V.7. Triggers

Les triggers (ou déclencheurs) sont des objets de la base de données. **Attachés à une table**, ils vont **déclencher l'exécution d'une instruction**, ou d'un bloc d'instructions, **lorsqu'une, ou plusieurs lignes sont insérées, supprimées ou modifiées** dans la table à laquelle ils sont attachés.

Dans ce chapitre, nous allons voir comment ils fonctionnent exactement, comment on peut les créer et les supprimer, et surtout, comment on peut s'en servir et quelles sont leurs restrictions.

V.7.1. Principe et usage

V.7.1.1. Qu'est-ce qu'un trigger ?

Tout comme les procédures stockées, les triggers servent à exécuter une ou plusieurs instructions. Mais à la différence des procédures, il n'est pas possible d'appeler un trigger : un trigger doit être déclenché par un événement.

Un trigger est **attaché à une table**, et peut **être déclenché** par :

— une insertion dans la table (requête `INSERT`) ;
— la suppression d'une partie des données de la table (requête `DELETE`) ;
— la modification d'une partie des données de la table (requête `UPDATE`).

Par ailleurs, une fois le trigger déclenché, ses instructions peuvent être exécutées soit juste avant l'exécution de l'événement déclencheur, soit juste après.

V.7.1.1.1. Que fait un trigger ?

Un trigger exécute un **traitement pour chaque ligne insérée, modifiée ou supprimée** par l'événement déclencheur. Donc si l'on insère cinq lignes, les instructions du trigger seront exécutées cinq fois, chaque itération permettant de traiter les données d'une des lignes insérées.

Les instructions d'un trigger suivent les mêmes principes que les instructions d'une procédure stockée. S'il y a plus d'une instruction, il faut les mettre à l'intérieur d'un bloc d'instructions. Les structures que nous avons vues dans les deux chapitres précédents sont bien sûr utilisables (structures conditionnelles, boucles, gestionnaires d'erreur, etc.), avec toutefois quelques restrictions que nous verrons en fin de chapitre.

Un trigger peut modifier et/ou insérer des données dans n'importe quelle table sauf les tables utilisées dans la requête qui l'a déclenché. En ce qui concerne la table à laquelle le trigger est attaché (qui est forcément utilisée par l'événement déclencheur), le trigger peut lire et modifier uniquement la ligne insérée, modifiée ou supprimée qu'il est en train de traiter.

V.7.1.2. À quoi sert un trigger ?

On peut faire de nombreuses choses avec un trigger. Voici quelques exemples d'usage fréquent de ces objets. Nous verrons plus loin certains de ces exemples appliqués à notre élevage d'animaux.

V.7.1.2.1. Contraintes et vérifications de données

Comme cela a déjà été mentionné dans le chapitre sur les types de données, MySQL n'implémente pas de contraintes d'assertion, qui sont des contraintes permettant de limiter les valeurs acceptées par une colonne (limiter une colonne `TINYINT` à `TRUE` (1) ou `FALSE` (0) par exemple). Avec des triggers se déclenchant avant l'`INSERT` et avant l'`UPDATE`, on peut vérifier les valeurs d'une colonne lors de l'insertion ou de la modification, et les corriger si elles ne font pas partie des valeurs acceptables, ou bien faire échouer la requête. On peut ainsi pallier l'absence de contraintes d'assertion.

V.7.1.2.2. Intégrité des données

Les triggers sont parfois utilisés pour remplacer les options des clés étrangères `ON UPDATE RESTRICT|CASCADE|SET NULL` et `ON DELETE RESTRICT|CASCADE|SET NULL`. Notamment pour des tables MyISAM, qui sont non-transactionnelles et ne supportent pas les clés étrangères. Cela peut aussi être utilisé avec des tables transactionnelles, dans les cas où le traitement à appliquer pour garder des données cohérentes est plus complexe que ce qui est permis par les options de clés étrangères.

Par exemple, dans certains systèmes, on veut pouvoir appliquer deux systèmes de suppression :

— une vraie suppression pure et dure, avec effacement des données, donc une requête `DELETE` ;
— un archivage, qui masquera les données dans l'application mais les conservera dans la base de données.

Dans ce cas, une solution possible est d'ajouter aux tables contenant des données archivables une colonne *archive*, pouvant contenir 0 (la ligne n'est pas archivée) ou 1 (la ligne est archivée). Pour une vraie suppression, on peut utiliser simplement un `ON DELETE RESTRICT|CASCADE|SET NULL`, qui se répercutera sur les tables référençant les données supprimées. Par contre, dans le cas d'un archivage, on utilisera plutôt un trigger pour traiter les lignes qui référencent les données archivées, par exemple en les archivant également.

V.7.1.2.3. Historisation des actions

On veut parfois garder une trace des actions effectuées sur la base de données, c'est-à-dire par exemple, savoir qui a modifié telle ligne, et quand. Avec les triggers, rien de plus simple, il suffit de mettre à jour des données d'historisation à chaque insertion, modification ou suppression. Soit directement dans la table concernée, soit dans une table utilisée spécialement et exclusivement pour garder un historique des actions.

V.7.1.2.4. Mise à jour d'informations qui dépendent d'autres données

Comme pour les procédures stockées, une partie de la logique "business" de l'application peut être codée directement dans la base de données, grâce aux triggers, plutôt que du côté applicatif (en PHP, Java ou quel que soit le langage de programmation utilisé). À nouveau, cela peut permettre d'harmoniser un traitement à travers plusieurs applications utilisant la même base de données.

Par ailleurs, lorsque certaines informations dépendent de la valeur de certaines données, on peut en général les retrouver en faisant une requête `SELECT`. Dans ce cas, il n'est pas indispensable de stocker ces informations. Cependant, utiliser les triggers pour stocker ces informations peut faciliter la vie de l'utilisateur, et peut aussi faire gagner en performance. Par exemple, si l'on a très souvent besoin de cette information, ou si la requête à faire pour trouver cette information est longue à exécuter. C'est typiquement cet usage qui est fait des triggers dans ce qu'on appelle les "vues matérialisées", auxquelles un chapitre est consacré dans la partie 6.

V.7.2. Création des triggers

V.7.2.1. Syntaxe

Pour créer un trigger, on utilise la commande suivante :

```
1 CREATE TRIGGER nom_trigger moment_trigger evenement_trigger
2 ON nom_table FOR EACH ROW
3 corps_trigger
```

- `CREATE TRIGGER nom_trigger` : les triggers ont donc un nom.
- `moment_trigger evenement_trigger` : servent à définir quand et comment le trigger est déclenché.
- `ON nom_table` : c'est là qu'on définit à quelle table le trigger est attaché.
- `FOR EACH ROW` : signifie littéralement "pour chaque ligne", sous-entendu "pour chaque ligne insérée/supprimée/modifiée" selon ce qui a déclenché le trigger.
- `corps_trigger` : c'est le contenu du trigger. Comme pour les procédures stockées, il peut s'agir soit d'une seule instruction, soit d'un bloc d'instructions.

V.7.2.1.1. Événement déclencheur

Trois événements différents peuvent déclencher l'exécution des instructions d'un trigger.

- L'insertion de lignes (`INSERT`) dans la table attachée au trigger.
- La modification de lignes (`UPDATE`) de cette table.
- La suppression de lignes (`DELETE`) de la table.

Un trigger est soit déclenché par `INSERT`, soit par `UPDATE`, soit par `DELETE`. Il ne peut pas être déclenché par deux événements différents. On peut par contre créer plusieurs triggers par table pour couvrir chaque événement.

V.7.2.1.2. Avant ou après

Lorsqu'un trigger est déclenché, ses instructions peuvent être exécutées à deux moments différents. Soit juste avant que l'événement déclencheur n'ait lieu (`BEFORE`), soit juste après (`AFTER`).

Donc, si vous avez un trigger `BEFORE UPDATE` sur la table *A*, l'exécution d'une requête `UPDATE` sur cette table va d'abord déclencher l'exécution des instructions du trigger, ensuite seulement les lignes de la table seront modifiées.

V.7.2.1.3. Exemple

Pour créer un trigger sur la table *Animal*, déclenché par une insertion, et s'exécutant après ladite insertion, on utilisera la syntaxe suivante :

```
1 CREATE TRIGGER after_insert_animal AFTER INSERT
2 ON Animal FOR EACH ROW
3 corps_trigger;
```

V.7.2.2. Règle et convention

Il ne peut exister qu'un seul trigger par combinaison `moment_trigger/evenement_trigger` par table. Donc un seul trigger `BEFORE UPDATE` par table, un seul `AFTER DELETE`, etc. Étant donné qu'il existe deux possibilités pour le moment d'exécution, et trois pour l'événement déclencheur, on a donc un **maximum de six triggers par table**.

Cette règle étant établie, il existe une convention quant à la manière de nommer ses triggers, que je vous encourage à suivre : *nom_trigger = moment_evenement_table*. Donc le trigger `BEFORE UPDATE ON Animal` aura pour nom : *before_update_animal*.

V.7.2.3. OLD et NEW

Dans le corps du trigger, MySQL met à disposition deux mots-clés : `OLD` et `NEW`.

- `OLD` : représente les valeurs des colonnes de la ligne traitée **avant qu'elle ne soit modifiée** par l'événement déclencheur. Ces valeurs peuvent être **lues, mais pas modifiées**.
- `NEW` : représente les valeurs des colonnes de la ligne traitée **après qu'elle a été modifiée** par l'événement déclencheur. Ces valeurs peuvent être **lues et modifiées**.

Il n'y a que dans le cas d'un trigger `UPDATE` que `OLD` et `NEW` coexistent. Lors d'une insertion, `OLD` n'existe pas, puisque la ligne n'existe pas avant l'événement déclencheur ; dans le cas d'une suppression, c'est `NEW` qui n'existe pas, puisque la ligne n'existera plus après l'événement déclencheur.

Premier exemple : l'insertion d'une ligne.

Exécutons la commande suivante :

```sql
INSERT INTO Adoption (client_id, animal_id, date_reservation, prix,
    paye)
VALUES (12, 15, NOW(), 200.00, FALSE);
```

Pendant le traitement de cette ligne par le trigger correspondant,

- `NEW.client_id` vaudra 12 ;
- `NEW.animal_id` vaudra 15 ;
- `NEW.date_reservation` vaudra `NOW()` ;
- `NEW.date_adoption` vaudra `NULL` ;
- `NEW.prix` vaudra 200.00 ;
- `NEW.paye` vaudra `FALSE` (0).

Les valeurs de `OLD` ne seront pas définies. Dans le cas d'une suppression, on aura exactement l'inverse.

Second exemple : la modification d'une ligne. On modifie la ligne que l'on vient d'insérer en exécutant la commande suivante :

```sql
UPDATE Adoption
SET paye = TRUE
WHERE client_id = 12 AND animal_id = 15;
```

Pendant le traitement de cette ligne par le trigger correspondant,

- `NEW.paye` vaudra `TRUE`, tandis que `OLD.paye` vaudra `FALSE`.
- Par contre les valeurs respectives de `NEW.animal_id`, `NEW.client_id`, `NEW.date_reservation`, `NEW.date_adoption` et `NEW.prix` seront les mêmes que `OLD.animal_id`, `OLD.client_id`, `OLD.date_reservation`, `OLD.date_adoption` et `OLD.prix`, puisque ces colonnes ne sont pas modifiées par la requête.

Dans le cas d'une insertion ou d'une modification, si un trigger peut potentiellement changer la valeur de `NEW.colonne`, il doit être exécuté avant l'événement (`BEFORE`). Sinon, la ligne aura déjà été insérée ou modifiée, et la modification de `NEW.colonne` n'aura plus aucune influence sur celle-ci.

V.7.2.4. Erreur déclenchée pendant un trigger

- Si un trigger `BEFORE` génère une erreur (non interceptée par un gestionnaire d'erreur), la requête ayant déclenché le trigger ne sera pas exécutée. Si l'événement devait également déclencher un trigger `AFTER`, il ne sera bien sûr pas non plus exécuté.
- Si un trigger `AFTER` génère une erreur, la requête ayant déclenché le trigger échouera.
- Dans le cas d'une table transactionnelle, si une erreur est déclenchée, un `ROLLBACK` sera fait. Dans le cas d'une table non-transactionnelle, tous les changements qui auraient été faits par le (ou les) trigger(s) avant le déclenchement de l'erreur persisteront.

V.7.3. Suppression des triggers

Encore une fois, la commande `DROP` permet de supprimer un trigger.

```
DROP TRIGGER nom_trigger;
```

Tout comme pour les procédures stockées, il n'est pas possible de modifier un trigger. Il faut le supprimer puis le recréer différemment.

Par ailleurs, si l'on supprime une table, on supprime également tous les triggers qui y sont attachés.

V.7.4. Exemples

V.7.4.1. Contraintes et vérification des données

V.7.4.1.1. Vérification du sexe des animaux

Dans notre table *Animal* se trouve la colonne *sexe*. Cette colonne accepte tout caractère, ou `NULL`. Or, seuls les caractères "M" et "F" ont du sens. Nous allons donc créer deux triggers, un pour l'insertion, l'autre pour la modification, qui vont empêcher qu'on donne un autre caractère que "M" ou "F" pour *sexe*.

Ces deux triggers devront se déclencher avant l'insertion et la modification. On aura donc :

```
-- Trigger déclenché par l'insertion
DELIMITER |
CREATE TRIGGER before_insert_animal BEFORE INSERT
ON Animal FOR EACH ROW
BEGIN
    -- Instructions
END |

-- Trigger déclenché par la modification
CREATE TRIGGER before_update_animal BEFORE UPDATE
ON Animal FOR EACH ROW
BEGIN
    -- Instructions
END |
DELIMITER ;
```

Il ne reste plus qu'à écrire le code du trigger, qui sera similaire pour les deux triggers. Et comme ce corps contiendra des instructions, il ne faut pas oublier de changer le délimiteur.

Le corps consistera en une simple structure conditionnelle, et définira un comportement à adopter si le sexe donné ne vaut ni ”M”, ni ”F”, ni `NULL`.

? Quel comportement adopter en cas de valeur erronée ?

Deux possibilités :

— on modifie la valeur du sexe, en le mettant à `NULL` par exemple ;
— on provoque une erreur, ce qui empêchera l'insertion/la modification.

Commençons par le plus simple : mettre le sexe à `NULL`.

```
DELIMITER |
CREATE TRIGGER before_update_animal BEFORE UPDATE
ON Animal FOR EACH ROW
BEGIN
    IF NEW.sexe IS NOT NULL   -- le sexe n'est ni NULL
    AND NEW.sexe != 'M'       -- ni "M"
    AND NEW.sexe != 'F'       -- ni "F"
      THEN
        SET NEW.sexe = NULL;
    END IF;
END |
DELIMITER ;
```

Test :

```
UPDATE Animal
SET sexe = 'A'
WHERE id = 20;  -- l'animal 20 est Balou, un mâle

SELECT id, sexe, date_naissance, nom
FROM Animal
WHERE id = 20;
```

id	sexe	date_naissance	nom
20	NULL	2007-04-24 12:45:00	Balou

Le sexe est bien `NULL`, le trigger a fonctionné.

Pour le second trigger, déclenché par l'insertion de lignes, on va implémenter le second comportement : on va déclencher une erreur, ce qui empêchera l'insertion, et affichera l'erreur.

? Mais comment déclencher une erreur ?

Contrairement à certains SGBD, MySQL ne dispose pas d'une commande permettant de déclencher une erreur personnalisée. La seule solution est donc de faire une requête dont on sait qu'elle va générer une erreur.

Exemple :

```
1 SELECT 1, 2 INTO @a;
```

```
1 ERROR 1222 (21000): The used SELECT statements have a different number of colu
```

Cependant, il serait quand même intéressant d'avoir un message d'erreur qui soit un peu explicite. Voici une manière d'obtenir un tel message : on crée une table *Erreur*, ayant deux colonnes, *id* et *erreur*. La colonne *id* est clé primaire, et *erreur* contient un texte court décrivant l'erreur. Un index `UNIQUE` est ajouté sur cette dernière colonne. On insère ensuite une ligne correspondant à l'erreur qu'on veut utiliser dans le trigger. Ensuite dans le corps du trigger, en cas de valeur erronée, on refait la même insertion. Cela déclenche une erreur de contrainte d'unicité, laquelle affiche le texte qu'on a essayé d'insérer dans *Erreur*.

```
1  -- Création de la table Erreur
2  CREATE TABLE Erreur (
3      id TINYINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
4      erreur VARCHAR(255) UNIQUE);
5
6  -- Insertion de l'erreur qui nous intéresse
7  INSERT INTO Erreur (erreur) VALUES
       ('Erreur : sexe doit valoir "M", "F" ou NULL.');
8
9  -- Création du trigger
10 DELIMITER |
11 CREATE TRIGGER before_insert_animal BEFORE INSERT
12 ON Animal FOR EACH ROW
13 BEGIN
14     IF NEW.sexe IS NOT NULL    -- le sexe n'est ni NULL
15     AND NEW.sexe != 'M'        -- ni "M"
16     AND NEW.sexe != 'F'        -- ni "F"
17       THEN
18         INSERT INTO Erreur (erreur) VALUES
               ('Erreur : sexe doit valoir "M", "F" ou NULL.');
19     END IF;
20 END |
21 DELIMITER ;
```

Test :

```
INSERT INTO Animal (nom, sexe, date_naissance, espece_id)
VALUES ('Babar', 'A', '2011-08-04 12:34', 3);
```

```
ERROR 1062 (23000): Duplicate entry 'Erreur : sexe doit valoir "M", "F" ou NULL
```

Et voilà, ce n'est pas parfait, mais au moins le message d'erreur permet de cerner d'où vient le problème. Et Babar n'a pas été inséré.

V.7.4.1.2. Vérification du booléen dans *Adoption*

Il est important de savoir si un client a payé ou non pour les animaux qu'il veut adopter. Il faut donc vérifier la valeur de ce qu'on insère dans la colonne *paye*, et refuser toute insertion/modification donnant une valeur différente de `TRUE` (1) ou `FALSE` (0). Les deux triggers à créer sont très similaires à ce que l'on a fait pour la colonne *sexe* d'*Animal*. Essayez donc de construire les requêtes vous-mêmes.

```
INSERT INTO Erreur (erreur) VALUES
    ('Erreur : paye doit valoir TRUE (1) ou FALSE (0).');

DELIMITER |
CREATE TRIGGER before_insert_adoption BEFORE INSERT
ON Adoption FOR EACH ROW
BEGIN
    IF NEW.paye != TRUE       -- ni TRUE
    AND NEW.paye != FALSE     -- ni FALSE
      THEN
        INSERT INTO Erreur (erreur) VALUES
            ('Erreur : paye doit valoir TRUE (1) ou FALSE (0).');
    END IF;
END |

CREATE TRIGGER before_update_adoption BEFORE UPDATE
ON Adoption FOR EACH ROW
BEGIN
    IF NEW.paye != TRUE       -- ni TRUE
    AND NEW.paye != FALSE     -- ni FALSE
      THEN
        INSERT INTO Erreur (erreur) VALUES
            ('Erreur : paye doit valoir TRUE (1) ou FALSE (0).');
    END IF;
END |
DELIMITER ;
```

Test :

```sql
UPDATE Adoption
SET paye = 3
WHERE client_id = 9;
```

```
ERROR 1062 (23000): Duplicate entry 'Erreur : paye doit valoir TRUE (1) ou FALS
```

V.7.4.1.3. Vérification de la date d'adoption

Il reste une petite chose à vérifier, et ce sera tout pour les vérifications de données : la date d'adoption ! En effet, celle-ci doit être postérieure ou égale à la date de réservation. Un client ne peut pas emporter chez lui un animal avant même d'avoir prévenu qu'il voulait l'adopter. À nouveau, essayez de faire le trigger vous-mêmes. Pour rappel, il ne peut exister qu'un seul trigger `BEFORE UPDATE` et un seul `BEFORE INSERT` pour chaque table.

```sql
INSERT INTO Erreur (erreur) VALUES
    ('Erreur : date_adoption doit être >= à date_reservation.');

DELIMITER |
DROP TRIGGER before_insert_adoption|
CREATE TRIGGER before_insert_adoption BEFORE INSERT
ON Adoption FOR EACH ROW
BEGIN
    IF NEW.paye != TRUE                                    -- On
        remet la vérification sur paye
    AND NEW.paye != FALSE
      THEN
        INSERT INTO Erreur (erreur) VALUES
            ('Erreur : paye doit valoir TRUE (1) ou FALSE (0).');

    ELSEIF NEW.date_adoption < NEW.date_reservation THEN    --
        Adoption avant réservation
        INSERT INTO Erreur (erreur) VALUES
            ('Erreur : date_adoption doit être >= à date_reservation.');
    END IF;
END |

DROP TRIGGER before_update_adoption|
CREATE TRIGGER before_update_adoption BEFORE UPDATE
ON Adoption FOR EACH ROW
BEGIN
```

```
22      IF NEW.paye != TRUE                                          -- On
            remet la vérification sur paye
23      AND NEW.paye != FALSE
24        THEN
25          INSERT INTO Erreur (erreur) VALUES
                ('Erreur : paye doit valoir TRUE (1) ou FALSE (0).');
26
27      ELSEIF NEW.date_adoption < NEW.date_reservation THEN    --
            Adoption avant réservation
28          INSERT INTO Erreur (erreur) VALUES
                ('Erreur : date_adoption doit être >= à date_reservation.');
29      END IF;
30  END |
31  DELIMITER ;
```

On aurait pu faire un second `IF` au lieu d'un `ELSEIF`, mais de toute façon, le trigger ne pourra déclencher qu'une erreur à la fois.

Test :

```
1  INSERT INTO Adoption (animal_id, client_id, date_reservation,
       date_adoption, prix, paye)
2  VALUES (10, 10, NOW(), NOW() - INTERVAL 2 DAY, 200.00, 0);
3
4  INSERT INTO Adoption (animal_id, client_id, date_reservation,
       date_adoption, prix, paye)
5  VALUES (10, 10, NOW(), NOW(), 200.00, 4);
```

```
1  ERROR 1062 (23000): Duplicate entry 'Erreur : date_adoption doit être >= à date
2
3  ERROR 1062 (23000): Duplicate entry 'Erreur : paye doit valoir TRUE (1) ou FALS
```

Les deux vérifications fonctionnent !

V.7.4.2. Mise à jour d'informations dépendant d'autres données

Pour l'instant, lorsque l'on a besoin de savoir quels animaux restent disponibles pour l'adoption, il faut faire une requête avec sous-requête.

```
1  SELECT id, nom, sexe, date_naissance, commentaires
2  FROM Animal
3  WHERE NOT EXISTS (
4      SELECT *
```

```
    FROM Adoption
    WHERE Animal.id = Adoption.animal_id
);
```

Mais une telle requête n'est pas particulièrement performante, et elle est relativement peu facile à lire. Les triggers peuvent nous permettre de stocker automatiquement une donnée permettant de savoir immédiatement si un animal est disponible ou non.

Pour cela, il suffit d'ajouter une colonne *disponible* à la table *Animal*, qui vaudra `FALSE` ou `TRUE`, et qui sera mise à jour grâce à trois triggers sur la table *Adoption*.

— À l'insertion d'une nouvelle adoption, il faut retirer l'animal adopté des animaux disponibles ;
— en cas de suppression, il faut faire le contraire ;
— en cas de modification d'une adoption, si l'animal adopté change, il faut remettre l'ancien parmi les animaux disponibles et retirer le nouveau.

```
-- Ajout de la colonne disponible
ALTER TABLE Animal ADD COLUMN disponible BOOLEAN DEFAULT TRUE;  -- À l'insertion, un animal est forcément disponible

-- Remplissage de la colonne
UPDATE Animal
SET disponible = FALSE
WHERE EXISTS (
    SELECT *
    FROM Adoption
    WHERE Animal.id = Adoption.animal_id
);

-- Création des trois triggers
DELIMITER |
CREATE TRIGGER after_insert_adoption AFTER INSERT
ON Adoption FOR EACH ROW
BEGIN
    UPDATE Animal
    SET disponible = FALSE
    WHERE id = NEW.animal_id;
END |

CREATE TRIGGER after_delete_adoption AFTER DELETE
ON Adoption FOR EACH ROW
BEGIN
    UPDATE Animal
    SET disponible = TRUE
    WHERE id = OLD.animal_id;
END |

```

```
31 CREATE TRIGGER after_update_adoption AFTER UPDATE
32 ON Adoption FOR EACH ROW
33 BEGIN
34     IF OLD.animal_id <> NEW.animal_id THEN
35         UPDATE Animal
36         SET disponible = TRUE
37         WHERE id = OLD.animal_id;
38 
39         UPDATE Animal
40         SET disponible = FALSE
41         WHERE id = NEW.animal_id;
42     END IF;
43 END |
44 DELIMITER ;
```

Test :

```
1 SELECT animal_id, nom, sexe, disponible, client_id
2 FROM Animal
3 INNER JOIN Adoption ON Adoption.animal_id = Animal.id
4 WHERE client_id = 9;
```

animal_id	nom	sexe	disponible	client_id
33	Caribou	M	0	9
54	Bubulle	M	0	9
55	Relou	M	0	9

```
1  DELETE FROM Adoption
       -- 54 doit redevenir disponible
2  WHERE animal_id = 54;
3  
4  UPDATE Adoption
5  SET animal_id = 38, prix = 985.00
       -- 38 doit devenir indisponible
6  WHERE animal_id = 33;
       -- et 33 redevenir disponible
7  
8  INSERT INTO Adoption (client_id, animal_id, date_reservation, prix,
       paye)
9  VALUES (9, 59, NOW(), 700.00, FALSE);
       -- 59 doit devenir indisponible
10 
11 SELECT Animal.id AS animal_id, nom, sexe, disponible, client_id
```

```
12 FROM Animal
13 LEFT JOIN Adoption ON Animal.id = Adoption.animal_id
14 WHERE Animal.id IN (33, 54, 55, 38, 59);
```

animal_id	nom	sexe	disponible	client_id
33	Caribou	M	1	NULL
38	Boule	F	0	9
54	Bubulle	M	1	NULL
55	Relou	M	0	9
59	Bavard	M	0	9

Désormais, pour savoir quels animaux sont disponibles, il suffira de faire la requête suivante :

```
1 SELECT *
2 FROM Animal
3 WHERE disponible = TRUE;
4
5 -- Ou même
6
7 SELECT *
8 FROM Animal
9 WHERE disponible;
```

V.7.4.3. Historisation

Voici deux exemples de systèmes d'historisation :

— l'un très basique, gardant simplement trace de l'insertion (date et utilisateur) et de la dernière modification (date et utilisateur), et se faisant directement dans la table concernée ;
— l'autre plus complet, qui garde une copie de chaque version antérieure des lignes dans une table dédiée, ainsi qu'une copie de la dernière version en cas de suppression.

V.7.4.3.1. Historisation basique

On va utiliser cette historisation pour la table *Race*. Libre à vous d'adapter ou de créer les triggers d'autres tables pour les historiser également de cette manière.

On ajoute donc quatre colonnes à la table. Ces colonnes seront toujours remplies automatiquement par les triggers.

```sql
-- On modifie la table Race
ALTER TABLE Race ADD COLUMN date_insertion DATETIME,
    -- date d'insertion
                 ADD COLUMN utilisateur_insertion VARCHAR(20),
                     -- utilisateur ayant inséré la ligne
                 ADD COLUMN date_modification DATETIME,
                     -- date de dernière modification
                 ADD COLUMN utilisateur_modification VARCHAR(20);
                     -- utilisateur ayant fait la dernière
                      modification

-- On remplit les colonnes
UPDATE Race
SET date_insertion = NOW() - INTERVAL 1 DAY,
    utilisateur_insertion = 'Test',
    date_modification = NOW()- INTERVAL 1 DAY,
    utilisateur_modification = 'Test';
```

J'ai mis artificiellement les dates d'insertion et de dernière modification à la veille d'aujourd'hui, et les utilisateurs pour l'insertion et la modification à "Test", afin d'avoir des données intéressantes lors des tests. Idéalement, ce type d'historisation doit bien sûr être mis en place dès la création de la table.

Occupons-nous maintenant des triggers. Il en faut sur l'insertion et sur la modification.

```sql
DELIMITER |
CREATE TRIGGER before_insert_race BEFORE INSERT
ON Race FOR EACH ROW
BEGIN
    SET NEW.date_insertion = NOW();
    SET NEW.utilisateur_insertion = CURRENT_USER();
    SET NEW.date_modification = NOW();
    SET NEW.utilisateur_modification = CURRENT_USER();
END |

CREATE TRIGGER before_update_race BEFORE UPDATE
ON Race FOR EACH ROW
BEGIN
    SET NEW.date_modification = NOW();
    SET NEW.utilisateur_modification = CURRENT_USER();
END |
DELIMITER ;
```

Les triggers sont très simples : ils mettent simplement à jour les colonnes d'historisation nécessaires ; ils doivent donc nécessairement être `BEFORE`.

Test :

```sql
INSERT INTO Race (nom, description, espece_id, prix)
VALUES ('Yorkshire terrier',
    'Chien de petite taille au pelage long et soyeux de couleur bleu et feu.',
    1, 700.00);

UPDATE Race
SET prix = 630.00
WHERE nom = 'Rottweiller' AND espece_id = 1;

SELECT nom, DATE(date_insertion) AS date_ins, utilisateur_insertion
    AS utilisateur_ins, DATE(date_modification) AS date_mod,
    utilisateur_modification AS utilisateur_mod
FROM Race
WHERE espece_id = 1;
```

nom	date_ins	utilisateur_ins	date_mod	utilisateur_mod
Berger allemand	2012-05-02	Test	2012-05-02	Test
Berger blanc suisse	2012-05-02	Test	2012-05-02	Test
Rottweiller	2012-05-02	Test	2012-05-03	sdz@localhost
Yorkshire terrier	2012-05-03	sdz@localhost	2012-05-03	sdz@localhost

V.7.4.3.2. Historisation complète

Nous allons mettre en place un système d'historisation complet pour la table *Animal*. Celle-ci ne change pas et contiendra la dernière version des données. Par contre, on va ajouter une table *Animal_histo*, qui contiendra les versions antérieures (quand il y en a) des données d'*Animal*.

```sql
CREATE TABLE Animal_histo (
  id SMALLINT(6) UNSIGNED NOT NULL,                -- Colonnes
      historisées
  sexe CHAR(1),
  date_naissance DATETIME NOT NULL,
  nom VARCHAR(30),
  commentaires TEXT,
  espece_id SMALLINT(6) UNSIGNED NOT NULL,
  race_id SMALLINT(6) UNSIGNED DEFAULT NULL,
  mere_id SMALLINT(6) UNSIGNED DEFAULT NULL,
  pere_id SMALLINT(6) UNSIGNED DEFAULT NULL,
  disponible BOOLEAN DEFAULT TRUE,

  date_histo DATETIME NOT NULL,                    -- Colonnes
      techniques
```

```
  utilisateur_histo VARCHAR(20) NOT NULL,
  evenement_histo CHAR(6) NOT NULL,
  PRIMARY KEY (id, date_histo)
) ENGINE=InnoDB;
```

Les colonnes *date_histo* et *utilisateur_histo* contiendront bien sûr la date à laquelle la ligne a été historisée, et l'utilisateur qui a provoqué cette historisation. Quant à la colonne *evenement_histo*, elle contiendra l'événement qui a déclenché le trigger (soit "DELETE", soit "UPDATE"). La clé primaire de cette table est le couple (*id*, *date_histo*).

Voici les triggers nécessaires. Cette fois, ils pourraient être soit `BEFORE`, soit `AFTER`. Cependant, aucun traitement ne concerne les nouvelles valeurs de la ligne modifiée (ni, a fortiori, de la ligne supprimée). Par conséquent, autant utiliser `AFTER`, cela évitera d'exécuter les instructions du trigger en cas d'erreur lors de la requête déclenchant celui-ci.

```
DELIMITER |
CREATE TRIGGER after_update_animal AFTER UPDATE
ON Animal FOR EACH ROW
BEGIN
    INSERT INTO Animal_histo (
        id,
        sexe,
        date_naissance,
        nom,
        commentaires,
        espece_id,
        race_id,
        mere_id,
        pere_id,
        disponible,

        date_histo,
        utilisateur_histo,
        evenement_histo)
    VALUES (
        OLD.id,
        OLD.sexe,
        OLD.date_naissance,
        OLD.nom,
        OLD.commentaires,
        OLD.espece_id,
        OLD.race_id,
        OLD.mere_id,
        OLD.pere_id,
        OLD.disponible,

        NOW(),
        CURRENT_USER(),
```

```
        'UPDATE');
END |

CREATE TRIGGER after_delete_animal AFTER DELETE
ON Animal FOR EACH ROW
BEGIN
    INSERT INTO Animal_histo (
        id,
        sexe,
        date_naissance,
        nom,
        commentaires,
        espece_id,
        race_id,
        mere_id,
        pere_id,
        disponible,

        date_histo,
        utilisateur_histo,
        evenement_histo)
    VALUES (
        OLD.id,
        OLD.sexe,
        OLD.date_naissance,
        OLD.nom,
        OLD.commentaires,
        OLD.espece_id,
        OLD.race_id,
        OLD.mere_id,
        OLD.pere_id,
        OLD.disponible,

        NOW(),
        CURRENT_USER(),
        'DELETE');
END |
DELIMITER ;
```

Cette fois, ce sont les valeurs avant modification/suppression qui nous intéressent, d'où l'utilisation de `OLD`.

Test :

```
UPDATE Animal
SET commentaires = 'Petit pour son âge'
WHERE id = 10;

```

```sql
DELETE FROM Animal
WHERE id = 47;

SELECT id, sexe, date_naissance, nom, commentaires, espece_id
FROM Animal
WHERE id IN (10, 47);

SELECT id, nom, date_histo, utilisateur_histo, evenement_histo
FROM Animal_histo;
```

id	sexe	date_naissance	nom	commentaires	espece_i
10	M	2010-07-21 15:41:00	Bobo	Petit pour son âge	1

id	nom	date_histo	utilisateur_histo	evenement_histo
10	Bobo	2012-05-03 21:51:12	sdz@localhost	UPDATE
47	Scroupy	2012-05-03 21:51:12	sdz@localhost	DELETE

V.7.4.3.3. Quelques remarques sur l'historisation

Les deux systèmes d'historisation montrés dans ce cours ne sont que deux possibilités parmi des dizaines. Si vous pensez avoir besoin d'un système de ce type, prenez le temps de réfléchir, et de vous renseigner sur les diverses possibilités qui s'offrent à vous. Dans certains systèmes, on combine les deux historisations que j'ai présentées. Parfois, on ne conserve pas les lignes supprimées dans la table d'historisation, mais on utilise plutôt un système d'archive, séparé de l'historisation. Au-delà du modèle d'historisation que vous choisirez, les détails sont également modifiables. Voulez-vous garder toutes les versions des données, ou les garder seulement pour une certaine période de temps ? Voulez-vous enregistrer l'utilisateur SQL ou plutôt des utilisateurs créés pour votre application, découplés des utilisateurs SQL ? Ne restez pas bloqués sur les exemples montrés dans ce cours (que ce soit pour l'historisation ou le reste), le monde est vaste !

V.7.5. Restrictions

Les restrictions sur les triggers sont malheureusement trop importantes pour qu'on puisse se permettre de ne pas les mentionner. On peut espérer qu'une partie de ces restrictions soit levée dans une prochaine version de MySQL, mais en attendant, il est nécessaire d'avoir celles-ci en tête. Voici donc les principales.

V.7.5.0.1. Commandes interdites

Il est impossible de travailler avec des transactions à l'intérieur d'un trigger. Cette restriction est nécessaire, puisque la requête ayant provoqué l'exécution du trigger pourrait très bien se trouver elle-même à l'intérieur d'une transaction. Auquel cas, toute commande `START TRANSACTION`, `COMMIT` ou `ROLLBACK` interagirait avec cette transaction, de manière intempestive.

Les requêtes préparées ne peuvent pas non plus être utilisées.

Enfin, on ne peut pas appeler n'importe quelle procédure à partir d'un trigger.

— Les procédures appelées par un trigger **ne peuvent pas envoyer d'informations au client MySQL**. Par exemple, elles ne peuvent pas exécuter un simple `SELECT`, qui produit un affichage dans le client (un `SELECT...INTO` par contre est permis). Elles peuvent toutefois renvoyer des informations au trigger grâce à des paramètres `OUT` ou `INOUT`.
— Les procédures appelées ne peuvent utiliser ni les transactions (`START TRANSACTION`, `COMMIT` ou `ROLLBACK`) ni les requêtes préparées. C'est-à-dire qu'**elles doivent respecter les restrictions des triggers**.

V.7.5.0.2. Tables utilisées par la requête

Comme mentionné auparavant, il est **impossible de modifier les données d'une table utilisée par la requête ayant déclenché le trigger** à l'intérieur de celui-ci.

Cette restriction est importante, et peut remettre en question l'utilisation de certains triggers.

Exemple : le trigger `AFTER INSERT ON Adoption` modifie les données de la table *Animal*. Si l'on exécute la requête suivante, cela posera problème.

```
1 INSERT INTO Adoption (animal_id, client_id, date_reservation, prix,
      paye)
2 SELECT Animal.id, 4, NOW(), COALESCE(Race.prix, Espece.prix), FALSE
3 FROM Animal
4 INNER JOIN Espece ON Espece.id = Animal.espece_id
5 LEFT JOIN Race ON Race.id = Animal.race_id
6 WHERE Animal.nom = 'Boucan' AND Animal.espece_id = 2;
```

```
1 ERROR 1442 (HY000): Can't update table 'animal' in stored function/trigger beca
```

Le trigger échoue puisque la table *Animal* est utilisée par la requête `INSERT` qui le déclenche. L'insertion elle-même est donc finalement annulée.

V.7.5.0.3. Clés étrangères

Une suppression ou modification de données déclenchée par une clé étrangère ne provoquera pas l'exécution du trigger correspondant. Par exemple, la colonne *Animal.race_id* possède une clé étrangère, qui référence la colonne *Race.id*. Cette clé étrangère a été définie avec l'option `ON DELETE SET NULL`. Donc en cas de suppression d'une race, tous les animaux de cette race seront modifiés, et leur *race_id* changée en `NULL`. Il s'agit donc d'une modification de données. Mais comme cette modification a été déclenchée par une contrainte de clé étrangère, les éventuels triggers `BEFORE UPDATE` et `AFTER UPDATE` de la table *Animal* ne seront pas déclenchés.

En cas d'utilisation de triggers sur des tables présentant des clés étrangères avec ces options, il vaut donc mieux supprimer celles-ci et déplacer ce comportement dans des triggers. Une autre solution est de ne pas utiliser les triggers sur les tables concernées. Vous pouvez alors remplacer les triggers par l'utilisation de procédures stockées et/ou de transactions.

?

Qu'avons-nous comme clés étrangères dans nos tables ?

- *Race* : `CONSTRAINT fk_race_espece_id FOREIGN KEY (espece_id) REFERENCES Espece (id) ON DELETE CASCADE;`
- *Animal* : `CONSTRAINT fk_race_id FOREIGN KEY (race_id) REFERENCES Race (id) ON DELETE SET NULL;`
- *Animal* : `CONSTRAINT fk_espece_id FOREIGN KEY (espece_id) REFERENCES Espece (id);`
- *Animal* : `CONSTRAINT fk_mere_id FOREIGN KEY (mere_id) REFERENCES Animal (id) ON DELETE SET NULL;`
- *Animal* : `CONSTRAINT fk_pere_id FOREIGN KEY (pere_id) REFERENCES Animal (id) ON DELETE SET NULL;`

Quatre d'entre elles pourraient donc poser problème. Quatre, sur cinq ! Ce n'est donc pas anodin comme restriction !

On va donc modifier nos clés étrangères pour qu'elles reprennent leur comportement par défaut. Il faudra ensuite créer (ou recréer) quelques triggers pour reproduire le comportement que l'on avait défini. À ceci près que la restriction sur la modification des données d'une table utilisée par l'événement déclencheur fait qu'on ne pourra pas reproduire certains comportements. On ne pourra pas mettre à `NULL` les colonnes *pere_id* et *mere_id* de la table *Animal* en cas de suppression de l'animal de référence.

Voici les commandes :

```
1 -- On supprime les clés
2 ALTER TABLE Race DROP FOREIGN KEY fk_race_espece_id;
3 ALTER TABLE Animal DROP FOREIGN KEY fk_race_id,
4                    DROP FOREIGN KEY fk_mere_id,
5                    DROP FOREIGN KEY fk_pere_id;
6
```

```sql
-- On les recrée sans option
ALTER TABLE Race ADD CONSTRAINT fk_race_espece_id FOREIGN KEY
    (espece_id) REFERENCES Espece (id);
ALTER TABLE Animal ADD CONSTRAINT fk_race_id FOREIGN KEY (race_id)
    REFERENCES Race (id),
                   ADD CONSTRAINT fk_mere_id FOREIGN KEY (mere_id)
                       REFERENCES Animal (id),
                   ADD CONSTRAINT fk_pere_id FOREIGN KEY (pere_id)
                       REFERENCES Animal (id);

-- Trigger sur Race
DELIMITER |
CREATE TRIGGER before_delete_race BEFORE DELETE
ON Race FOR EACH ROW
BEGIN
    UPDATE Animal
    SET race_id = NULL
    WHERE race_id = OLD.id;
END|

-- Trigger sur Espece
CREATE TRIGGER before_delete_espece BEFORE DELETE
ON Espece FOR EACH ROW
BEGIN
    DELETE FROM Race
    WHERE espece_id = OLD.id;
END |
DELIMITER ;
```

V.7.5.1. En résumé

- Un trigger est un objet **stocké dans la base de données**, à la manière d'une table ou d'une procédure stockée. La seule différence est qu'un trigger est **lié à une table**, donc en cas de suppression d'une table, les triggers liés à celle-ci sont supprimés également
- Un trigger **définit une ou plusieurs instructions**, dont l'exécution est **déclenchée par une insertion, une modification ou une suppression** de données dans la table à laquelle le trigger est lié.
- Les instructions du trigger peuvent être exécutées **avant la requête ayant déclenché celui-ci, ou après**. Ce comportement est à définir à la création du trigger.
- Une table ne peut posséder qu'un seul trigger par combinaison événement/moment (`BEFORE UPDATE`, `AFTER DELETE`,…)
- Les triggers sous MySQL sont soumis à d'importantes (et potentiellement très gênantes) **restrictions**.

Sécuriser une base de données et automatiser les traitements ne se limite bien sûr pas à ce que

nous venons de voir. Les deux prochaines parties vous donneront de nouveaux outils pour avoir une base de données bien construite, sûre et efficace. Cependant, tout ne pourra pas être abordé dans ce cours, donc n'hésitez pas à poursuivre votre apprentissage.