Triggers (ou déclencheurs)

1 Introduction

Un trigger permet de programmer toutes les règles de gestion qui n'ont pas pu être mises en place par des contraintes au niveau des tables (clés primaires, contraintes d'intégrité référentielle, contraintes de domaine...).
Un trigger peut être vu comme une procédure stockée qui est déclenchée par le SGBD en réponse à une modification survenue sur une table ou une vue (`INSERT, UPDATE, DELETE`). A la différence de la procédure stockée, l'exécution d'un trigger n'est pas explicitement opérée par une commande (par exemple `CALL` sous isql*plus), c'est l'événement de mise à jour de la table (ou de la vue) qui exécute automatiquement le code programmé dans le trigger.

2 Triggers associés à une table

Un trigger est associé à une table particulière. Il peut être déclenché avant (`BEFORE`) ou après (`AFTER`) un évènement d'insertion (`INSERT`), de suppression (`DELETE`) ou de modification (`UPDATE`) survenu sur une table.

Syntaxe :

```
CREATE [OR REPLACE] TRIGGER nomtrigger
BEFORE | AFTER
INSERT | DELETE | UPDATE OF col1, col2 …
[OR INSERT | DELETE | UPDATE OF col1, col2 …]
ON nomTable
[FOR EACH ROW]
DECLARE
      -- zone de déclaration des variables locales
BEGIN
      -- instructions
EXCEPTION
      -- traitement des exceptions
END;
```

- `BEFORE | AFTER` : précise si le trigger doit être déclenché avant ou après la mise à jour de la table.
- `INSERT | DELETE | UPDATE OF col1, col2 …` : précise l'évènement qui va déclencher le trigger. Les triggers d'insertion (`INSERT`) ou de suppression (`DELETE`) portent sur le tuple en entier de la table alors que les triggers de modification (`UPDATE`) portent sur un attribut particulier d'un tuple.
- `ON nomTable` : spécifie la table `nomTable` associée au trigger.
- `FOR EACH ROW` : le trigger sera exécuté autant de fois qu'il y a de lignes concernées par la mise à jour. Si on désire exécuter le trigger une seule fois quelque soit le nombre de lignes concernées, on n'utilisera pas cette directive dans la déclaration du trigger.

2.1 Row triggers (ou déclencheurs de lignes)

Un déclencheur de lignes va s'exécuter autant de fois qu'il y a de tuples concernés par la mise à jour d'une table. Il est déclaré avec la directive `FOR EACH ROW`. Ce n'est que dans ce type de trigger qu'on a accès aux anciennes valeurs (`:OLD`) et aux nouvelles valeurs (`:NEW`) des colonnes du tuple affecté par la mise à jour de la table.

Prenons par exemple le schéma relationnel suivant :

CAMPINGS (idCamping, nomCamping, villeCamping, nbEtoilesCamping, *nbBungalows*)
BUNGALOWS (idBungalow, nomBungalow, superficieBungalow, idCamping#)

L'attribut *nbBungalows* de la table `Campings` est un attribut calculé qui doit être mis à jour à chaque modification de la table `Bungalows`.

Insertion :

Dans le cas d'une insertion, seule la directive `:NEW` est intéressante. Un accès à l'ancienne valeur `:OLD` retournera `NULL` puisque l'ancienne valeur n'existe pas dans le cas d'une insertion.

Lors de l'insertion d'un tuple dans la table `Bungalows`, il faut incrémenter l'attribut *nbBungalows* de la table `Campings`. Pour cela, on crée un trigger `AFTER INSERT` qui se déclenchera après l'insertion d'un tuple dans la table `Bungalows`.
La directive `:NEW` va permettre d'accéder à la valeur de l'attribut `idCamping` correspondant au tuple inséré dans la table `Bungalows`.

```
CREATE OR REPLACE TRIGGER tr_aft_ins_Bungalows
AFTER INSERT ON Bungalows
FOR EACH ROW
BEGIN
   UPDATE CAMPINGS SET nbBungalows = nbBungalows + 1 WHERE idCamping =
                                                          :NEW.idCamping;
END;
```

Suppression:

Dans le cas d'une suppression, seule la directive `:OLD` est intéressante. Un accès à la nouvelle valeur `:NEW` retournera `NULL` puisque la nouvelle valeur n'existe pas, elle a été supprimée.

Lors de la suppression d'un tuple dans la table `Bungalows`, il faut décrémenter l'attribut *nbBungalows* de la table `Campings`. Pour cela, on crée un trigger `AFTER DELETE` qui se déclenchera après la suppression d'un tuple dans la table `Bungalows`.
La directive `:OLD` va permettre d'accéder à la valeur de l'attribut `idCamping` correspondant au tuple supprimé dans la table `Bungalows`.

```
CREATE OR REPLACE TRIGGER tr_aft_del_Bungalows
AFTER DELETE ON Bungalows
FOR EACH ROW
BEGIN
   UPDATE CAMPINGS SET nbBungalows = nbBungalows - 1 WHERE idCamping =
                                                          :OLD.idCamping;
END;
```

Modification :

Les triggers de type UPDATE permettent de manipuler à la fois les directives :NEW et :OLD. En effet, la modification d'un tuple d'une table fait intervenir une nouvelle donnée (:NEW) qui en remplace une ancienne (:OLD).

Si on modifie le camping auquel appartient un bungalow, il faut incrémenter l'attribut *nbBungalows* du nouveau camping mais aussi décrémenter l'attribut *nbungalows* de l'ancien camping. Pour cela, on crée un trigger AFTER UPDATE OF idCamping qui se déclenchera après la modification de l'attribut idCamping d'un tuple dans la table Bungalows.
La directive :NEW va permettre d'accéder à la nouvelle valeur de l'attribut idCamping et la directive :OLD va permettre d'accéder à l'ancienne valeur de l'attribut idCamping.

```
CREATE OR REPLACE TRIGGER tr_aft_upd_Bungalows
AFTER UPDATE OF idCamping ON Bungalows
FOR EACH ROW
BEGIN
   UPDATE CAMPINGS SET nbBungalows = nbBungalows - 1 WHERE idCamping =
                                                          :OLD.idCamping;
   UPDATE CAMPINGS SET nbBungalows = nbBungalows + 1 WHERE idCamping =
                                                          :NEW.idCamping;
END;
```

Regroupement d'évènements :

Des évènements (INSERT, UPDATE ou DELETE) peuvent être regroupés au sein d'un même trigger s'ils sont de même type (BEFORE ou AFTER). Ainsi, un seul trigger est à coder. Les instructions suivantes permettent de retrouver la nature de l'évènement déclencheur :

- IF (INSERTING) THEN s'exécute dans le cas d'une insertion.
- IF (UPDATING) THEN s'exécute dans le cas de la modification d'une colonne.
- IF (DELETING) THEN s'exécute dans le cas d'une suppression.

```
CREATE OR REPLACE TRIGGER tr_aft_maj_Bungalows
AFTER INSERT OR DELETE OR UPDATE OF idCamping ON Bungalows
FOR EACH ROW
BEGIN
   IF (INSERTING OR UPDATING) THEN
      UPDATE CAMPINGS SET nbBungalows = nbBungalows + 1 WHERE idCamping =
                                                          :NEW.idCamping;
   END IF;
   IF (DELETING OR UPDATING) THEN
      UPDATE CAMPINGS SET nbBungalows = nbBungalows - 1 WHERE idCamping =
                                                          :OLD.idCamping;
   END IF;
END;
```

RAISE_APPLICATION_ERROR

Dans un trigger BEFORE, il est possible d'empêcher la mise à jour d'une table en levant une erreur (ou exception) via la procédure RAISE_APPLICATION_ERROR.

Prenons par exemple le schéma relationnel suivant :

BUNGALOWS (idBungalow, nomBungalow, superficieBungalow, idCamping#)
SERVICES (idService, nomService, categorieService)
PROPOSER (idBungalow#, idService#)

On souhaite ajouter la règle de gestion : un bungalow propose 5 services au maximum.

Une solution peut être de programmer un trigger `BEFORE INSERT` qui se déclenchera avant l'insertion d'un tuple dans la table `Proposer` et qui empêchera l'insertion dans la table `Proposer` en levant une exception si le nombre de services proposés pour un bungalow est déjà égal à 5.

```
CREATE OR REPLACE TRIGGER tr_bef_ins_Proposer
BEFORE INSERT ON Proposer
FOR EACH ROW
DECLARE
   v_nbServices NUMBER;
BEGIN
   SELECT COUNT(idService) INTO v_nbServices
   FROM Proposer
   WHERE idBungalow = :NEW.idBungalow;
   IF v_nbServices >= 5 THEN
      RAISE_APPLICATION_ERROR(-20001, 'Le bungalow propose déjà 5 services');
   END IF;
END;
```

2.2 Statement triggers (ou déclencheurs d'état)

Un déclencheur d'état porte sur la globalité de la table et non sur chaque tuple de la table. Il va donc s'exécuter qu'**une seule fois**. Il est déclaré sans la directive `FOR EACH ROW`. Il n'est pas possible d'avoir accès aux valeurs des tuples mis à jour par l'évènement.

```
CREATE OR REPLACE TRIGGER tr_maj_Locations
BEFORE INSERT OR UPDATE OR DELETE ON Locations
BEGIN
   IF TO_CHAR(SYSDATE,'fmDAY') = 'DIMANCHE' THEN
      RAISE_APPLICATION_ERROR(-20010, 'Désolé pas de location le dimanche');
   END IF;
END;
```

3 Triggers associés à une vue

Un trigger `INSTEAD OF` permet de mettre à jour une vue multitable qui ne peut être modifiée directement. Le code du trigger sera alors exécuté à la place du code qui déclenche le trigger.

Syntaxe :

```
CREATE [OR REPLACE] TRIGGER nomtrigger
INSTEAD OF
INSERT | DELETE | UPDATE OF col1, col2 …
[OR INSERT | DELETE | UPDATE OF col1, col2 …]
ON nomVue
FOR EACH ROW
DECLARE
      -- zone de déclaration des variables locales
BEGIN
      -- instructions
EXCEPTION
      -- traitement des exceptions
END;
```

- `INSTEAD OF` : précise que le trigger est associé à une vue et que son code sera exécuté à la place de l'instruction qui tente de modifier les données de la vue. Dans un trigger `INSTEAD OF`, il n'y a donc pas de directives `BEFORE` ou `AFTER`.
- `FOR EACH ROW` : cette directive n'est plus optionnelle.

Exemple :

Prenons par exemple le schéma relationnel suivant :

CAMPINGS (idCamping, nomCamping, villeCamping, nbEtoilesCamping, *nbBungalows*)
EMPLOYES (idEmploye, nomEmploye, prenomEmploye, salaireEmploye, idCamping#, idEmployeChef#)

On souhaite créer la vue multitable suivante :

```
CREATE VIEW EmployesEtCampings AS
SELECT idEmploye, nomEmploye, prenomEmploye, e.idCamping, nomCamping, villeCamping,
       nbEtoilesCamping
FROM Employes e
JOIN Campings c ON c.idCamping = e.idCamping;
```

Le trigger `INSTEAD OF` suivant va permettre d'insérer des données dans les tables `Employes` et `Campings` à travers la vue `EmployesEtCampings`.

```
CREATE OR REPLACE TRIGGER TriggerAuLieuInsererVue
INSTEAD OF INSERT ON EmployesEtCampings
FOR EACH ROW
DECLARE
  v_nbEmployes NUMBER;
  v_nbCampings NUMBER;
BEGIN
  SELECT count(*) INTO v_nbCampings
  FROM Campings
  WHERE idCamping = :NEW.idCamping;

  IF v_nbCampings = 0 THEN
    INSERT INTO Campings(idCamping, nomCamping, villeCamping, nbEtoilesCamping, nbBungalows)
    VALUES (:NEW.idCamping, :NEW.nomCamping, :NEW.villeCamping, :NEW.nbEtoilesCamping, 0);
  END IF;

  SELECT count(*) INTO v_nbEmployes
  FROM Employes
  WHERE idEmploye = :NEW.idEmploye;

  IF v_nbEmployes = 0 THEN
    INSERT INTO Employes(idEmploye, nomEmploye, prenomEmploye, idCamping)
    VALUES (:NEW.idEmploye, :NEW.nomEmploye, :NEW.prenomEmploye, :NEW.idCamping);
  END IF;
END;
```