Institut Universitaire de Technologie
Aix+Marseille Université

Département Informatique
2ème année

Bases de Données Avancées
Les triggers d'ORACLE

Rosine Cicchetti

Sommaire

Préambule

Les bases de données telles qu'elles ont été vues jusqu'à présent sont dites passives dans la mesure où tout traitement sur la base est déclenchée explicitement par un utilisateur ou un programme. Par opposition aux précédentes, les ***bases de données actives*** peuvent avoir des réactions automatiques. Plus précisément, lorsque certains événements surviennent, le SGBD peut déclencher des règles actives dites aussi règles Événement-Condition-Action (ou règles ECA pour simplifier). La sémantique de telles règles est la suivante : ***quand un événement survient, si une condition est satisfaite, alors l'action définie est exécutée***.

Les règles actives ont des applications diverses. Elles permettent par exemple : de réagir en temps réel quand une situation critique est détectée ; d'envoyer des messages d'alerte ; d'effectuer des contrôles d'intégrité des données ; de propager automatiquement des mises à jour sur des attributs calculés ; de gérer la confidentialité des données…

La définition de règles actives permet de développer des applications plus modulaires (au lieu d'implémenter tel ou tel contrôle dans tous les programmes utilisateurs, une règle active directement spécifiée au niveau de la base effectue les vérifications requises), et, théoriquement, plus flexibles et faciles à maintenir. Néanmoins le développement et la mise au point de règles actives peuvent s'avérer ardus et devenir rapidement un « cauchemar » pour le programmeur (nous verrons plus loin pourquoi). Dans ORACLE, le concept de règle active, appelé trigger (ou déclencheur en français) est proposé sous deux formes distinctes :

- directement définis comme des « objets ORACLE », les triggers sont gérés par le noyau du SGBD et implémentent des réactions globales, centralisées et communes ;
- intégrés comme des modules de traitement dans les applications SQL*FORMS, les triggers participent à la personnalisation de ces applications. Dans ce cas, ce n'est pas la base de données qui est active mais simplement l'application conçue au-dessus de cette base.

À propos de ce document :

Ce support de cours a été conçu pour des étudiants connaissant la programmation en PL/SQL. Il propose une introduction aux bases de données actives et détaille l'utilisation des triggers créés dans ORACLE. Un survol des triggers de SQL*FORMS est également donné.

Conventions :

Dans ce support, les mots clefs SQL et PL/SQL sont indiqués en police `courier`, en majuscules et en gras, les commandes ORACLE sont en `courier`, en minuscules et en gras. Les éléments optionnels des ordres sont indiqués entre crochets. Les éléments que doit spécifier l'utilisateur apparaissent entre < et >. L'alternative est représentée par : `|`. Tous les éléments issus directement ou non du schéma d'une base de données sont notés en majuscules, ceux qui sont spécifiés par le développeur en PL/SQL sont indiqués en minuscules. La base de données exemple est celle utilisée en TP.

1 - Les triggers comme objets ORACLE

Techniquement, les triggers objets ORACLE (appelés par la suite triggers ORACLE, pour simplifier) sont des blocs stockés, écrits en PL/SQL, assez similaires aux procédures stockées (Cf. support de cours : Le langage PL/SQL d'ORACLE). Ils sont dotés d'un en-tête particulier et soumis à diverses restrictions. Globalement, la définition d'un trigger se présente de la manière suivante.

```
<specification_en_tete_trigger>
<specification_evenement_declencheur>
<specification_condition>
<specification_action>
```

Avant de détailler la syntaxe de définition de triggers ORACLE, nous étudions, dans le paragraphe suivant, les concepts fondamentaux pour l'utilisation de triggers.

1.1 - Concepts fondamentaux

Les triggers sont des mécanismes complexes qui font appel à diverses notions. Nous les détaillons ainsi que les clauses Événement, Condition et Action des triggers ORACLE.

1.1.1 - La partie Événement

Les événements déclencheurs de triggers ORACLE correspondent à des opérations effectuées sur la base de données. On distingue l'opération réalisée par le SGBD de son événement qui est en quelque sorte l'ordre d'exécuter cette opération, ordre donné par un utilisateur ou un programme. Les événements déclencheurs de triggers sont associés à des opérations de :

- manipulation de données (LMD) : seules les trois opérations de mise à jour **INSERT**, **DELETE** et **UPDATE** sont concernées. Par simplification, les triggers déclenchés par ces ordres sont appelés, par la suite, triggers LMD ;
- de définition de données (LDD) : les commandes **ALTER**, **CREATE**, **DROP** et **RENAME** peuvent être utilisées. Les triggers déclenchés par les ordres mentionnés sont, par la suite, appelés triggers LDD.
- de contrôle de données (LCD) : les ordres **GRANT** et **REVOKE** ainsi que diverses commandes relevant de l'administration de la base peuvent déclencher des triggers, par la suite, appelés triggers LCD.

Ainsi donc, l'événement déclencheur d'un trigger ORACLE est l'un des ordres mentionnés ci-dessus. De plus, les ordres du LMD peuvent être combinés par des **OR** et pour **UPDATE** il est possible de spécifier les attributs dont la mise à jour déclenchera le trigger. Si aucun attribut n'est précisé, toute opération de modification provoquera l'exécution du trigger. Les spécifications suivantes sont valides pour la clause Événement d'un trigger ORACLE.

```
INSERT
INSERT OR UPDATE OF <nom_attribut1> [,<nom_attribut2>…]
INSERT OR UPDATE OR DELETE
CREATE
```

Tout trigger LMD ORACLE est associé exactement à ***une*** relation de la base de données.

Outre l'événement déclencheur, la définition de trigger doit spécifier la chronologie liant

l'opération de mise à jour qui déclenche le trigger et l'action réalisée par le trigger. Plus précisément soit l'action réalisée par le trigger est effectuée avant l'opération qui le déclenche, soit elle est effectuée après. Les mots clefs **BEFORE** et **AFTER** permettent de spécifier ces deux cas de figure.

La figure 1 illustre la chronologie d'un trigger de type **BEFORE**. À l'instant *t1*, l'ordre d'exécution d'une opération sur la base est détecté. L'action du trigger déclenché par cette détection est alors exécutée entre les instants *t2* et *t3*. Puis, si aucune exception n'est levée par l'exécution de l'action du trigger, l'opération qui a déclenché le trigger est effectivement réalisée entre les instants *t4* et *t5*.

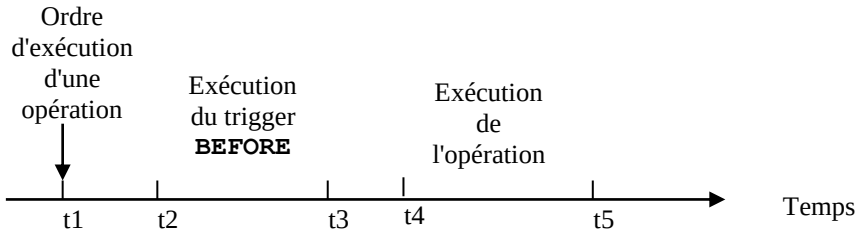

Figure 1 : Chronologie d'exécution pour un trigger **BEFORE**

La figure 2 illustre la chronologie d'un trigger de type **AFTER**. À l'instant *t1*, l'ordre d'exécution d'une opération est détecté. Cette opération est alors réalisée entre les instants *t2* et *t3*. Puis, si aucune exception n'est levée par l'exécution de l'opération, l'action du trigger est alors exécutée entre les instants *t4* et *t5*.

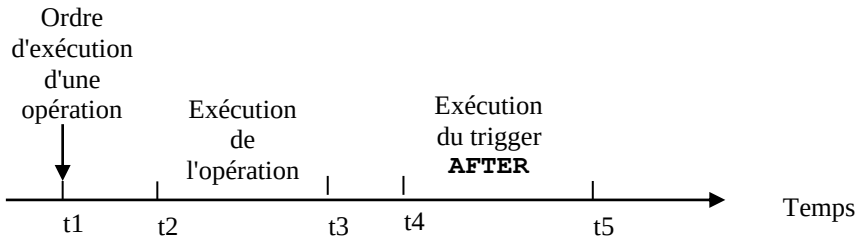

Figure 2 : Chronologie d'exécution pour un trigger **AFTER**

Évidemment, il faut soigneusement réfléchir à la chronologie spécifiée par rapport non seulement à l'opération déclenchant le trigger mais aussi à l'action réalisée. Les contraintes d'intégrité, en particulier de référence, peuvent aussi avoir une grande importance sur la chronologie choisie.
Par exemple, si l'action d'un trigger implémente la vérification d'une contrainte d'intégrité ou l'affectation de valeurs par défaut lors d'une insertion ou d'une modification dans une relation, il faut utiliser un trigger de type **BEFORE**, pour les raisons suivantes :

- si la contrainte n'est pas satisfaite par la mise à jour, le trigger **BEFORE** déclenche une exception et la mise à jour n'a pas lieu (en utilisant un trigger **AFTER**, la mise à jour serait opérée avant d'être défaite par l'échec du trigger. De toutes façons, ORACLE ne vous permet pas de mettre en œuvre ce type de démarche (Cf. paragraphe 2.3.1)) ;
- si une affectation de valeurs par défaut doit être opérée, le trigger **BEFORE** transmet ces valeurs à l'ordre de mise à jour évidemment avant que celui-ci opère (en utilisant un trigger **AFTER**, il faudrait réaliser une opération supplémentaire de modification (s'il s'agit de donner une valeur à un attribut quelconque) ou au pire une suppression et une nouvelle insertion (s'il s'agit de donner une valeur à une clef primaire), d'où plusieurs opérations réalisées alors qu'une seule doit l'être. De toutes façons, ORACLE ne vous permet pas de mettre en œuvre ce type de démarche (Cf. paragraphe 2.3.1)) ;

Prenons un autre exemple : l'action d'un trigger implémente la propagation d'une mise à jour sur un attribut calculé. Dans ce cas, il faut utiliser un trigger de type **AFTER**. En effet :

- si, pour une raison ou une autre, l'opération de mise à jour échoue, une exception système est levée et le trigger **AFTER** ne s'exécute pas. Comme la mise à jour n'a pas eu lieu, il est logique que sa propagation ne soit pas répercutée sur l'attribut calculé, la base reste donc cohérente (en utilisant un trigger **BEFORE**, il faudrait défaire l'action du trigger car la base est devenue incohérente, d'où deux opérations réalisées alors qu'aucune ne doit l'être. De toutes façons, ORACLE ne vous permet pas de mettre en œuvre ce type de démarche (Cf. paragraphe 2.3.1)) ;
- si l'opération de mise à jour est réalisée alors l'attribut calculé est mis à jour par le trigger **AFTER**. Si pour une raison ou une autre, le trigger échoue, ORACLE se charge de défaire l'opération de mise à jour.

1.1.2 - Granularité des triggers LMD

Pour les triggers ORACLE déclenchés par un ordre du LMD, et ***uniquement pour ceux-ci***, il est possible de spécifier la granularité du trigger qui peut être orienté instance ou orienté ensemble.

Lorsqu'un trigger est orienté instance (dans ORACLE, on parle de *row trigger*), sa partie action (le corps du bloc PL/SQL) est exécuté pour chaque tuple concerné par l'ordre de mise à jour déclencheur du trigger. Par exemple, si un trigger est déclenché par un **UPDATE** et que la condition spécifiée dans la clause **WHERE** du **UPDATE** est vérifiée pour *n* tuples, l'action du trigger est exécutée itérativement *n* fois et chaque exécution successive concerne un des *n* tuples. Si un tel trigger comporte une partie condition, celle-ci est évaluée individuellement pour chaque tuple à chaque nouvelle exécution (Cf. paragraphe 2.1.3). La spécification d'un trigger orienté instance se fait grâce aux mots clefs **FOR EACH ROW**. En revanche, l'action d'un trigger orienté ensemble *statement trigger* dans ORACLE) n'est exécutée qu'une seule fois (quel que soit le nombre de tuples concernés). Par défaut (i.e. en l'absence des mots clefs **FOR EACH ROW**), un trigger est orienté ensemble.
Revenons sur la chronologie d'exécution des triggers **BEFORE** ou **AFTER**. Si ces triggers sont orientés ensemble, l'enchaînement chronologique des exécutions décrit par les figures 1 et 2 s'applique exactement de la même manière. Ce principe général est toujours vrai pour les triggers orientés instance, néanmoins l'enchaînement chronologique peut être affiné en faisant apparaître les itérations successives.
La figure 3 illustre l'enchaînement des exécutions pour un trigger LMD orienté instance de type **BEFORE**. À l'instant *t1*, un ordre de mise à jour est détecté. Une lecture des *n* tuples concernés par l'opération est réalisée. L'action du trigger est alors exécutée pour le premier des *n* tuples puis l'opération de mise à jour concernant ce premier tuple est effectuée. L'action du trigger est exécutée à nouveau pour le deuxième tuple concerné, elle est suivie de l'opération de mise à jour portant sur ce deuxième tuple et ainsi de suite jusqu'à ce que les *n* tuples considérés aient été traités.

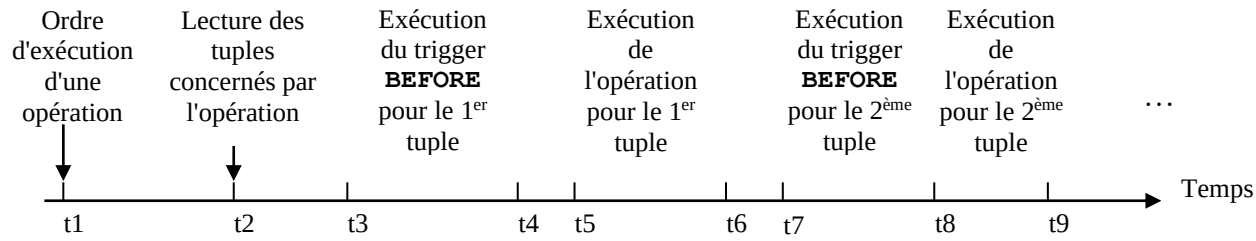

Figure 3 : Chronologie d'exécution pour un trigger **BEFORE** orienté instance

Si le trigger orienté instance est de type **AFTER**, la chronologie des exécutions, illustrée par la

figure 4, est similaire à celle des triggers de type **BEFORE**, à ceci près que :

- chaque itération commence par l'exécution de l'opération de mise à jour et s'achève par l'exécution de l'action du trigger ;
- et qu'il n'y a pas de lecture préalable des tuples concernés par l'opération de mise à jour.

Ce dernier point fait, qu'à coût d'exécution identique pour l'opération de mise à jour et l'action du trigger, un trigger de type **AFTER** est légèrement plus performant qu'un trigger de type **BEFORE**.

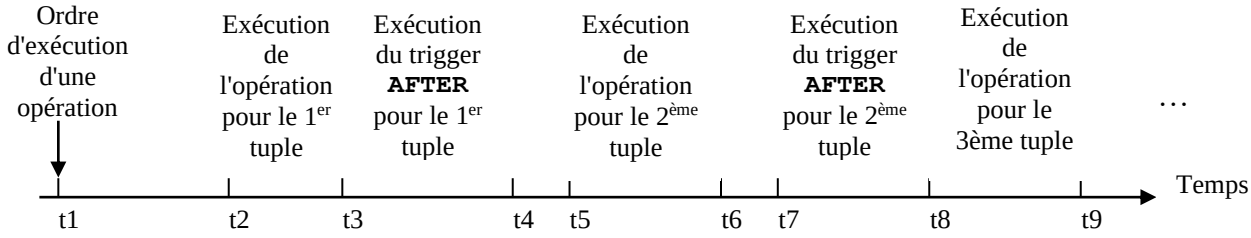

Figure 4 : Chronologie d'exécution pour un trigger **AFTER** orienté instance

Les triggers orientés instance sont prévus pour effectuer des traitements individuels de tuple (comme la mise en œuvre de règles de gestion ou de contraintes d'intégrité complexes) alors que ceux orientés ensemble opèrent des traitements globaux. Évidemment les triggers déclenchés par un ordre du LDD sont par nature globaux et donc orientés ensemble.

1.1.3 - La partie Condition

Cette partie Condition ne peut être spécifiée que pour ***des triggers orientés instance***, donc des triggers LMD. Ces triggers sont associés à une seule relation. La partie condition (via le mot clef **WHEN**) n'autorise que des combinaisons booléennes de conditions simples sur ***cette*** relation, en fait elles sont similaires aux conditions de ***sélection*** utilisées dans la clause **WHERE** d'une requête SQL[1]. La définition de condition de jointure imbriquée, i.e. de sous-requête faisant appel à d'autres relations, est ***exclue***. Cette contrainte peut sembler très restrictive. Néanmoins, elle peut être contournée car toute condition plus complexe peut être implémentée dans la partie Action du trigger.

Si la condition spécifiée est évaluée à **TRUE**, alors l'action du trigger est exécutée, dans les autres cas (**FALSE** ou **NULL**) le trigger n'est pas déclenché. Comme il s'agit de triggers orientés instance, l'évaluation de la condition puis l'exécution éventuelle de l'action sont effectuées pour chaque tuple concerné***. L'évaluation de la condition n'a aucun effet sur l'ordre de mise à jour déclencheur du trigger***. Dans ce contexte, la chronologie de l'exécution pour des triggers de type **BEFORE** ou **AFTER** peut encore être affinée, comme l'illustrent les figures 5 et 6, en faisant apparaître les évaluations de conditions spécifiées dans la clause **WHEN**.

Dans le schéma d'exécution proposé par la figure 5, on suppose que le premier des *n* tuples lus après la détection de l'ordre de mise à jour, satisfait la partie condition du trigger dont l'action est réalisée pour ce tuple ainsi que l'opération de mise à jour. Le deuxième tuple est supposé ne pas satisfaire la condition du trigger. L'action de ce dernier n'est donc pas déclenchée, mais l'opération de mise à jour a lieu. Puis la condition du trigger est évaluée pour le troisième tuple et comme elle est satisfaite, l'action du trigger est réalisée …

[1] Les conditions PL/SQL ne peuvent pas être utilisées (e.g. pas d'appel à des fonctions stockées).

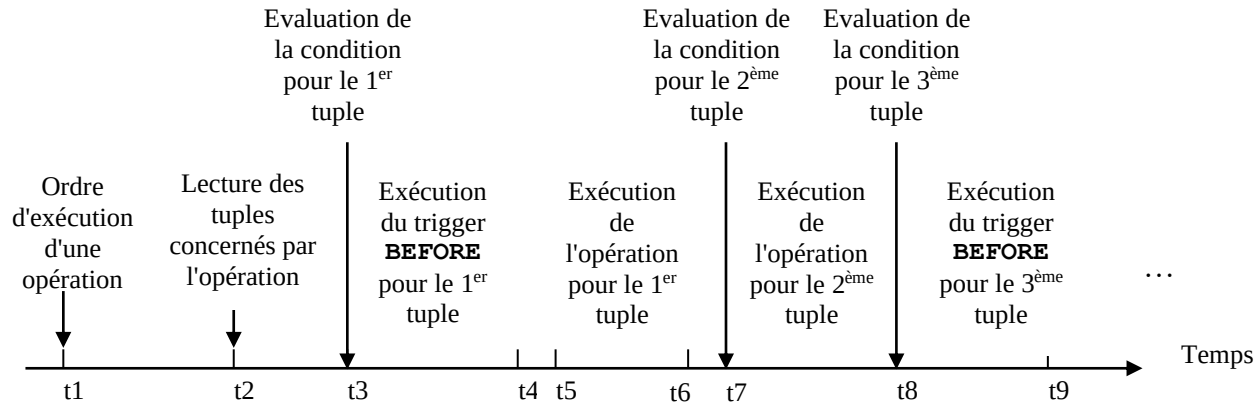

Figure 5 : Chronologie d'exécution pour un trigger **BEFORE** orienté instance avec clause **WHEN**

La figure 6 illustre le mécanisme d'exécution d'un trigger orienté instance de type **AFTER** et doté d'une clause **WHEN**. Comme précédemment, les opérations de mise à jour sont réalisées pour tous les tuples concernés, mais l'action du trigger ne l'est que si l'évaluation de la partie condition est vraie. Nous considérons ici que le premier et le troisième tuple satisfont la condition du trigger mais pas le deuxième.

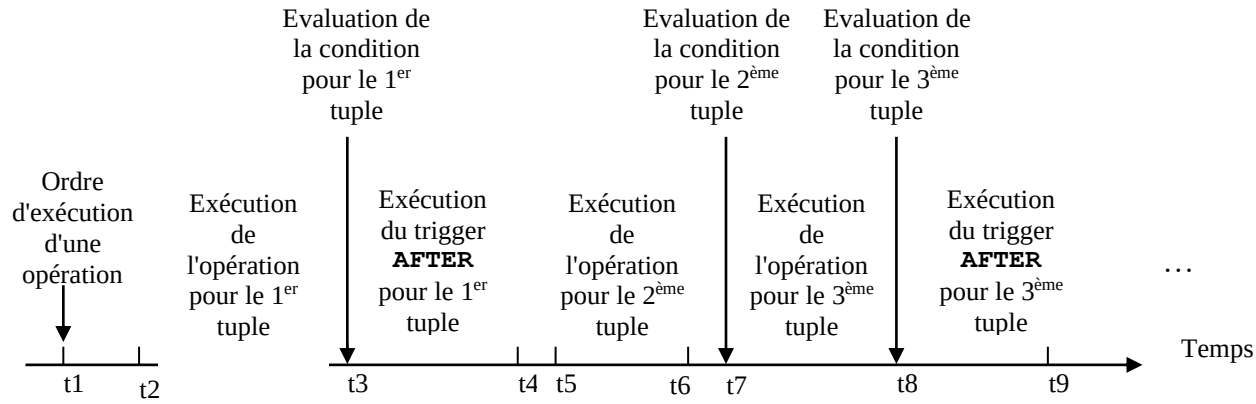

Figure 6 : Chronologie d'exécution pour un trigger **AFTER** orienté instance avec clause **WHEN**

1.1.4 - La partie Action

L'action d'un trigger ORACLE est un appel de procédure (écrite en PL/SQL ou en Java) ou un bloc PL/SQL respectant plusieurs contraintes importantes.

- le corps du trigger ne peut comporter aucun des ordres de gestion de transaction (i.e. **COMMIT**, **ROLLBACK** et **SAVEPOINT**) simplement parce qu'exécuté au cours d'une transaction utilisateur, le trigger ne peut en modifier les effets ;
- il ne peut pas inclure d'ordre relevant du LDD ;
- sa taille ne peut pas excéder 32 K ;
- lorsqu'un trigger est associé à la mise à jour d'une relation et que ***cette mise à jour est en cours, le trigger ne peut en aucun cas intervenir en lecture ou en écriture sur cette relation***. Le message d'erreur `'ORA-04091: table en mutation, déclencheur/fonction ne peut la voir'` signale toute tentative ne respectant pas cette contrainte (Cf. paragraphe 1.3.1).

En revanche, l'action d'un trigger ORACLE peut intégrer :

- des requêtes de type **SELECT** … **INTO** … **FROM** … ou **SELECT** … **FROM** … pour la définition de curseurs du moment que la relation interrogée (apparaissant dans la clause **FROM** de la requête de définition du curseur) n'est pas celle concernée par l'opération de mise à jour qui déclenche le trigger ;
- des ordres LMD sur d'autres relations de la base de données ;
- toute instruction valide en PL/SQL et ne faisant pas l'objet des restrictions mentionnées ci-avant avec bien sûr l'appel de procédures stockées pour lesquels les arguments passés peuvent être des valeurs d'attributs via **:NEW** et **:OLD** (Cf. paragraphe 2.1.5). ***Le code de ces procédures ne doit pas inclure d'ordre de gestion de transactions***.

Malgré les restrictions importantes indiquées, il est particulièrement facile, en définissant des triggers, de rendre les applications difficiles à mettre au point voire incontrôlables, tout simplement parce qu'un développement plus modulaire signifie aussi un code plus morcelé écrit par des équipes ou des programmeurs différents. Dans ce contexte, conserver une vision globale des traitements est difficile d'autant qu'ils prennent différentes formes (contraintes d'intégrité définies sur le schéma de la base, spécification de vues, procédures et fonctions stockées, programmes d'application divers et enfin triggers) et ***peuvent interagir***.

La figure 3 illustre un cas extrême de déclenchement récursif de triggers. Le trigger T1 est déclenché par un ordre **INSERT** sur la relation R1. Sa partie Action intègre un ordre de modification de la relation R2. Or, comme un trigger T2 est défini sur R2 et déclenché par un ordre **UPDATE**, l'exécution de T1 déclenche l'exécution de T2 (on parle d'exécution en cascade de triggers). L'action de T2 effectue une suppression de tuples dans la relation R3 mais cette opération a pour effet de déclencher un troisième trigger T3. Comme ce dernier opère une insertion de tuples dans R1, il a pour conséquence de re-déclencher le trigger T1 et ainsi de suite.

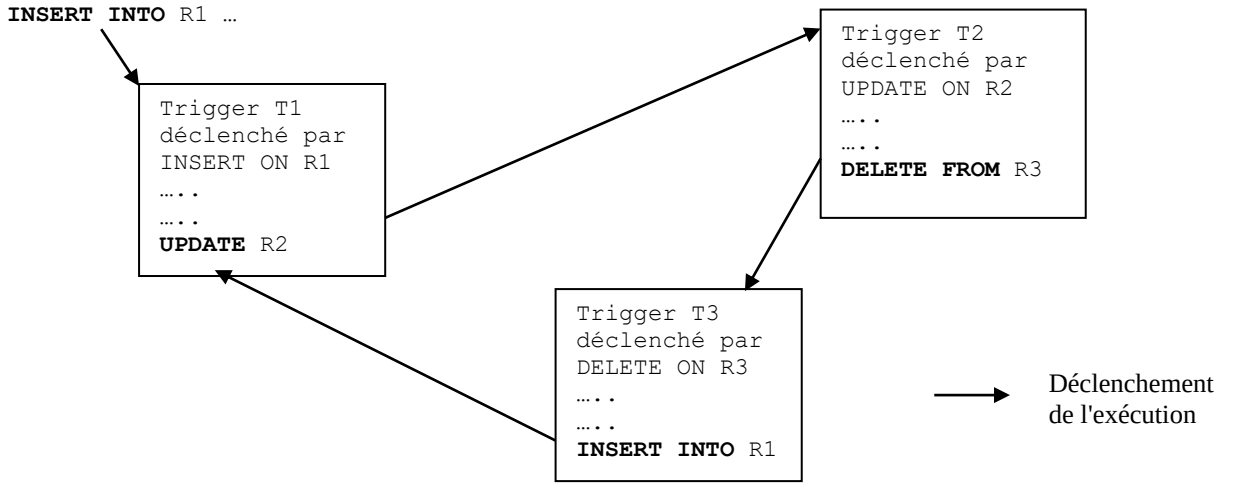

Figure 7 : déclenchement récursif de triggers

Concrètement, le cas décrit par la figure 7 ne constitue pas une boucle infinie, tout simplement parce que l'ordre intégré dans l'action du trigger T2 est **DELETE FROM** R3 et que la boucle d'exécution n'inclut aucun ordre **INSERT INTO** R3. Il arrivera donc forcément un instant où la relation R3 (qui contient un nombre fini de tuples) sera vide et donc l'exécution récursive finira par s'arrêter. Néanmoins, ORACLE introduit un « garde-fou » pour ce type de récursion ainsi que pour éviter les boucles infinies, en limitant le nombre d'appels en cascade de triggers à 32.

Lorsqu'un trigger LMD combine deux ou les trois ordres de mise à jour comme événements déclencheurs, on peut, dans l'action du trigger, distinguer quel événement particulier a été déclencheur de manière à différencier le code à exécuter. Pour effectuer cette distinction, les prédicats conditionnels **INSERTING**, **DELETING**, **UPDATING** doivent être utilisés comme condition d'un **IF**. Par exemple, **INSERTING** est évalué à **TRUE** si c'est un **INSERT** qui a déclenché le trigger.
Si l'événement d'un trigger est un ordre **UPDATE** avec précision de plusieurs attributs (**UPDATE OF** <nom_attribut1>, <nom_attribut2> [, <nom_attribut3>…]), alors le prédicat peut être de la forme : **UPDATING** ('<nom_attributi>' [, '<nom_attributj>'…]).

1.1.5 - Paramètres des événements LMD

Un événement véhicule de l'information via ses paramètres. Pour ***les triggers orientés instance*** (donc uniquement ceux déclenchés par un ordre du LMD), les paramètres de l'événement sont accessibles et/ou manipulables dans les parties condition et action du trigger en utilisant les prédicats **NEW** et **OLD**. **NEW** symbolise le tuple après l'opération de mise à jour et **OLD** le représente avant cette mise à jour, comme l'illustre la figure 8. Pour manipuler les valeurs avant et après, il suffit de ***préfixer le nom des attributs de la relation par NEW ou OLD***.

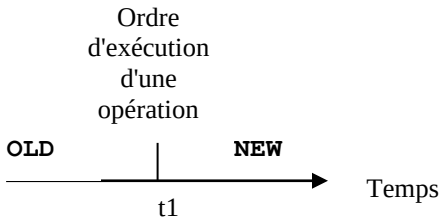

Figure 8 : paramètres d'un événement

Évidemment, suivant l'opération de mise à jour considérée, **NEW** et **OLD** peuvent ne pas avoir de signification. Plus précisément, les cas suivants doivent être considérés :

- si un ordre **INSERT** est déclencheur du trigger, **NEW** correspond forcément au nouveau tuple inséré mais **OLD.**<nom_attribut> est forcément à **NULL** (avant l'insertion, le tuple n'existe pas !) ;
- si l'ordre est un **UPDATE**, **NEW** fait référence aux valeurs après la modification et **OLD** aux anciennes valeurs du tuple modifié. Bien sûr, pour tout attribut non présent dans la clause **SET** du **UPDATE**, on a : **NEW.**<nom_attribut> = **OLD.**<nom_attribut> (Cf. exemple suivant) ;
- si l'ordre est un **DELETE**, **OLD** représente le tuple à supprimer et **NEW.**<nom_attribut> est forcément à **NULL** (après la suppression, le tuple n'existe plus !).

Exemple *: prenons l'exemple suivant de l'ordre de modification d'une note dont la valeur initiale est 10.*

```
UPDATE NOTATION SET NOTE_CC = 14
WHERE NUM_ET = 2422 AND CODE = 'BD' ;
```

Comme seul l'attribut NOTE_CC a été modifié par l'ordre précédent, nous avons :

```
NEW.NUM_ET = OLD.NUM_ET = 2422
NEW.CODE = OLD.CODE = 'BD'
NEW.NOTE_TEST = OLD.NOTE_TEST

NEW.NOTE_CC = 14 et OLD.NOTE_CC = 10
```

□

Que les triggers soient de type **BEFORE** ou **AFTER**, les valeurs référencées par **NEW.**<nom_attribut> ou **OLD.**<nom_attribut> ***sont accessibles en lecture*** et peuvent être utilisées :

- dans la partie Condition du trigger, i.e. la clause **WHEN** ;
- dans la partie Action, il faut alors préfixer l'expression par le caractère « : » (**:NEW.**<nom_attribut> ou **:OLD.**<nom_attribut>).

Si **:OLD.**<nom_attribut> ne peut jamais être modifié dans l'action du trigger (et pour cause !), **:NEW.**<nom_attribut> peut l'être par un trigger de type **BEFORE** (par exemple pour donner une valeur par défaut à un attribut) mais évidemment jamais par un trigger de type **AFTER** (car l'opération a déjà effectivement été réalisée).
S'il existe deux triggers, un de type **BEFORE** et un de type **AFTER**, pour une même relation et si le trigger **BEFORE** modifie des attributs **:NEW.**<nom_attribut>, ce sont les valeurs modifiées qui seront (éventuellement) lues par le trigger **AFTER**.

1.2 - Définition de trigger ORACLE

Avec les versions 7 d'ORACLE, le nombre de triggers créés par relation est limité à deux (un de type **BEFORE** et un de type **AFTER**). À partir de la version 8.1.0, cette restriction est éliminée. Il est donc possible de créer plusieurs triggers pour chaque relation de la base de données, mais le développeur n'a pas la maîtrise de l'ordre d'exécution de ces triggers par ORACLE (Cf. paragraphe 2.3.2). Un trigger peut être dans deux états (Cf. paragraphe 2.3.4) :

- activé (**ENABLE**) : il est exécuté par ORACLE lorsque son événement déclencheur est détecté ;
- désactivé (**DISABLE**) : bien que créé et stocké dans la base, le trigger n'est jamais déclenché … jusqu'au moment où le développeur le remet dans l'état **ENABLE**.

À sa création, un trigger est automatiquement activé. La définition de trigger respecte la syntaxe indiquée ci-après.

```
CREATE [OR REPLACE] TRIGGER <nom_trigger>
<chronologie>
<specification_evenements_declencheurs>
[FOR EACH ROW]
[WHEN ( <condition> )]
[DECLARE
     <declarations_locales> ]
BEGIN
     <liste_instructions>
[EXCEPTION
     <gestion_exceptions> ]
END ;
/
```

où :

- **OR REPLACE** permet de créer une nouvelle version d'un trigger existant sans avoir à le détruire préalablement ;

- <nom_trigger> doit respecter les contraintes des identificateurs SQL et PL/SQL. Les triggers créés par un même utilisateur doivent avoir des noms uniques. Les triggers sont stockés dans une table système particulière (Cf. paragraphe 2.3.4) aussi, bien que cela soit fortement déconseillé pour éviter toute ambiguïté pour les développeurs et les utilisateurs, rien n'interdit qu'un trigger porte le même nom qu'une relation, procédure, index… ou tout autre « objet ORACLE » n'étant pas un trigger défini par l'utilisateur ;
- <chronologie> est soit **BEFORE**, soit **AFTER** ;
- <specification_evenements_declencheurs> prend la forme :

  1. pour des événements du LMD :
     <evenements_declencheurs_LMD> **ON** <nom_relation>
     avec <evenements_declencheurs_LMD> qui peut être :
     - **INSERT**
     - **DELETE**
     - **UPDATE** [**OF** <nom_attribut1> [, <nom_attribut2> …]]
     - toute combinaison booléenne des événements précédents liés par des **OR**.

  2. pour des événements du LDD :
     <evenements_declencheurs_LDD> **ON DATABASE**
     avec <evenements_declencheurs_LDD> qui peut être :
     - **ALTER**
     - **CREATE**
     - **DROP**
     - **RENAME**

  3. pour des événements du LCD :
     <evenements_declencheurs_LCD> **ON DATABASE**
     avec <evenements_declencheurs_LCD> qui peut être :
     - **GRANT**
     - **REVOKE**

- Les mots clefs **FOR EACH ROW** doivent être précisés si le trigger est orienté instance. Ils sont omis pour un trigger orienté ensemble ;
- La clause **WHEN** ne peut être spécifiée que pour les triggers **FOR EACH ROW**. Dans ce cas, <condition> est une combinaison booléennes de conditions SQL simples dans lesquelles **NEW** et **OLD** peuvent être utilisés pour préfixer le nom des attributs à contrôler ;
- <declarations_locales> est la liste de toutes les déclarations locales nécessaires au trigger et qui respectent la syntaxe PL/SQL ;
- <liste_instructions> constitue le corps du trigger écrit en PL/SQL et respectant les contraintes données au paragraphe 2.1.4.
  Les instructions SQL ou PL/SQL peuvent inclure **:NEW** et **:OLD** pour préfixer le nom des attributs manipulés par l'ordre déclencheur du trigger ;
- <gestion_exceptions> décrit la manière dont les exceptions levées lors de l'exécution du trigger sont traitées.

Remarques :
- Toute déclaration et toute instruction de n'importe quelle section se terminent par le caractère « ; ».
- La partie déclarative n'est obligatoire que s'il y a des variables, constantes, curseurs, exceptions, types, procédures ou fonctions locaux au trigger.
- Le mot clef **DECLARE** ne doit être utilisé que pour introduire des déclarations, s'il n'y a pas d'élément local à déclarer, il est omis.

Les deux exemples qui suivent sont des triggers orientés instance donc déclenchés par des ordres du LMD. Le premier est de type **AFTER** et permet la propagation de mise à jour sur un attribut calculé. Le second, de type **BEFORE**, « normalise » les saisies de l'utilisateur.

Exemple *: supposons qu'il existe dans la base une relation calculée à partir de PROF dont le schéma est le suivant : ENCADREMENT(NOM_DEP, EFFECTIF_PROF). Lorsqu'un nouvel enseignant est inséré dans la relation PROF, l'effectif du département concerné, s'il est indiqué dans l'ordre d'insertion, est mis à jour automatiquement par le trigger Maj_Effectif_Prof.*

```
CREATE OR REPLACE TRIGGER Maj_Effectif_Prof
AFTER INSERT ON PROF
FOR EACH ROW
WHEN (NEW.NOM_DEP IS NOT NULL)

   UPDATE ENCADREMENT SET EFFECTIF_PROF = NVL(EFFECTIF_PROF,0) + 1
   WHERE NOM_DEP = :NEW.NOM_DEP ;
```

*Le trigger défini propage la mise à jour **après** toute insertion dans la relation PROF. C'est un trigger orienté instance car chaque tuple inséré a un effet sur l'effectif calculé. Il intègre une clause **WHEN** permettant de tester si la valeur de NOM_DEP dans le tuple inséré existe (utilisation de **NEW**.NOM_DEP). Si c'est le cas, l'action du trigger consiste simplement à mettre à jour dans la relation ENCADREMENT, le tuple donnant l'effectif du département concerné par l'insertion d'un enseignant (utilisation de :**NEW**.NOM_DEP). Le trigger est automatiquement déclenché après l'ordre d'insertion suivant :*

```
INSERT INTO PROF (NUM_PROF, NOM_PROF, NOM_DEP)
VALUES(24,'DUBOIS','INFORMATIQUE') ;
```

En revanche pour l'insertion ci-après, n'indiquant pas de département pour l'enseignant inséré, la condition du trigger est évaluée mais comme elle est fausse, l'action n'est pas exécutée.

```
INSERT INTO PROF (NUM_PROF, NOM_PROF, PRENOM_PROF, ADR_PROF)
VALUES(25,'SUN','PIERRE', 'AUBAGNE') ;
```

□

Exemple *: le trigger suivant, déclenché par une insertion dans ETUDIANT ou par la modification du nom ou du prénom, permet de mettre en majuscule les valeurs données à ces deux attributs.*

```
CREATE OR REPLACE TRIGGER Nom_Prenom_Majuscule
BEFORE INSERT OR UPDATE OF NOM_ET, PRENOM_ET ON ETUDIANT
FOR EACH ROW
BEGIN
   :NEW.NOM_ET := UPPER(:NEW.NOM_ET);
   :NEW.PRENOM_ET := UPPER(:NEW.PRENOM_ET);
END ;
```

*Le trigger défini opère **avant** toute insertion ou modification des attributs spécifiés, ce qui lui permet de transformer les nouvelles valeurs données à l'opération de mise à jour avant que celle-ci ne soit effectivement réalisée. L'ordre suivant ne déclenche pas le trigger, car ni le nom ni le prénom ne sont modifiés.*

```
UPDATE ETUDIANT SET ADR_ET = 'MARSEILLE'
```

```
WHERE NUM_ET = 3 ;
```
□

Dans l'exemple qui suit, un trigger orienté ensemble est défini. Déclenché par l'une des trois opérations de mise à jour, il utilise les prédicats conditionnels (**INSERTING**, **UPDATING**, **DELETING**) pour connaître la nature de l'opération ayant déclenché de cette exécution.

Exemple *: on souhaite faire un audit des opérations de mise à jour réalisées sur la relation NOTATION et pour cela, on veut en conserver la trace dans la relation suivante : AUDIT_NOTATION(DATE_OPERATION, OPERATION, UTILISATEUR).*

```
CREATE OR REPLACE TRIGGER Archivage_operation_NOTATION
AFTER INSERT OR DELETE OR UPDATE ON NOTATION
DECLARE
     operation VARCHAR2(12);
BEGIN
     IF INSERTING THEN operation := 'Insertion';
     ELSIF UPDATING THEN operation := 'Modification';
     ELSE operation := 'Suppression';
     END IF ;
     INSERT INTO AUDIT_NOTATION VALUES (sysdate, operation, user);
END ;
```

*Le trigger défini opère **après** toute opération de mise à jour sur NOTATION (une opération ne sera effectivement répertoriée que si elle a eu lieu). C'est un trigger orienté ensemble car quel que soit le nombre de tuples concernés par l'opération de mise à jour, on souhaite seulement connaître le type de l'opération réalisée. Le trigger utilise les prédicats conditionnels pour savoir quel ordre exact l'a déclenché puis il réalise une insertion dans la relation AUDIT_NOTATION en utilisant les fonctions **sysdate** et **user**.* □

```
CREATE OR REPLACE TRIGGER Message
AFTER CREATE ON DATABASE
DECLARE
     operation VARCHAR2(12);
BEGIN
     DBMS_OUTPUT.PUT_LINE('Relation créée');
END ;
```

1.3 - Utilisation de trigger ORACLE

Pour pleinement tirer profit des triggers ORACLE, il faut tenir compte de plusieurs aspects fondamentaux pour leur développement : l'interdiction d'opérer sur une relation en cours de mise à jour, l'ordre d'exécution des triggers, la gestion des exceptions, l'échec volontaire d'un trigger…

1.3.1 Relations en cours de mise à jour (*table en mutation*)

Une relation est considérée comme en cours de mise à jour (ou en mutation), lorsqu'une mise à jour (**INSERT**, **UPDATE** ou **DELETE**) est en train d'être effectuée sur cette relation[2]. ***Tous les triggers orientés instance ne peuvent opérer ni en lecture ni en écriture sur une relation en mutation***. L'objectif d'une telle restriction imposée par ORACLE est d'éviter que les triggers ne consultent des ensembles de données incohérents.

[2] ou que la relation est affectée par un **DELETE CASCADE** utilisé pour gérer les contraintes référentielles.

Cas particulier :
Il n'y a pas d'erreur ORA-04091 (la table ne sera pas en mutation) dans le cas suivant :
L'instruction DML déclenchante est un INSERT INTO … VALUES(…) avec valeurs littérales « en dur » (donc forcément une seule ligne inséré contrairement à une requête de chargement de type INSERT/SELECT qui pourrait traiter plusieurs lignes d'un coup)

***Exemple** : le trigger suivant est supposé afficher la meilleure note de test dans une matière chaque fois qu'une modification de note de test a lieu ou qu'une nouvelle insertion est réalisée dans NOTATION.*

```
CREATE OR REPLACE TRIGGER Nelle_meilleure_note
AFTER INSERT OR UPDATE OF NOTE_TEST ON NOTATION
FOR EACH ROW
WHEN (NEW.NOTE_TEST IS NOT NULL)
DECLARE
  note_max NOTATION.NOTE_TEST%TYPE ;

BEGIN
   SELECT MAX(NOTE_TEST) INTO note_max
   FROM NOTATION WHERE CODE = :NEW.CODE ;
   DBMS_OUTPUT.PUT_LINE('Meilleure note en '|| :NEW.CODE ||' est : '||
                        TO_CHAR(note_max));
END ;
```

L'ordre de modification suivant est exécuté et doit déclencher, après la mise à jour de chaque tuple concerné, le trigger précédent :

```
UPDATE NOTATION SET NOTE_TEST = NOTE_TEST * 1.1
WHERE CODE = 'BD' ;
```

Le problème qui se pose est que dès la première modification de tuple, la relation NOTATION est considérée comme en cours de mise à jour, donc mutante. La tentative de requête (première instruction de l'action du trigger) échoue, une exception système est levée et la première modification réalisée est défaite. Le message suivant est affiché :

```
ORA-04091: table NOTATION en mutation, déclencheur/fonction ne peut la voir
```

Pour remédier à ce problème, on pourrait spécifier un trigger orienté ensemble, mais ceci interdit d'introduire une partie condition et de tester le code de la matière des tuples modifiés ou insérés. Le trigger suivant est valide (mais ne réalise pas tout à fait la même action).

```
CREATE OR REPLACE TRIGGER Meilleure_note_par_matiere
AFTER INSERT OR UPDATE OF NOTE_TEST ON NOTATION
DECLARE
     CURSOR notes_max IS
     SELECT CODE, MAX(NOTE_TEST) NOTEMAX FROM NOTATION GROUP BY CODE ;

BEGIN
     FOR parcours_notes IN notes_max
     LOOP
        DBMS_OUTPUT.PUT_LINE('Meilleure note en '|| parcours_notes.CODE ||
                             ' est : '|| TO_CHAR(parcours_notes.NOTEMAX));
     END LOOP ;
END ;
```

*Comme le trigger est de type **AFTER** et surtout qu'il est orienté ensemble, son action ne sera effectivement réalisée qu'après la mise à jour de tous les tuples concernés par l'opération déclenchante. Ainsi la relation n'est pas considérée comme en mutation.* □

1.3.2 - Ordre d'exécution des triggers

Lorsque plusieurs triggers sont définis pour une même relation, l'ordre de leur exécution est transparent pour le développeur. La seule chose connue est qu'ORACLE exécute séquentiellement les triggers de même type avant de passer à ceux d'un autre type. Pour un ensemble de triggers de même type, l'ordre d'exécution d'ORACLE est arbitraire. Ainsi, chaque fois qu'un enchaînement d'exécution des triggers doit être défini, le développeur doit intégrer leur action dans un même bloc et il peut alors avoir recours aux prédicats conditionnels **INSERTING**, **UPDATING** et/ou **DELETING**.
Lorsque plusieurs triggers sont déclenchés successivement pour une même relation, chacun a accès aux valeurs des attributs via **NEW** et **OLD**. **:OLD.**<nom_attribut> correspond aux valeurs initiales lorsque la première opération de mise à jour est détectée et **:NEW.**<nom_attribut> correspond aux valeurs courantes, i.e. celles fixées par l'exécution la plus récente d'un trigger **INSERT** ou **UPDATE**.

1.3.3 - Gestion des exceptions

Si une exception utilisateur ou système est levée pendant l'exécution d'un trigger, alors l'action du trigger ainsi que l'opération qui l'a déclenché sont défaites (sauf si l'exception en question est gérée par le développeur). La gestion des exceptions se fait de la même manière que dans tout bloc PL/SQL. Cependant, dans la définition de triggers, le développeur peut utiliser des exceptions utilisateurs anonymes. De telles exception, non déclarées, permettent de ***faire échouer le trigger*** si une condition est vérifiée. L'utilisation de telles exceptions permet d'annuler (cas d'un trigger **AFTER**) ou de ne pas effectuer (cas d'un trigger **BEFORE**) l'opération de mise à jour associée au trigger. Le cas échéant, c'est donc ORACLE qui se charge de réaliser un **ROLLBACK**. L'utilisation de telles exceptions se fait de la manière suivante, dans le corps du trigger :

```
IF <condition> THEN
RAISE_APPLICATION_ERROR(<numero_exception>, 'message erreur') ;
```

ou, dans la partie **EXCEPTION** :

```
WHEN <exception>
THEN RAISE_APPLICATION_ERROR(<numero_exception>, 'message erreur') ;
```

<numero_exception> est un numéro variant dans l'intervalle [-20000 .. - 20999], intervalle réservé par ORACLE pour les exceptions utilisateurs anonymes. Si un trigger orienté instance comporte le déclenchement d'une exception, par **RAISE_APPLICATION_ERROR**, celle-ci peut être levée à chaque exécution itérative du trigger.

Exemple *: reprenons le trigger mettant à jour l'effectif des enseignants dans la relation ENCADREMENT. Si aucun département n'est donné lors de l'insertion d'un enseignant, alors l'échec du trigger est déclenché*

```
CREATE OR REPLACE TRIGGER Maj_Effectif_Prof
AFTER INSERT ON PROF
FOR EACH ROW
BEGIN
IF :NEW.NOM_DEP IS NOT NULL THEN
   UPDATE ENCADREMENT SET EFFECTIF_PROF = NVL(EFFECTIF_PROF,0)  + 1
   WHERE NOM_DEP = :NEW.NOM_DEP ;
```

```
ELSE
  RAISE_APPLICATION_ERROR(-20002,'L''enseignant '||TO_CHAR(:NEW.NUM_PROF)||
                          ' n''est affecté à aucun departement') ;
END IF ;
END ;                                                                    □
```

1.3.4 - Gestion de triggers ORACLE

Les triggers ORACLE sont écrits et compilés de la même manière que des blocs stockés PL/SQL. La création d'un trigger dans la base peut être effectuée avec des erreurs de compilation. La commande ORACLE **show errors** peut être utilisée pour consulter les erreurs.

Evidemment, pour exécuter un trigger, il faut réaliser, sous SQL, l'ordre qui permet de le déclencher.

De la même manière que pour les autres blocs stockés, ORACLE répertorie les dépendances entre les triggers et les autres objets Oracle. Si ces derniers sont modifiés, le trigger passe dans un état invalide et il est recompilé automatiquement à sa prochaine utilisation. Pour éviter cela et donc effectuer une re-compilation manuelle, il faut utiliser la commande suivante.

```
ALTER TRIGGER <nom_trigger> COMPILE ;
```

Un trigger peut être détruit par :

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

Un trigger peut être activé ou désactivé en utilisant la commande **ALTER** comme suit.

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

Tous les triggers d'une même relation peuvent être activés ou désactivés par :

```
ALTER TABLE <nom_relation> ENABLE ALL TRIGGER ;
ALTER TABLE <nom_relation> DISABLE ALL TRIGGER ;
```

Les triggers étant des blocs stockés, ils sont décrits dans le dictionnaire, plus précisément dans la vue système USER_TRIGGERS. La commande **desc** utilisée sur cette vue rend le résultat suivant.

```
Nom                           NULL ?    Type
 ----------------------------------------- --------
 TRIGGER_NAME                             VARCHAR2(30)
 TRIGGER_TYPE                             VARCHAR2(16)
 TRIGGERING_EVENT                         VARCHAR2(216)
 TABLE_OWNER                              VARCHAR2(30)
 BASE_OBJECT_TYPE                         VARCHAR2(16)
 TABLE_NAME                               VARCHAR2(30)
 COLUMN_NAME                              VARCHAR2(4000)
 REFERENCING_NAMES                        VARCHAR2(128)
 WHEN_CLAUSE                              VARCHAR2(4000)
 STATUS                                   VARCHAR2(8)
 DESCRIPTION                              VARCHAR2(4000)
 ACTION_TYPE                              VARCHAR2(11)
 TRIGGER_BODY                             LONG
```

L'attribut TRIGGER_TYPE indique le type du trigger, i.e. sa chronologie (**BEFORE** ou **AFTER**) et sa nature orienté instance (**FOR EACH ROW**) ou ensemble (STATEMENT). L'attribut TRIGGERING_EVENT stocke les événements déclencheurs du trigger. L'attribut STATUS prend les valeurs ENABLE ou DISABLE. L'attribut WHEN_CLAUSE donne le code de la partie condition du trigger. L'attribut TRIGGER_BODY stocke le code de la partie action du trigger.