Cours 4 : Les sous-programmes et les déclencheurs PL/SQL



Auteur : Riadh ZAAFRANI

2ème année SI – 2ème e-services

Mars 2010

Les sous-programmes

- Le langage **PL/SQL**, est un langage algorithmique complet ; il bénéficie de la possibilité de structuration de code, avec un procédé de décomposition de gros blocs de code en plus petits modules qui peuvent être appelés par d'autres modules.
- PL/SQL fournit les structures suivantes :
- **Procédure :** C'est un bloc PL/SQL nommé qui exécute une ou plusieurs actions et est appelé comme une commande PL/SQL. On peut passer et récupérer de l'information d'une procédure à travers sa liste d'arguments.

Les sous-programmes

- **Fonction :** C'est un bloc PL/SQL nommé qui renvoie une seule valeur et est utilisé comme une expression PL/SQL. On peut passer de l'information à une fonction à travers sa liste d'arguments.
- **Bloc anonyme :** C'est un bloc PL/SQL non nommé qui exécute une ou plusieurs actions. Un bloc anonyme permet au développeur de contrôler la portée des identifiants et la gestion des exceptions.
- **Package :** Un ensemble nommé de procédures, fonctions, types et variables. Un package n'est pas vraiment un module, c'est une application.

Les blocs nommés

- Tous les types de blocs PL/SQL nommés ont la structure suivante :
- **EN-TETE**
  IS/AS
  …
  BEGIN
  …
  [EXCEPTION]
  …
  END;
- L'entête indique comment un bloc nommé ou un programme doit être appelé.

Les procédures

- Une **procédure** est un sous-programme qui effectue un traitement particulier.
- Lorsqu'une procédure est créée, elle est d'abord compilée puis stockée dans la base de données sous sa forme compilée. Ce **code compilé** peut ensuite être exécuté à partir d'un autre bloc PL/SQL; le code source de la procédure est également stocké et n'a nul besoin d'être analysé une seconde fois à l'exécution.

Les procédures

- Le code, qui crée une **procédure** est :
- **CREATE [OR REPLACE] PROCEDURE**
  <nom_procédure>
  [(<nom_argument> [{IN | OUT | IN OUT]}] TYPE [,…])]
  {IS | AS}
  …
  BEGIN
  …
  EXCEPTION
  …
  END [<nom_procédure>];
- **IS | AS** Les deux mots clés sont équivalents; ils déterminent le début de la section des déclarations.

Les procédures

```
SQL> CREATE PROCEDURE AugmenterSalaire
  2 IS
  3 CURSOR c_employé IS SELECT fonction,salaire
  4         FROM employés
  5         FOR UPDATE OF salaire;
  6 begin
  7    for v_employé in c_employé Loop
  8       CASE v_employé.FONCTION
  9          When 'Chef des ventes' Then
 10               UPDATE employés SET salaire = salaire * 1.25
 11                    WHERE CURRENT OF c_employé;
 12          Else
 13               UPDATE employés SET salaire = salaire * 1.1
 14                    WHERE CURRENT OF c_employé;
 15       END CASE;
 16    end loop; COMMIT;
 17 END AugmenterSalaire;
 18 /
```

- **Procédure créée.**

Les procédures

- La création de la procédure n'implique pas l'exécution du code; elle est d'abord compilée puis stockée dans la base de données sous sa forme compilée, le code compilé pouvant ensuite être exécuté à partir d'un autre bloc PL/SQL.
- La syntaxe pour exécuter une procédure est :
- **<nom_procédure> [(valeur_argument [,…])];**
- Si la procédure n'a aucun argument, il faut l'appeler sans parenthèse ou sans aucun argument entre les parenthèses.

Les procédures

```
SQL> begin
  2   AugmenterSalaire;
  3 end;
  4 /
```

- **Procédure PL/SQL terminée avec succès.**

```
SQL> begin
  2   AugmenterSalaire();
  3 end;
  4 /
```

- **Procédure PL/SQL terminée avec succès.**

Les procédures

- On peut aussi utiliser l'instruction **CALL** pour appeler une procédure stockée, à partir de SQL*PLUS.
- **SQL>** CALL AugmenterSalaire();
- **Appel terminé.**

Les fonctions

- Une fonction est très semblable à une procédure.
- La différence réside en ce qu'un appel de procédure est en soi une instruction PL/SQL, tandis qu'un appel de fonction fait partie d'une expression.
- La syntaxe de création d'une fonction est :

Les fonctions

```
CREATE [OR REPLACE] FUNCTION <nom_fonction>
[(<nom_argument> [{IN | OUT | IN OUT]}] TYPE [,…])]
RETURN <Type_valeur>
{IS | AS}
…
BEGIN
…
   RETURN <expression>;
EXCEPTION
…
END [<nom_fonction>];
```

- **Type_valeur** C'est le type de la valeur de retour de la fonction.
- Une fonction doit contenir au moins une clause **RETURN** dans sa section d'exécution.

Les fonctions

- **SQL>** CREATE FUNCTION NombreProduits
  2 RETURN number
  3 IS
  4 v_nombre_produits number;
  5 begin
  6 SELECT count(*) INTO v_nombre_produits FROM produits;
  7 RETURN v_nombre_produits;
  8 end NombreProduits;
  9 /
- **Fonction créée.**

Les fonctions

- **SQL>** begin
  2 dbms_output.put_line('Le nombre de produits est :'||**NombreProduits**);
  3 end;
  4 /
- **Le nombre de produits est :4**
- **Procédure PL/SQL terminée avec succès.**

La suppression des blocs

- Comme les tables, les procédures et les fonctions peuvent faire l'objet d'une suppression, provoquant leur élimination du dictionnaire de données :
- **DROP PROCDEDURE <nom_procédure>;**
- **DROP FUNCTION <nom_fonction>;**

Les arguments

- Comme dans tout autre langage, vous pouvez créer des procédures et des fonctions qui reçoivent des arguments ayant différents modes et dont le passage se fera par valeur ou par référence.
- Les arguments peuvent avoir trois modes : **IN**, **OUT** ou **IN OUT**, IN étant la valeur par défaut.
- La syntaxe de déclaration des arguments est :
- **NOM_ARGUMENT [{IN | OUT | IN OUT}] TYPE [,…]**

Les arguments

- **SQL>** CREATE PROCEDURE AugmenterSalaire(Montant IN number)
  2 IS
  3 begin
  4 UPDATE employés SET salaire = salaire + montant;
  5 END AugmenterSalaire;
  6 /
- **Procédure créée.**

Les arguments : IN

- La valeur de l'argument réel est passée à la procédure lors de son invocation.
- A l'intérieur de celle-ci, l'argument formel se comporte comme une constante PL/SQL, à savoir qu'il est en lecture seule et ne peut donc être modifié.
- Lorsque la procédure se termine et que le contrôle repasse à l'environnement appelant, l'argument réel n'est pas modifié.

Les arguments : IN

- **SQL>** CREATE PROCEDURE AugmenterSalaire(Montant IN number)
  2 IS
  3 begin
  4 UPDATE employés SET salaire = salaire+montant;
  5 montant := 2000;
  6 END AugmenterSalaire;
  7 /
- **Avertissement : Procédure créée avec erreurs de compilation.**
- Comme vous pouvez l'observer, il est impossible d'affecter une valeur a un argument de type **IN**.

Les arguments : OUT

- La valeur de l'argument réel est ignorée lors de son invocation de la procédure.
- A l'intérieur de celle-ci, l'argument formel se comporte comme une variable PL/SQL, n'ayant pas été initialisée, contenant donc la valeur NULL et supportant les opérations de lecture et d'écriture.
- Lorsque la procédure se termine et que le contrôle repasse à l'environnement appelant, le contenu de l'argument formel est affecté à l'argument réel.

Les arguments : OUT

- **SQL>** CREATE PROCEDURE NombreProduits(nombre_produits OUT number)
  2 IS
  3 begin
  4 dbms_output.put_line('La valeur est :'|| nombre_produits);
  5 SELECT count(*) INTO nombre_produits FROM produits;
  6 end NombreProduits;
  7 /
- **SQL>** Declare
  2 v_nombre_produits number := 0;
  3 begin
  4 NombreProduits(v_nombre_produits);
  5 dbms_output.put_line ('La valeur est :'|| v_nombre_produits);
  6 end;
  7 /
- **La valeur est :**
- **La valeur est : 4**

Les arguments : IN OUT

- Ce mode est une combinaison de IN et OUT.
- La valeur de l'argument réel est passée à la procédure lors de son invocation.
- Lorsque la procédure se termine et que le contrôle repasse à l'environnement appelant, le contenu de l'argument formel est affecté à l'argument réel.

Les arguments : IN OUT

- **SQL>** CREATE PROCEDURE NombreProduits(nombre_produits IN OUT number)
  2 IS
  3 begin
  4 dbms_output.put_line('La valeur est :'|| nombre_produits);
  5 SELECT count(*) INTO nombre_produits FROM produits;
  6 end NombreProduits;
  7 /
- **SQL>** Declare
  2 v_nombre_produits number := 0;
  3 begin
  4 NombreProduits(v_nombre_produits);
  5 dbms_output.put_line('La valeur est :'|| v_nombre_produits);
  6 end;
  7 /
- **La valeur est : 0**
- **La valeur est : 4**

Association des arguments par position / par nom

- Les arguments peuvent être indiqués :
- **Par position :** Dans ce cas, chaque argument est remplacé par la valeur occupant la même position dans la liste.
- **Par nom :** Dans ce cas, chaque argument est passé dans un ordre quelconque en faisant apparaître la correspondance :
- **(nom_argument => valeur_argument [,…]);**

Passage de paramètres

- Le passage d'un paramètre de sous-programme peut s'effectuer de deux manières : par **référence** ou par **valeur**.
- Dans le premier cas, un pointeur sur le paramètre réel est passé au paramètre formel correspondant, et dans le second, la valeur du paramètre réel est copiée dans le paramètre formel.
- Par défaut, PL/SQL passera les paramètres **IN** par valeur et les paramètres **IN OUT** et **OUT** par référence.

Les blocs locaux

- Les procédures et les fonctions peuvent être utilisées sans être stockées dans la base, elles sont déclarées dans la section déclarative d'un bloc PL/SQL.

La surcharge de blocs

- Deux ou plusieurs blocs peuvent avoir le même nom et une liste différente de paramètres. De tels modules sont dits **surchargés**.
- Le compilateur compare le paramètre réel aux paramètres des listes des deux modules et il exécute alors le code du programme dont l'entête correspond.

La surcharge de blocs

```
SQL> Declare
  2   FUNCTION ControlValeur(a_date  DATE)
  3   RETURN  BOOLEAN IS
  4   Begin
  5     RETURN a_date <= SYSDATE;
  6   end;
  7   FUNCTION ControlValeur(a_nombre  NUMBER)
  8   RETURN  BOOLEAN IS
  9   Begin
 10     RETURN a_nombre >= 0;
 11    end;
 12  Begin
 13    If ControlValeur(UID) then
 14       dbms_output.put_line(UID);
 15    end if;
 16  end;
 17  /
```


Les packages

- Un package est une structure PL/SQL qui permet de stocker un ensemble d'objets logiquement associés et comprend deux parties distinctes :
  - La **spécification** qui contient la déclaration d'objets publics.
  - Le **corps** qui concerne l'implémentation des sous programmes publics et privés. Ces derniers sont invisibles de l'extérieur

Les packages

- la **spécification** et le **corps** sont stockés séparément dans le dictionnaire de données.
- La séparation de la déclaration de la **spécification** de celle du **corps** du package, permettra par exemple de compiler des modules qui appellent des sous_programmes spécifiés mais non encore implémentés.

La spécification de package

- La spécification d'un package contient des informations relatives au contenu du package.
- Les règles de déclarations sont :
  - L'ordre d'apparition des éléments contenus dans un package peut être quelconque.
  - Tous les types d'éléments ne doivent pas nécessairement être présents.
  - Toutes les déclarations de procédures et de fonctions doivent être des déclarations préalables. Une déclaration préalable décrit simplement le sous-programme et ses arguments sans en inclure le code.

La spécification de package

- La syntaxe de création d'un package :
- **CREATE [OR REPLACE] PACKAGE NOM_PACKAGE {IS | AS}**
  [Déclarations des variables et des types]
  [Déclarations d'exceptions]
  [Spécifications des curseurs]
  [Spécifications des modules]
  END [NOM_PACKAGE];

Exemple de spécification de package

- Le package **GererProduit** suivant regroupe essentiellement deux fonctions booléennes et une procédure.
- La procédure AddProduit reçoit les données d'un produit à créer, invoque les fonctions VerifieCatégorie et VerifieFournisseur pour valider les valeurs reçues puis insère un n-uplet à la table produit.

Exemple de spécification de package

```
SQL> CREATE OR REPLACE PACKAGE GererProduit
  2 AS
  3       CURSOR c_produit RETURN PRODUIT%ROWTYPE;
  4       v_produit c_produit%ROWTYPE;
  5       e_PasDeCatégorie EXCEPTION;
  6       e_PasDeFournisseur EXCEPTION;
  7       TYPE t_Produits IS TABLE OF PRODUIT%ROWTYPE
  8          INDEX BY BINARY_INTEGER;
  9       FUNCTION VerifieCatégorie(a_code_catégorie
CATEGORIE.CODE_CATÉGORIE%TYPE)
 10             RETURN BOOLEAN;
 11       FUNCTION VerifieFournisseur(a_no_fournisseur
FOURNISSEUR.NO_FOURNISSEUR%TYPE)
 12             RETURN BOOLEAN;
 13       PROCEDURE AddProduit(a_produit PRODUIT%ROWTYPE);
 14  END GererProduit;
 15  /
```

Package créé.

Eléments publics et privés d'un package

- L'un des concepts centraux des packages est le niveau de confidentialité de ses éléments.
- Un élément de package, peut être en effet public ou privé.
- **Public :** Défini dans les spécifications. Un élément public peut être référencé dans d'autres programmes et blocs PL/SQL.
- **Privé :** Défini uniquement dans le corps du package. Un élément privé ne peut être référencé en dehors du package.
- La séparation des éléments privés et publics offre aux développeurs un contrôle sur leurs données et programmes.

Comment référencer les éléments d'un package

- Un package est propriétaire de ses objets.
- Pour référencer n'importe lequel de ces objets; on préfixe le nom de l'objet avec le nom du package :
- **NOM_PACKAGE.NOM_OBJET;**

Le corps de package

- Le corps d'un package ne peut pas précéder la spécification du package.
- Il peut inclure des déclarations additionnelles qui sont globales dans le corps du package, mais qui ne sont pas visibles dans la spécification.
- Toute déclaration préalable dans la spécification de package doit s'accompagner d'une définition dans le corps du package, la spécification du sous-programme devant être identique dans les deux emplacements.

Le corps de package

- La syntaxe de création est la suivante :
- **CREATE [OR REPLACE] PACKAGE BODY NOM_PACKAGE {IS | AS}**
  [Déclarations des variables et des types]
  [Spécifications et SELECT des curseurs]
  [Implémentations des modules]
  [BEGIN
  Ordres exécutables]
  [EXCEPTIONS
  Exceptions]
  END [NOM_PACKAGE];

Exemple du corps de package

```
CREATE OR REPLACE PACKAGE BODY GererProduit
IS
    v_utilisateur VARCHAR2(25);
    CURSOR c_produit RETURN PRODUIT%ROWTYPE
        IS SELECT NumProduit, NomProduit,No_fournisseur,
        Code_catégorie, quantité,prixunitaire FROM produit;
FUNCTION VerifieCatégorie(a_code_catégorie
catégories.code_catégorie%TYPE)
RETURN BOOLEAN AS
begin
   for v_code_catégorie in (Select code_catégorie from
catégories
                    Where code_catégorie = a_code_catégorie)
loop
        RETURN TRUE; -- Catégorie trouvée
   end loop;
   RETURN FALSE;
end VerifieCatégorie;
```

Exemple du corps de package

```
FUNCTION VerifieFournisseur(a_no_fournisseur
FOURNISSEUR.NO_FOURNISSEUR%TYPE)
            RETURN BOOLEAN;
begin
  for v_no_fournisseur in (Select no_fournisseur
          from fournisseur
          Where no_fournisseur = a_no_fournisseur)
loop
       RETURN TRUE;  -- Fournisseur trouvé
  end loop;
  RETURN FALSE;
end VerifieFournisseur;
```

Exemple du corps de package

```
PROCEDURE AddProduit(a_produit
PRODUIT%ROWTYPE) AS
BEGIN
IF NOT VerifieCatégorie(a_produit.Code_catégorie) THEN
      RAISE e_PasDeCatégorie
END IF;
IF NOT VerifieFournisseur(a_produit.No_fournisseur) THEN
       RAISE e_PasDeFournisseur
END IF;
INSERT INTO PRODUIT VALUES
          (a_produit.NumProduit, a_produit.NomProduit,
           a_produit.No_fournisseur,
a_produit.Code_catégorie,
           a_produit.quantité,a_produit.prixunitaire);
end AddProduit;
```

Exemple du corps de package

```
Begin
    SELECT USER INTO v_utilisateur FROM DUAL;
Exception
     When e_PasDeCatégorie then
        dbms_output.put_line('Erreur code_catégorie');
     When e_PasDeFournisseur then
        dbms_output.put_line('Erreur no_fournisseur');
     When others then
        dbms_output.put_line('Erreur Package
GererProduit');
END GererProduit;
/
```

- Corps de package créé.

Les déclencheurs

- Le mécanisme de ***triggers*** (que l'on peut traduire par 'déclencheur' ou 'réflexe') est une procédure qui est déclenchée par des évènements de **mise-à-jour *spécifiés par l'utilisateur*** et ne s'exécute que quand une condition est satisfaite.
- On peut considérer les ***triggers*** comme une extension du système de contraintes proposé par la clause **CHECK** : à la différence de cette dernière, **l'évènement déclencheur est explicitement indiqué**, et **l'action n'est pas limitée à la simple alternative acceptation/rejet**.

Les déclencheurs

- Les possibilités offertes sont très intéressantes. Citons :
- **1.** La possibilité d'éviter les risques d'incohérence dus à la présence de redondance.
- **2.** L'enregistrement automatique de certains évènements (*auditing*).
- **3.** La spécification de contraintes liées à l'évolution des données (exemple : le prix d'une séance ne peut qu'augmenter).
- **4.** Toute règle complexe liée à l'environnement d'exécution (restrictions sur les horaires, les utilisateurs, etc).

Principes des triggers

- Le modèle d'exécution des *triggers* est basé sur la séquence ***Evénement-Condition-Action*** **(ECA)** que l'on peut décrire ainsi :
- 1. un *trigger* est déclenché par un ***évènement***, spécifié par le programmeur, qui est en général une insertion, destruction ou modification sur une table ;
- 2. la première action d'un *trigger* est de tester une ***condition*** : si cette condition ne s'évalue pas à TRUE, l'exécution s'arrête ;
- 3. enfin ***l'action*** proprement dite peut consister en toute opération sur la base de données : les SGBD fournissent un langage impératif permettant de créer de véritables procédures.

Principes des triggers

- Une caractéristique importante de cette procédure (action) est de pouvoir manipuler simultanément les valeurs **ancienne** et **nouvelle** de la donnée modifiée, ce qui permet de faire des tests sur l'évolution de la base.
- Parmi les autres caractéristiques importantes, citons les deux suivantes :

1. Tout d'abord un ***trigger*** peut être exécuté au choix une fois pour un seul ordre SQL, ou à chaque ligne concernée par cet ordre.
2. Ensuite l'action déclenchée peut intervenir ***avant*** l'évènement, ou ***après***.

Principes des triggers

- L'utilisation des ***triggers*** permet de rendre une base de données ***dynamique***:
- Une opération sur la base peut en déclencher d'autres, qui elles-mêmes peuvent entraîner en cascade d'autres réflexes.
- Ce mécanisme n'est pas sans danger à cause des risques de boucle infinie.

Syntaxe des triggers

- La syntaxe de création d'un *trigger* est la suivante :
- **CREATE [OR REPLACE]**
  TRIGGER <nom-trigger>
  <quand> <événements> ON <table>
  [FOR EACH ROW] [WHEN <condition>]
  [DECLARE…]
  BEGIN
  <action>
  [EXCEPTION…]
  END [nom-trigger];

Syntaxe des triggers

- Les choix possibles sont :
- – **quand** peut être **BEFORE** ou **AFTER**.
- – **événements** spécifie **DELETE**, **UPDATE** ou **INSERT** séparés par des **OR**.
- – **FOR EACH ROW** est optionnel. En son absence le ***trigger*** est déclenché une fois pour toute requête modifiant la table, ***et ce sans condition***. Sinon condition est toute condition booléenne SQL.

Syntaxe des triggers

- – **action** est une procédure qui peut être implantée, sous Oracle, avec le langage PL/SQL. Elle peut contenir des ordres SQL ***mais pas de mise-à-jour de la table courante***.
- Les anciennes et nouvelles valeurs du tuple courant sont référencées par **:new.attr** et **:old.attr**.
- Il est possible de modifier new et old. Par exemple **:new.prix=500;** forcera l'attribut prix à 500 dans un BEFORE *trigger*.
- Remarque : la disponibilité de new et old dépend du contexte. Par exemple new est à NULL dans un *trigger* déclenché par DELETE.

Exemples des triggers

- Voici un exemple de *trigger* qui maintient la capacité d'un cinéma après chaque modification de la table *Salle*.
- **CREATE TRIGGER CumulCapacite**
  AFTER UPDATE OF capacite ON Salle
  FOR EACH ROW WHEN (:new.capacite != :old.capacite)
  BEGIN
  UPDATE Cinema
  SET capacite = capacite - :old.capacite + :new.capacite
  WHERE nom = :new.nomCinema;
  END;

Exemples des triggers

- Pour garantir la validité du cumul, il faudrait créer des *triggers* sur les événements **UPDATE**, **DELETE** et **INSERT**.
- Une solution plus concise est de recalculer systématiquement le cumul : dans ce cas on peut utiliser un *trigger* qui se déclenche globalement pour la requête :
- **CREATE TRIGGER CumulCapaciteGlobal**
  AFTER INSERT OR DELETE OR UPDATE OF capacite
  ON Salle
  BEGIN
  UPDATE Cinema C
  SET capacite = (SELECT SUM (capacite) FROM Salle S
  WHERE C.nom = S.nomCinema);
  END;

contrôle de l'exécution des triggers

- L'algorithme de contrôle de l'exécution d'une instruction LMD déclenchante est :

1. Exécute les déclencheurs **BEFORE**, s'il en existe
2. Exécute les déclencheurs **AFTER**, s'il en existe.

Exemples des triggers

- Voici un autre exemple de *trigger*. Nous allons d'abord créer une table **temp_affichage**, une séquence **compteur** et un package **TriggerMoment** :
- **SQL>** CREATE Table Temp_affichage(
  2 ID_Affichage Number(2),
  3 Affichage Varchar(50));
  Table créée.
- **SQL**> CREATE SEQUENCE compteur START WITH 1 INCREMENT BY 1;
  Séquence créée.
- **SQL**> CREATE OR REPLACE Package TriggerMoment AS
  2 v_compteur NUMBER := 0;
  3 END TriggerMoment;
  4 /
  Package créé.

Exemples des triggers

- Nous allons ensuite créer 3 déclencheurs UPDATE sur la table produit :
- **SQL**> CREATE OR REPLACE TRIGGER **BEFOREinstruction**
  2 BEFORE UPDATE ON produit
  3 BEGIN
  4 INSERT INTO Temp_affichage values (compteur.NextVal,
  5 'Before niveau instruction : compteur = ' ||
  6 TriggerMoment.v_compteur);
  7 TriggerMoment.v_compteur := TriggerMoment.v_compteur + 1;
  8 END BEFOREinstruction;
  9 /
- **Déclencheur créé.**

Exemples des triggers

- **SQL>** CREATE OR REPLACE TRIGGER **AFTERinstruction1**
  2 AFTER UPDATE ON produit
  3 BEGIN
  4 INSERT INTO Temp_affichage values (compteur.NextVal,
  5 'After niveau instruction 1 : compteur = ' ||
  6 TriggerMoment.v_compteur);
  7 TriggerMoment.v_compteur := TriggerMoment.v_compteur + 1;
  8 END Afterinstruction1;
  9 /
- **Déclencheur créé.**

Exemples des triggers

- **SQL>** CREATE OR REPLACE TRIGGER **AFTERinstruction2**
  2 AFTER UPDATE ON produit
  3 BEGIN
  4 INSERT INTO Temp_affichage values (compteur.NextVal,
  5 'After niveau instruction 2 : compteur = ' ||
  6 TriggerMoment.v_compteur);
  7 TriggerMoment.v_compteur := TriggerMoment.v_compteur + 1;
  8 END Afterinstruction2;
  9 /
- **Déclencheur créé.**

Exemples des triggers

- Nous allons émettre maintenant l'instruction UPDATE, qui affecte l'ensemble des lignes de la table produit :
- SQL> update produit set prix_unitaire = 30;
  2 ligne(s) mise(s) à jour.
- **SQL>** select * from temp_affichage;
- **ID_AFFICHAGE AFFICHAGE**
  ---------- ----------
  1 Before niveau instruction : compteur = 0
  2 After niveau instruction 2 : compteur = 1
  3 After niveau instruction 1 : compteur = 2

Exemples des triggers

- Chaque déclencheur qui s'exécute **voit les changements introduits par les déclencheurs précédents**, de même que **toutes les modifications déjà apportées** par l'instruction dans la base de données.
- Cela est reflété par la valeur de compteur affichée par chaque déclencheur.
- L'ordre d'exécution des déclencheurs de même type n'étant pas défini, il conviendra donc, si l'ordre importe, de grouper toutes les opérations en un seul déclencheur.

Exemples des triggers

- L'activation d'un déclencheur peut être effectuée au niveau d'une instruction ou d'une ligne.
- Un déclencheur de niveau ligne est exécuté une fois pour chaque ligne traitée par l'instruction déclenchante.

Exemples des triggers

- L'algorithme complet de contrôle de l'exécution d'une instruction LMD déclenchante est :

1. Exécute les déclencheurs **BEFORE de niveau instruction**, s'il en existe.
2. Pour chaque ligne affectée par l'instruction :
   1. Exécute les déclencheurs **BEFORE de niveau ligne**, s'il en existe.
   2. Exécute **l'instruction**.
   3. Exécute les déclencheurs **AFTER de niveau ligne**, s'il en existe.
3. Exécute les déclencheurs **AFTER de niveau instruction**, s'il en existe.

Exemples des triggers

- Nous avons détruit les déclencheurs précédents et nous avons créé deux déclencheurs de niveau instruction et deux de niveau ligne :
- **SQL**> CREATE OR REPLACE TRIGGER **BEFOREinstruction**
```
 2  BEFORE UPDATE ON produit
 3  BEGIN
 4      INSERT INTO Temp_affichage values
(compteur.NextVal,
 5       'Before niveau instruction : compteur = ' ||
 6       TriggerMoment.v_compteur);
 7      TriggerMoment.v_compteur :=
TriggerMoment.v_compteur + 1;
 8  END BEFOREinstruction;
 9  /
```
- **Déclencheur créé.**

Exemples des triggers

- SQL> CREATE OR REPLACE TRIGGER **AFTERinstruction**
```
 2  AFTER UPDATE ON produit
 3  BEGIN
 4      INSERT INTO Temp_affichage values
(compteur.NextVal,
 5       'After niveau instruction : compteur = ' ||
 6       TriggerMoment.v_compteur);
 7      TriggerMoment.v_compteur :=
TriggerMoment.v_compteur + 1;
 8  END Afterinstruction;
 9  /
```
- **Déclencheur créé.**

Exemples des triggers

- **SQL**> CREATE OR REPLACE TRIGGER **BEFOREligne**
```
 2  BEFORE UPDATE ON produit
 3  FOR EACH ROW
 4  BEGIN
 5      INSERT INTO Temp_affichage values
(compteur.NextVal,
 6       'Before niveau ligne : compteur = ' ||
 7       TriggerMoment.v_compteur);
 8      TriggerMoment.v_compteur :=
TriggerMoment.v_compteur + 1;
 9  END BEFOREligne;
10  /
```
- **Déclencheur créé.**

Exemples des triggers

- SQL> CREATE OR REPLACE TRIGGER **AFTERligne**
```
 2  AFTER UPDATE ON produit
 3  FOR EACH ROW
 4  BEGIN
 5      INSERT INTO Temp_affichage values
(compteur.NextVal,
 6       'After niveau ligne : compteur = ' ||
 7       TriggerMoment.v_compteur);
 8      TriggerMoment.v_compteur :=
TriggerMoment.v_compteur + 1;
 9  END AFTERligne;
10  /
```
- **Déclencheur créé.**

Exemples des triggers

- **SQL**> update produit set prix_unitaire = 30;
  2 ligne(s) mise(s) à jour.
- **SQL**> select * from temp_affichage;
- **ID_AFFICHAGE AFFICHAGE**

  ---

  1 Before niveau instruction : compteur = 0
  2 Before niveau ligne : compteur = 1
  3 After niveau ligne : compteur = 2
  4 Before niveau ligne : compteur = 3
  5 After niveau ligne : compteur = 4
  6 After niveau instruction : compteur = 5
- **6 ligne(s) sélectionnée(s).**

Exemples des triggers

Un dernier exemple :

SQL> CREATE SEQUENCE S_NoProduit START WITH 3 INCREMENT BY 1;
Séquence créée.
SQL> select S_NoProduit.nextval from dual;

NEXTVAL
3

SQL> Create or replace trigger **ProduitId**
2 BEFORE INSERT ON produit
3 FOR EACH ROW
4 BEGIN
5 SELECT S_NoProduit.nextval into :new.NoProduit FROM DUAL;
6 END ProduitId;
7 /
Déclencheur créé.

Exemples des triggers

SQL> insert into produit
(NOMPRODUIT,PRIX_UNITAIRE,NO_FOURNISSEUR,
CODE_CATÉGORIE)
2 values ('chang', 50,1,1);
1 ligne créée.
SQL> select * from produit where noproduit = 3;

NOPRODUIT	NOMPRODUIT	PRIX_UNITAIRE	NO_FOURNISSEUR	CODE_CATÉGORIE
3	**chang**	**50**	**1**	**1**

Les prédicats

- Dans un déclencheur, trois fonctions booléennes appelées prédicats peuvent être utilisées dans le corps du trigger pour déterminer de quelle opération il s'agit :
- **INSERTING** : la fonction retourne la valeur TRUE si l'instruction déclenchante est un INSERT.
- **UPDATING** : la fonction retourne la valeur TRUE si l'instruction déclenchante est un UPDATE.
- **DELETING** : la fonction retourne la valeur TRUE si l'instruction déclenchante est un DELETE.
- Exemple : If UPDATING THEN …
  ELSE If DELETING THEN …ELSE…