, en utilisant la commande `RAISE`.
3. Traitement : dans la section `EXCEPTION`.

4.2. Exceptions utilisateur

Exemple

```
DECLARE
   v_salaire number(10,2);
   SALAIRE_TROP_BAS EXCEPTION;
BEGIN
   SELECT sal INTO v_salaire FROM emp WHERE dept='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');
END;
```

SOMMAIRe

- Introduction
- Éléments de base
- Curseurs
- Exceptions
- Procédures et fonction
- Déclencheurs (Triggers)

5. Procédures et fonctions

Définition

- Une procédure ou une fonction est un bloc PL/SQL nommé, puis compilé et stocké dans la base.
- Une fonction est identique à une procédure, à la différence qu'elle retourne une valeur.
- Les procédures et les fonctions créées sont stockées dans la table système : `user_source`

5. Procédures et fonctions

Syntaxe d'une procédure

```
CREATE [OR REPLACE] PROCEDURE nom_procedure
   [(parametre1 [mode] type1, parametre2 [mode] type2,…)] IS|AS
   [declaration de variables…]
BEGIN
   instructions…
END [nom_procedure];
```

Remarque

`mode` peut être `IN` (paramètre en entrée) ou `OUT` (paramètre en sortie).

Exemple d'une procédure sans paramètre

```
CREATE OR REPLACE PROCEDURE list_nom_clients
IS
   CURSOR c_nom_clients IS SELECT nom, adresse FROM clients;
BEGIN
   FOR v_client IN c_nom_clients LOOP
      DBMS_OUTPUT.PUT_LINE('nom :'|| UPPER(v_client.nom) ||' ville :'
      v_client.adresse);
   END LOOP;
END;
```

Exemple d'une procédure avec paramètres

```
CREATE OR REPLACE PROCEDURE list_nom_clients
   (ville IN varchar2, result OUT number)
IS
   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;
```

Syntaxe d'une fonction

```
CREATE [OR REPLACE] FUNCTION nom_fonction
   [(parametre1 [mode] type1, parametre2 [mode] type2,…)]
   RETURN type_retour IS|AS
   [declaration de variables…]
BEGIN
   …
   RETURN valeur_retour;
END [nom_fonction];
```

Exemple d'une fonction sans paramètre

```
CREATE OR REPLACE FUNCTION nombre_clients
RETURN NUMBER
IS
   v_nb NUMBER(10);
   CURSOR c_nb_clients IS SELECT count(*) FROM clients;
BEGIN
   OPEN c_nb_clients;
   FETCH c_nb_clients INTO v_nb;
   RETURN v_nb;
END;
```

Exemple d'une fonction avec paramètre

```
CREATE OR REPLACE FUNCTION euro_to_dollar(somme IN NUMBER)
RETURN NUMBER
IS
   taux constant NUMBER(10) := 1.06;
BEGIN
   RETURN somme * taux;
END;
```

Syntaxe de suppression d'une procédure ou une fonction

```
DROP PROCEDURE nom_procedure

DROP FUNCTION nom_fonction
```

5. Procédures et fonctions

Compilation

Afin de compiler une fonction ou une procédure, il faut taper une dernière ligne « / ».

Exécution

On peut exécuter une procédure directement avec :

```
EXECUTE nom_procedure(param…);
```

Utilisation

- Les procédures et les fonctions peuvent être utilisées dans d'autres procédures et fonctions ou dans des blocs PL/SQL anonymes.
- Les fonctions peuvent aussi être utilisées dans des requêtes SQL.

SOMMAIRe

6. Déclencheurs (triggers)

Définition

- C’est un programme PL/SQL qui se déclenche automatiquement suite à un évènement ou au passage à un état spécifié de la base de données. Il peut définir des actions qui ne sont pas modélisables avec les contraintes d’intégrité.
- Dans ce cours, on s’intéresse aux déclencheurs qui s’exécutent avant ou après une opération LMD (`INSERT, UPDATE, DELETE`)

6. Déclencheurs (triggers)

Syntaxe d'un trigger

```
CREATE [OR REPLACE] TRIGGER nom_trigger
   {BEFORE | AFTER} evenement [OF liste_colonnes] ON nom_table
   [FOR EACH ROW [WHEN condition] ]
   corps_trigger
```

A quel moment se déclenche un trigger

- BEFORE : le code dans le corps du trigger s'exécute avant les évènements qui déclenchent le trigger.
- AFTER : il s'exécute après les évènements qui déclenchent le trigger.

6. Déclencheurs (triggers)

Syntaxe d'un trigger

```
CREATE [OR REPLACE] TRIGGER nom_trigger
   {BEFORE | AFTER} evenement [OF liste_colonnes] ON nom_table
   [FOR EACH ROW [WHEN condition] ]
   corps_trigger
```

Quelles sont les évènements du déclenchement du trigger

- Il peut être un `INSERT`, `UPDATE`, `DELETE` ou une combinaison de ces opérations.
- Un `UPDATE` peut être suivi optionnellement par `OF liste_colonnes` (Le trigger se déclenche seulement si une des colonnes listées est modifiée).

6. Déclencheurs (triggers)

Syntaxe d'un trigger

```
CREATE [OR REPLACE] TRIGGER nom_trigger
  {BEFORE | AFTER} evenement [OF liste_colonnes] ON nom_table
  [FOR EACH ROW [WHEN condition] ]
  corps_trigger
```

- Un déclencheur sur instruction : il s'exécute une fois pour l'évènement du déclencheur, même si aucune ligne n'est affectée. C'est le type par défaut.
- Un déclencheur sur ligne : il s'exécute une fois pour chaque ligne affectée par l'évènement déclencheur. Il n'est pas exécuté si aucune ligne n'est affectée par l'évènement. Il est défini à l'aide de la clause `FOR EACH ROW`

6. Déclencheurs (triggers)

Syntaxe d'un trigger

```
CREATE [OR REPLACE] TRIGGER nom_trigger
   {BEFORE | AFTER} evenement [OF liste_colonnes] ON nom_table
   [FOR EACH ROW [WHEN condition] ]
   corps_trigger
```

Condition du déclenchement

- `[WHEN condition]` : c'est une partie optionnelle. Cela peut être toute condition booléenne SQL. Elle permet de limiter l'action du trigger au lignes qui répondent à une certaine condition.

6. Déclencheurs (triggers)

Syntaxe d'un trigger

```
CREATE [OR REPLACE] TRIGGER nom_trigger
   {BEFORE | AFTER} evenement [OF liste_colonnes] ON nom_table
   [FOR EACH ROW [WHEN condition] ]
   corps_trigger
```

Le corps du trigger

Il détermine l'action à exécuter. C'est un bloc PL/SQL ou un appel à une procédure.

Syntaxe du corps d'un trigger

```
[DECLARE]
BEGIN
[EXCEPTION]
END;
```

6. Déclencheurs (triggers)

Accès aux valeurs modifiées

- Afin d'accéder aux valeurs modifiées par l'évènement déclencheur, on peut utiliser les variables `new` et `old`
- Si on ajoute un client avec un nouveau nom, alors on peut récupérer ce nom avec la variable `:new.nom`
- Dans le cas d'une suppression ou une modification, on peut accéder à l'ancienne valeur de la variable nom avec la variable `:old.nom`

Exemple 1 : Accès aux valeurs modifiées

- On veut 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 client dans la table `clients`
- D'abord, on crée la table `log_clients` contenant toutes les colonnes de `clients`, mais elle contient en plus 3 autres colonnes : `username, datemodif, typemodif.`

Exemple 1 : Accès aux valeurs modifiées

```
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.quantite,:new.prix,:new.echeance, user ,sysdate,
   'INSERT');
END;
```

Exemple 2

- On dispose de deux tables :
  - `etudiant (matr_etu , nom, . . . , cod_cla )`
  - `classe (cod_cla , nbr_etu )`
- On veut écrire un trigger qui met à jour la table classe suite à une insertion d'un nouvel étudiant.

6. Déclencheurs (triggers)

Exemple 2

```
CREATE OR REPLACE TRIGGER maj_nb_etud
   AFTER INSERT ON etudiant
   FOR EACH ROW
BEGIN
   update classe
   set nbr_etu=nbr_etu+1
   where cod_cla=:new.cod_cla;
END;
```

Exemple 3

- On dispose d'une table :
  - `film (titre, annee, metrage, studio, categorie)`
- On veut écrire un trigger qui interdit les tentatives de diminuer le métrage d'un film.
- Pour cela, on écrit un trigger qui génère une exception de type `raise_application_error`. Celle-ci est une procédure qui affiche un message d'erreur et provoque l'échec du bloc du trigger.

6. Déclencheurs (triggers)

Exemple 3

```
CREATE OR REPLACE TRIGGER metrage
   BEFORE UPDATE OF metrage ON film
   FOR EACH ROW
BEGIN
   IF :old.metrage>:new.metrage THEN
      raise_application_error(n°erreur,’vous ne pouvez pas
      diminuer le metrage’);
   END IF;
END;
```

6. Déclencheurs (triggers)

Les prédicats INSERTING, UPDATING, DELETING

- Quand l'évènement déclencheur est une combinaison d'INSERT, UPDATE, DELETE, les prédicats INSERTING, UPDATING, DELETING, permettent de connaitre la raison de chaque déclenchement :
- INSERTING : True si le déclenchement est du à une insertion, False sinon.
- UPDATING : True s'il est du à une mise à jour, False sinon.
- DELETING : True s'il est du à une suppression, False sinon.

Exemple avec les prédicats INSERTING, DELETING

```
CREATE OR REPLACE TRIGGER maj_nb_etude
  AFTER INSERT OR DELETE ON etudiant
  FOR EACH ROW
BEGIN
  IF INSERTING THEN
    UPDATE classe SET nbr_etud=nbr_etud+1 where cod_cla=:new.cod_cla;
  ELSIF DELETING THEN
    UPDATE classe SET nbr_etud=nbr_etud-1 where cod_cla=:old.cod_cla;
  END IF;
END;
```

6. Déclencheurs (triggers)

Manipulation des triggers

- Activer ou désactiver un trigger :

```
ALTER TRIGGER <nom_trigger>  { ENABLE | DISABLE } ;
```

- Supprimer un trigger :

```
DROP TRIGGER <nom_trigger >;
```

- Déterminer les triggers de votre base de données :

```
SELECT <nom_trigger> FROM USER_TRIGGERS
```