TP3 : DOCUMENT A (AVANT SEANCE DE TP)

Programmation dans les bases de données (Langage PL/SQL):

Le langage SQL (**Structured Query Language**) est le langage de requêtes basique pour bon nombre de SGBDR (système gestion de bases de données relationnels) tel que :

- **MySQL** (un SGBDR open-source libre de doit)

- **PostgreSQL** (un SGBDR open-source libre de doit)

- **SQL Server** (produit Microsoft sous licence et très utilisé comme SGBDR)

- **ORACLE SQL** (produit sous licence et très utilisé dans les grandes entreprises)

- Ect.

Cependant, le langage SQL est un langage déclaratif non procédural permettant d'exprimer uniquement des requêtes dans un langage relativement simple. Il n'intègre aucune structure de contrôle, de traitements itératifs (permettant par exemple d'exécuter une boucle Tant que) ou de déclaration de variables.
Le PL/SQL (Procedural Language / Structured Query Language) c'est le langage de programmation, avec des variables, des boucles, des tests, etc. et du SQL. Ce langage (PL/SQL) permet de définir un ensemble de commandes contenues dans ce que l'on appelle un "bloc" PL/SQL, ce qui offre de nombreuses possibilités pour un SGBDR.
Comme c'est le cas pour le langage SQL, il existe certaines nuances (certaines différences) du langage PL/SQL d'un SGBDR à un autre (malgré que globalement ce langage reste commun aux SGBDR MySQL, PostgreSQL, SQL Server et ORACLE SQL).

Mais avant de d'aborder un quelconque autre point dans le présent document il est nécessaire d'aborder ici quelques éléments du langage PL-SQL utile pour l'écriture des blocs d'instructions **dans MySQL**.

Syntaxe PL-SQL

Règles du langage :

- Pas de différence entre majuscule et minuscule (IF ou If ou if c'est la même chose) même s'il est préférable et conseillé d'écrire les commandes en majuscule.
- La fin d'une instruction se termine toujours par un « ; »

Déclaration de variables :

DECLARE <Nom_Variable> Type_Variable ;

Exemple 1:

```
DECLARE   my_int  INT;
```

Exemple 2:

```
DECLARE   my_num   NUMERIC(8,2);
```

Exemple 3:

```
DECLARE my_text TEXT;
```

Exemple 4:

```
DECLARE my_date DATE ;
```

Exemple 5:

```
DECLARE my_varchar VARCHAR(30) ;
```

Les types de variable sont ceux utilisés dans le SGBDR (TINYINT, SMALLINT, INT ou INTEGER, BIGINT, TEXT, DATE, VARCHAR(255), etc.).

On peut déclarer plusieurs variables à la fois sans répéter à chaque fois le mot DECLARE :

```
DECLARE <Nom_Variable1>  Type_Variable1 ;
        <Nom_Variable2>  Type_Variable2 ;
                .    .    .
        <Nom_VariableN>  Type_VariableN ;
```

On peut faire des affectations au moment de la déclaration d'une variable (initialisation de variable) :

DECLARE <Nom_Variable> Type_Variable DEFAULT value;

Exemple 1:

```
DECLARE my_pi  FLOAT DEFAULT 3.1415926;
```

Exemple 2:

```
DECLARE my_date DATE DEFAULT '2008-02-01';
```

Exemple 3:

```
DECLARE my_varchar VARCHAR(30) DEFAULT 'bonjour';
```

Bloc exécutable :

C'est un bloc d'instructions qui est entre :

```
BEGIN
....
--Actions (instructions) séparés par des « ; »
.....
END;
```

Commentaires :

```
-- pour un commentaire sur une seule ligne.
/* pour un commentaire sur
   Plusieurs lignes */
```

Affectation dans le programme (cette syntaxe diffère d'un SGBDR à un autre):

```
SET<Nom_Variable> := valeur ;
```

Exemples:

```
SET my_int :=20;
SET my_bigint := POWER(my_int,3);
SET my_date := CURRENT_DATE;
```

Structures conditionnelles :

1/- Condition **si/alors/FinSi**

```
IF (condition)
THEN actions ;
END IF ;
```

2/- Condition **si/alors/sinon/FinSi**

```
IF (condition)
THEN actions ;
ELSE actions ;
END IF ;
```

3/- Condition **Cas / vaut/FinCas**

```
CASE variable
WHEN valeur1 THEN action1 ;
WHEN valeur2 THEN action2 ;
. . .
WHEN valeurN THEN actionN ;
END CASE ;
```

Testes dans les conditions :

>, <, <=, >=, BETWEEN, NOT BETWEEN, IN, NOT IN, = (égalité et non pas une affectation, l'affectation c'est :=), <>, !=, LIKE, IS NULL, IS NOT NULL.

Opérateurs mathématiques :

+, -, *, /, DIV, %

Opérateurs logiques :

AND, OR, XOR

Fonctions de chaîne de caractères :

SUBSTRING, LENGTH, CONCAT, LOWER, UPPER, etc.

Fonctions numériques :

ABS, POWER, SQRT, CEILING, GREATEST, MOD, RAND, etc.

Fonctions de dates et d'heures :
CURRENT_DATE, CURRENT_TIME, TO_DAYS, FROM_DAYS, DATE_SUB, etc.
Structures itératives :
Tant que :
WHILE (condition) DO actions; END WHILE;
Répétez jusqu'a :
REPEAT actions ; UNTIL (condition) END REPEAT;
Pour :
FOR <Variable_Name> IN Valeur_Départ . . Valeur_Arrivée LOOP actions; END LOOP;
Exemple : FOR X IN 0. . 15 LOOP actions; END LOOP;

Le PL/SQL est un programme structuré en blocs, chaque bloc peut être exécuté :

- Directement comme une commande SQL dans le SGBDR (dans la fenêtre d'exécution des commandes du SGBDR)
- Automatiquement sur un événement déclencheur précis (exécution par usage de Trigger). (Lisez la suite et vous saurez c'est quoi un trigger)
- Dans une procédure stockée (PSM : Persistent Stored Module) appelé à partir d'un programme extérieur. (Procédure stockées et fonctions stockées seront abordées dans la séance de TP INCHALAH)

Donc, nous avons au total 3 manières d'exécuter un programme PL/SQL.

Les commandes d'insertion, de modification et de suppression de tuples dans une table (INSERT / UPDATE / DELETE):

- La commande d'insertion d'un tuple dans une table :
  La commande INSERT permet de faire des insertions de lignes (tuples ou instances) dans une table.
  INSERT INTO <NomTable> (attribut1, attribut2, …., AttributN) VALUES
  ('Valeur1', 'Valeur2', ….. , 'ValeurN');
  Pensez à mettre chaque valeur entre des guillemets simples ou des guillemets doubles (pas besoin d'utiliser des guillemets pour les nombres entiers et décimaux, flottant ou double, par contre vous avez pour obligation de les mettre pour les autres types). Pour les valeurs auto-incrémentales, mettre la valeur à « NULL » (sans guillemets). Les date doivent être écrite sous la forme 'Année-mois-jours' comme par exemple '2020-05-17'. Pour les nombres décimaux, la séparation entre la partie réel et la partie décimale se fait par « . » et non pas par « ; » (exemple : '1.2' et non pas '1,2'). Il est toujours possible de faire des insertions multiples dans un même INSERT :
  INSERT INTO NomTable (attribut1, attribut2, …., AttributN)
  VALUES
  ('Valeur11', 'Valeur12', ….. , 'Valeur1N'),
  ('Valeur21', 'Valeur22', ….. , 'Valeur2N'),
  ……
  ('ValeurM1', 'ValeurM2', ….. , 'ValeurMN') ;

- La commande de modification d'un tuple dans une table :
  La commande UPDATE permet d'effectuer des modifications sur des lignes existantes d'une table. Très souvent cette commande est utilisée avec WHERE pour spécifier sur quelles lignes doivent porter la ou les modifications.
  La syntaxe d'une requête UPDATE est la suivante :

```
UPDATE NomTable
SET nom_colonne1 = 'nouvelle valeur1' --le mot SET veut dire définir
WHERE condition ;
```

  Cette syntaxe permet d'attribuer une nouvelle valeur 1 à la colonne nom_colonne_1 pour les lignes qui **respectent la condition stipulée** avec WHERE. Il est aussi possible d'attribuer la même valeur à la colonne nom_colonne_1 pour toutes les lignes de la table si la condition WHERE n'était pas utilisée.
  Exemple d'un UPDATE :

```
UPDATE Etudiant
SET  Note = '12.5'
WHERE idEtudiant = 2120548 ;
```

  A noter, pour spécifier en une seule fois plusieurs modification, il faut séparer les attributions de valeurs par des virgules. Ainsi la syntaxe deviendrait la suivante :

```
UPDATE table
SET colonne_1 = 'valeur 1',
      colonne_2 = 'valeur 2',
       ....
      colonne_N = 'valeur N'
WHERE condition ;
```

  Exemple :

```
UPDATE Etudiant
SET  Note = '16.5',
       Mention = 'Bien',
       Remarque = 'Bon étudiant'
WHERE idEtudiant = 2120548 ;
```

- La commande de suppression d'un tuple dans une table :
  La commande DELETE permet de supprimer des lignes dans une table qui respecte la condition stipulée par un WHERE (En utilisant le WHERE il est possible de sélectionner les lignes concernées qui seront supprimées par la commande DELETE). La syntaxe pour supprimer des lignes est la suivante :

```
DELETE FROM NomTable
WHERE condition ;
```

  Exemple :

```
DELETE FROM Etudiant
WHERE idEtudiant = 2120548 ;
```

Notez que s'il n'y a pas de condition WHERE alors toutes les lignes seront supprimées et la table sera alors vide.

Les trigger / Les déclencheurs SQL

Les triggers (ou déclencheurs en français) sont des objets (des scripts) stockés dans le SGBDR (système de gestion de base de données relationnelles) qui sont attachés à une table et qui se déclenchent suite à un événement produit sur la table à laquelle ils sont rattachés. Le déclenchement d'un **trigger** permet l'exécution d'une instruction, ou d'un bloc d'instructions (plusieurs instructions). Contrairement à l'utilisation des fonctions qui s'exécutent par un appel de fonctions, il n'est pas possible d'appeler un trigger. Un trigger est automatiquement déclenché par un événement qui se produit sur la table à laquelle il est rattaché. Cet événement peut être :

- une insertion d'un ou de plusieurs tuples dans la table (requête INSERT) ;
- la suppression d'un ou de plusieurs tuples dans la table (requête DELETE) ;
- la modification d'un ou de plusieurs tuples dans la table (requête UPDATE).

Ainsi, un trigger exécute un traitement pour chaque ligne insérée, modifiée ou supprimée. Une fois le trigger déclenché, ses instructions peuvent être exécutées soit **juste avant** (BEFORE) l'exécution de la commande d'insertion/suppression/ modification, **soit juste après** (AFTER).

S'il y a plus d'une instruction à exécuter dans un trigger, il faut mettre ces instructions à l'intérieur d'un bloc d'instructions qui commence par un « BEGIN » et se termine par un « END ».

À quoi sert un trigger ?

Un trigger est utile dans de nombreuses situations (en voici deux):

Contraintes de vérification de données

MySQL ne permet pas de mettre en œuvre des contraintes visant à limiter les valeurs acceptées par une colonne (limiter une colonne à un ensemble de valeurs prédéfinie, par exemple). Avec des triggers se déclenchant avant l'INSERT d'un tuple et avant l'UPDATE d'un tuple, 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 à ce genre de problèmes de vérification des données par l'usage des Triggers.

Mise à jour de données

Lorsque certaines manipulations sur une tables nécessite la mise à jour d'autres données sur dans d'autres tables, utiliser les triggers est un bon moyen pour mettre en liens des données appartenant à des tables différentes.

Syntaxe de création des triggers

Pour créer un trigger, on utilise la commande suivante :

CREATE TRIGGER Nom_trigger
Moment_ Déclenchement_trigger Evenement_ Déclenchement_trigger
ON Nom_table
FOR EACH ROW
Corps_trigger ;

La signification de chaque champ de la syntaxe de création d'un trigger est la suivante:

- CREATE TRIGGER Nom_trigger : les triggers ont un nom.
- Moment_ Déclenchement_trigger Evenement_ Déclenchement_trigger : ces deux informations servent à définir quand (es ce que le trigger se déclenche **avant** « BEFORE » **la commande** d'insertion/suppression/modification **ou après** « AFTER » ?) et comment le trigger est déclenché (es ce que le trigger se déclenche suite à une commande d'insertion INSERT / ou une commande de suppression DELETE / ou une commande de modification UPDATE ?).
- ON Nom_table : un trigger est attaché à une table précise, il est créé pour cette table et il est à l'écoute (en attente) d'un événement précis qui peut être effectué sur cette table.
- FOR EACH ROW : signifie littéralement "pour chaque ligne", sous-entendu "pour chaque ligne insérée/supprimée/modifiée" selon ce qui a déclenché le trigger. Pour bien comprendre l'utilité de cette instruction, nous prenons l'exemple de déclanchement d'un trigger suite à une opération d'insertion. Nous avons vu précédemment que la commande INSERT permet de faire une insertion multiple (insérer plusieurs tuples à la fois dans une table par une seule commande INSERT). Dans le cas de d'insertion multiple dans une table par une seule commande INSERT, l'absence de FOR EACH ROW implique que le déclanchement du trigger se produira uniquement pour le premier tuple de la liste de tuples à insérer alors que plusieurs tuples seront insérer dans la même commande INSERT.
- Corps_trigger : c'est le contenu du trigger. C'est ici que nous allons spécifier les actions qui seront faite par le trigger. Il peut s'agir soit d'une seule instruction, soit d'un bloc d'instructions.

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) dans la table attachée au trigger ;

- la suppression de lignes (DELETE) dans la table attachée au trigger.

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

Déclenchement du trigger Avant ou Après

Lorsqu'un trigger est déclenché, ses instructions peuvent être exécutées à deux moments différents : soit juste avant que la commande INSERT/UPDATE/DELETE 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 les lignes de la table seront modifiées :

```
CREATE TRIGGER Avant_Modification_TableA
BEFORE  UPDATE
ON TableA
FOR EACH ROW
Corps_trigger;
```

De même, si vous avez un trigger AFTER INSERT sur la table B, l'exécution d'une requête INSERT sur cette table va déclencher l'exécution des instructions du trigger **après** l'opération d'insertion de lignes dans la table B :

```
CREATE TRIGGER Aprés_Insertion_TableB
AFTER INSERT
ON TableB
FOR EACH ROW
Corps_trigger;
```

Règle et convention

Il ne peut exister qu'un seul trigger par combinaison de couple « moment_trigger/evenement_trigger » par table. Donc un seul trigger BEFORE UPDATE par table, et un seul AFTER DELETE par table, 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 et que je vous encourage à suivre : nom_trigger = moment_evenement_table. Donc le trigger BEFORE UPDATE ON TableA aura pour nom : before_update_TableA ou Avant_Modification_TableA.

Les mots clés 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 soit supprimer par l'événement déclencheur.
NEW représente les nouvelles valeurs des colonnes de la ligne traitée qu'on veut soit insérer par une commande INSERT, soit les mettre à la place d'autres valeurs par une commande UPDATE.
Il n'y a que dans le cas d'un trigger UPDATE que deux mots-clés OLD et NEW coexistent (sont utilisé les deux). Lors d'une insertion, OLD n'existe pas, puisque la ligne n'existe pas avant l'événement la commande d'insertion. Dans le cas d'une suppression, c'est NEW qui n'existe pas, puisque la ligne n'existera plus après la commande de suppression.

Exemple: soit les deux relations suivantes :

ETUDIANT (**EtudiantId**, SpécialitéId_E, Date_inscription, Frais_inscription, Payement_frais_ inscription)
Spécialité (**SpécialitéId**, Désignation, Date_Création)

Exécutons la commande suivante (l'insertion d'une ligne dans la table étudiant) :

```
INSERT INTO ETUDIANT (EtudiantId, SpécialitéId_E, Date_inscription, Frais_inscription, Payement_frais_ inscription) VALUES ('1612', '3', 'NOW()', '220', 'FALSE');
```

Pendant le traitement de cette ligne par le trigger correspondant :

- NEW.EtudiantId vaudra 1612;
- NEW.SpécialitéId_E vaudra 3;
- NEW.Date_inscription vaudra NOW() (c'est-à-dire la date système actuelle au moment de l'inscription de l'étudiant dans la spécialité);
- NEW.Frais_inscription vaudra 220 ;
- NEW.Payement_frais_inscription vaudra FALSE (donc 0, c'est-à-dire frais d'inscription non réglé par l'étudiant le jour de son inscription dans la spécialité).

Dans la commande d'insertion, les valeurs de NEW sont définies et connus par contre les valeurs d'OLD ne seront pas connus et ne sont pas définies. Si nous prenons le cas d'une suppression, on aura exactement l'inverse (c'est-à-dire des valeurs d'OLD seront connus et définies, par contre les valeurs de NEW sont non définies).

Si maintenant on modifie la ligne qu'on vient d'insérer en exécutant la commande suivante :

```
UPDATE ETUDIANT
SET Payement_frais_inscription = 'TRUE'
WHERE EtudiantId = 1612 AND SpécialitéId_E = 3;
```

Pendant le traitement de cette ligne par le trigger correspondant :

NEW.Payement_frais_inscription vaudra TRUE (c'est la valeur obtenue après l'opération de modification), tandis que OLD.Payement_frais_inscription vaudra FALSE (c'est la valeur initiale avant l'opération de modification).

Par contre, les valeurs respectives de NEW.EtudiantId, NEW.SpécialitéId_E, NEW.Date_inscription et NEW.Frais_inscription seront les mêmes que OLD. EtudiantId, OLD. SpécialitéId_E, OLD.Date_inscription et OLD.Frais_inscription, puisque ces colonnes ne sont pas modifiées par la requête.

Gestion des triggers

La gestion des Triggers englobent deux points (création et suppression):

La création des triggers :

L'exécution de la commande CREATE TRIGGER permet de créer un trigger :

```
CREATE TRIGGER trigger_name
trigger_time [BEFORE/AFTER] trigger_event[INSERT/ UPDATE/ DELETE]
ON table_name
FOR EACH ROW
Commande  or   BEGIN
                  Commandes
                  END ;
```

La Suppression des triggers :

La commande DROP TRIGGER permet de supprimer un trigger : DROP TRIGGER Nom_trigger;

Par contre, il n'est pas possible de modifier un trigger (il **n'existe pas de commande** ALTER TRIGGER Nom_trigger dans MySQL, vous pouvez la trouvez dans d'autre SQBDR tel que SQL Server mais pas dans MySQL). Pour modifier un Trigger, il faut le supprimer puis le recréer différemment. Notez également que suppression d'une table engendre également la suppression de tous les triggers qui y sont attachés.

Exemples d'utilisation d'un trigger :

Vérification des données : (Nous prenons l'exemple de vérification des valeurs d'un attribut appartenant à une table)

Supposons que la table Etudiant comporte un nouvel attribut «Genre» qui détermine si l'étudiant est un homme (H) ou une femme (F) :
ETUDIANT (**EtudiantId**, SpécialitéId_E, Date_inscription, Frais_inscription, Payement_frais_inscription, Genre)
SPECIALITE (**SpécialitéId**, Désignation, Date_Création)

Sans vérification, cette colonne «Genre» accepte tout caractère. Or, seuls les caractères "H", "F" ou NULL ont du sens pour cet attribut. Nous allons donc créer deux triggers, un déclenché par une commande d'insertion INSERT sur la table Etudiant, un autre déclenché par une commande de modification UPDATE sur la table Etudiant, qui vont empêcher que l'on donne un autre caractère que "H" ou "F" pour l'attribut «Genre» dans une opération d'insertion ou de modification sur la table Etudiant.

Ces deux triggers devront se déclencher avant l'insertion (si Trigger se déclenche après l'insertion, le Trigger ne servira à rien puisque les valeurs seront mises dans la table) ou la modification (si Trigger se déclenche après la modification, sa sera trop tard puisque les valeurs seront mises dans la base). On aura donc :

```
-- Trigger 1 déclenché par l'insertion
DELIMITER ##
CREATE TRIGGER Avant_Insertion_Etudiant
BEFORE INSERT
ON ETUDIANT
FOR EACH ROW
BEGIN
   -- Bloc d'instructions du trigger
END ##

-- Trigger 2 déclenché par la modification
DELIMITER $$
CREATE TRIGGER Avant_Modification_Etudiant
BEFORE UPDATE
ON ETUDIANT
FOR EACH ROW
BEGIN
   -- Bloc d'instructions du trigger
END $$
DELIMITER ;
```

Le caractère « ; » termine toute instruction SQL (c'est le cas de toutes les instructions SQL). Or, cela pose problème, puisque l'usage du caractère « ; » à l'intérieur du corps du Trigger pour indiquer la fin d'une l'instruction SQL sera interpréter comme l'endroit ou le Trigger s'arrête, se qui engendre confusion. Cette confusion déclenche une erreur puisqu'en réalité le Trigger n'est pas terminée (le Trigger n'est pas complet !). Comment faire pour écrire des instructions SQL à l'intérieur d'une Trigger alors ? Il suffit d'utiliser la commande DELIMITER pour proposer un autre caractère ou suites de caractères (un ensemble de caractères) pour changer le délimiteur du Trigger. Vous pouvez utiliser le ou les caractères de votre choix comme délimiteur pour le **trigger**. Ainsi DELIMITER $$ indique que la délimitation du **trigger** est «$$ ». Les deux délimiteurs les plus couramment utilisés sont: DELIMITER // et DELIMITER |. Notez que les instructions SQL

à l'intérieur du Trigger seront toujours délimitées par « ; » (Ils ne sont pas influencées par la commande DELIMITER). A la fin des Triggers pensez à remettre votre délimiteur à sa valeur initiale « ; » (par l'instruction DELIMITER ;).

Il ne reste plus qu'à écrire le code du trigger, qui sera, dans ce cas de figure et par pure hasard, similaire pour les deux triggers. Le corps des deux triggers est une simple structure conditionnelle qui définira le comportement à adopter si le genre ne vaut ni "H", ni "F", ni NULL.

Quel comportement adopter en cas de valeur erronée ?
Deux possibilités :

- Modifier la valeur du «Genre», en la mettant à NULL, par exemple ;
- Provoquer une erreur, ce qui empêchera l'insertion/la modification de s'exécuter (cette deuxième option est très utilisé dans le cas ou il est impossible d'imposer une valeur par défaut).

Commençons par l'issue la plus simple : mettre le «Genre» à NULL.

```
-- Trigger 1 déclenché par l'insertion
DELIMITER $$
CREATE TRIGGER Avant_Insertion_Etudiant
BEFORE INSERT
ON ETUDIANT
FOR EACH ROW
BEGIN
    IF NEW.Genre IS NOT NULL    -- le Genre n'est ni NULL
    AND NEW.Genre != 'H'        -- ni Homme
    AND NEW.Genre != 'F'        -- ni Femme
    THEN
    SET NEW.Genre = NULL;       -- le Genre reçoit la valeur NULL
    END IF;
END $$
DELIMITER ;

-- Trigger 2 déclenché par la modification
DELIMITER $$
CREATE TRIGGER Avant_Modification_Etudiant
BEFORE UPDATE
ON ETUDIANT
FOR EACH ROW
BEGIN
    IF NEW.Genre IS NOT NULL    -- le Genre n'est ni NULL
    AND NEW.Genre != 'H'        -- ni Homme
    AND NEW.Genre != 'F'        -- ni Femme
```

```
        THEN
        SET NEW.Genre = NULL;          -- le Genre reçoit la valeur NULL
        END IF;
END $$
DELIMITER ;
```

Tester si les deux triggers fonctionnent correctement avec les commandes suivantes (n'oublier pas d'ajouter l'attribut Genre à la table étudiant avant de faire ce teste):

```
INSERT INTO ETUDIANT (EtudiantId, SpécialitéId_E, Date_inscription, Frais_inscription, Payement_frais_inscription, Genre) VALUES ('1612', '3', NOW(), '220', 'FALSE', 'H');
```
Cette commande d'insertion se déroule normalement.

```
INSERT INTO ETUDIANT (EtudiantId, SpécialitéId_E, Date_inscription, Frais_inscription, Payement_frais_inscription, Genre) VALUES ('1522', '4', NOW(), '220', 'FALSE', 'A');
```
Cette commande d'insertion se déroule normalement. Par contre la valeur de l'attribut «Genre» pour cet étudiant sera automatiquement mise à NULL (si c'est le cas le trigger Avant_Insertion_Etudiant fonctionne correctement)

```
UPDATE ETUDIANT
SET Genre = 'A'
WHERE EtudiantId = 1612 ;  -- l'étudiant 1612 est un Homme
```
Si après exécution de la requête le genre de l'étudiant 1612 vaut NULL, alors le trigger Avant_Modification_Etudiant fonctionne correctement.

<u>Deuxième issue possible : déclencher une erreur, ce qui empêchera l'insertion ou la modification et affichera une erreur</u>.

Il existe plusieurs manières de déclencher une erreur dans MySQL. En voici une très originale parmi les différentes manières possibles d'obtenir un message d'erreur : on crée une table Erreur avec deux colonnes, id_erreur et Description_erreur. La colonne id_erreur est une clé primaire de la table, et Description_erreur contient un texte décrivant l'erreur. Un index UNIQUE est ajouté sur cette seconde colonne (c'est une colonne avec des valeurs uniques, c'est-à-dire vous ne pouvez pas avoir deux lignes avec la même description d'erreur, sinon cela génère une erreur par le SQBD). On insère ensuite une ligne correspondant à l'erreur que l'on veut utiliser dans le trigger (par exemple la phrase insertion impossible pour le trigger Avant_Insertion_Etudiant ou modification impossible pour le trigger Avant_Modification_Etudiant). Dans le corps du trigger, en cas de valeur erronée, on refait la même insertion dans la table erreur avec la même description d'erreur. Cette action déclenche une erreur de contrainte d'unicité (liée à l'index UNIQUE), laquelle affiche le texte que l'on a essayé d'insérer dans Erreur :

```
-- Création de la table Erreur
CREATE TABLE Erreur (id_erreur  INT NOT NULL AUTO_INCREMENT,
                     Description_erreur  varchar(255) NOT NULL,
                     PRIMARY KEY (id_erreur),
                     UNIQUE KEY (Description_erreur)
                     ) ;
-- Insertion de l'erreur qui nous intéresse
INSERT INTO Erreur (id_erreur, Description_erreur)
VALUES
(NULL, 'Erreur : le genre doit être H, F ou NULL');
-- Création des triggers
-- Trigger 1 déclenché par l'insertion
DELIMITER $$
CREATE TRIGGER Avant_insertion_Etudiant
BEFORE INSERT
ON ETUDIANT
FOR EACH ROW
BEGIN
    IF NEW.Genre IS NOT NULL   -- le genre n'est ni NULL
   AND NEW.Genre != 'H'        -- ni Homme
   AND NEW.Genre != 'F'        -- ni Femme
   THEN
     INSERT INTO Erreur (id_erreur, Description_erreur)
     VALUES (NULL, 'Erreur : le genre doit être H, F ou NULL');
  END IF;
END $$
DELIMITER ;
-- Trigger 2 déclenché par la modification
DELIMITER $$
CREATE TRIGGER Avant_Modification_Etudiant
BEFORE UPDATE
ON ETUDIANT
FOR EACH ROW
BEGIN
    IF NEW.Genre IS NOT NULL   -- le genre n'est ni NULL
   AND NEW.Genre != 'H'        -- ni Homme
   AND NEW.Genre != 'F'        -- ni Femme
   THEN
     INSERT INTO Erreur (id_erreur, Description_erreur)
     VALUES (NULL, 'Erreur : le genre doit être H, F ou NULL');
  END IF;
END $$
DELIMITER ;
```

Tester si le trigger Avant_insertion_Etudiant fonctionne avec la commande suivante :

INSERT INTO ETUDIANT (EtudiantId, SpécialitéId_E, Date_inscription, Frais_inscription, Payement_frais_inscription, Genre) VALUES ('1613', '4', NOW(), '220', ' TRUE', 'A');

L'exécution de la requête d'insertion génère une erreur :

#1062 - Duplicata du champ 'Erreur : le genre doit être H, F ou NULL' pour la clef 'Description_erreur'

Et voilà, nous avons au moins un message d'erreur qui permet de cerner d'où vient le problème (malgré qu'à l' origine ce message est provoqué par le faite d'avoir insérer une même valeur pour l'attribut Description_erreur de la table Erreur).

Tester si le trigger Avant_Modification_Etudiant fonctionne avec la commande suivante :

INSERT INTO ETUDIANT (EtudiantId, SpécialitéId_E, Date_inscription, Frais_inscription, Payement_frais_inscription, Genre) VALUES ('1613', '4', NOW(), '220', ' TRUE', 'H');
Cette commande d'insertion se déroule normalement.

L'exécution de la requête de modification génère une erreur :

```
UPDATE ETUDIANT
SET Genre = 'A'
WHERE EtudiantId = 1612 ;  -- l'étudiant 1613 est mis initialement à H
```

#1062 - Duplicata du champ 'Erreur : le genre doit être H, F ou NULL' pour la clef 'Description_erreur'

Un autre exemple de création de trigger dans le cas le cas d'une insertion ou d'une modification :

Reprenons l'exemple des deux relations précédente mais cette fois ci, l'attribut Genre est enlevé de la table ETUDIANT (exécutez : ALTER TABLE ETUDIANT DROP Genre ; et le tour est jouer !!!!) :

ETUDIANT (**EtudiantId**, SpécialitéId_E, Date_inscription, Frais_inscription, Payement_frais_inscription)
SPECIALITE (**SpécialitéId**, Désignation, Date_Création)

On suppose pour cet exemple que la date d'inscription d'un étudiant est introduite manuellement par un opérateur. Cette date d'inscription de l'étudiant (attribut Date_inscription de la table ETUDIANT) doit forcement être supérieur à la date de création de la spécialité (attribut Date_Création de la table SPECIALITE), c'est normale un étudiant ne peut s'inscrire dans une spécialité avant sa création. Un deuxième point à vérifier est celui de la valeur de l'attribut Payement_frais_inscription de la table

ETUDIANT qui ne peut prendre que deux valeurs 'TRUE' ou 'FALSE' (vrai dans le cas ou l'étudiant a payé ces frais d'inscription et faux dans le cas contraire).

Ces deux contraintes doivent êtres vérifiées (les deux) **avant** (BEFORE) chaque opération d'insertion d'un étudiant (commande INSERT). Et c'est le cas également lors de la modification d'un étudiant existant dans la table ETUDIANT (vérifier la valeur de ces deux attribut **avant** (BEFORE) validation de l'opération de modification par la commande UPDATE).
Pour rappel, il ne peut exister qu'un seul trigger BEFORE UPDATE et qu'un seul BEFORE INSERT pour chaque table (donc on ne pourra pas définir deux triggers BEFORE INSERT, un pour vérifier l'attribut Date_inscription et un autre pour vérifier l'attribut Payement_frais_inscription. De même il n'est pas possible de définir deux triggers BEFORE UPDATE, un pour vérifier l'attribut Date_inscription et un autre pour vérifier l'attribut Payement_frais_inscription). Cela sous entend qu'on devrait créer un seul trigger BEFORE UPDATE pour vérifier les deux contraintes **à la fois avant** (BEFORE) la validation d'une modification (UPDATE) et un seul trigger BEFORE INSERT pour vérifier les deux contraintes **à la fois avant** (BEFORE) la validation d'une insertion (INSERT).

```
-- Création de la table Erreur si elle n'existe pas déjà
CREATE TABLE Erreur (id_erreur  INT NOT NULL AUTO_INCREMENT,
                     Description_erreur  varchar(255) NOT NULL,
                     PRIMARY KEY (id_erreur),
                     UNIQUE KEY Description_erreur (Description_erreur)
                     ) ;
-- Insertion de la première erreur
INSERT INTO Erreur (id_erreur, Description_erreur)
VALUES
(NULL, 'Erreur : la date d'inscription de l'étudiant doit être après date création spécialité.');
-- Insertion de la seconde erreur
INSERT INTO Erreur (id_erreur, Description_erreur)
VALUES
(NULL, 'Erreur : Payement_frais_inscription doit valoir TRUE (1) ou FALSE (0)');

-- Création des triggers
-- Trigger 1 déclenché par l'insertion sur la table ETUDIANT
DROP TRIGGER IF EXISTS Avant_Insertion_Etudiant ;
-- N'oubliez pas que nous avons déjà définie un trigger Avant_Insertion_Etudiant
-- comme on ne peut pas modifier un trigger dans MySQL, il faut le supprimer d'abord
-- puis le définir de nouveau
-- Définir le délimiteur
DELIMITER |
CREATE TRIGGER Avant_Insertion_Etudiant
```

```
BEFORE INSERT
ON ETUDIANT
FOR EACH ROW
BEGIN
  IF NEW.Payement_frais_inscription != 'TRUE'
  AND NEW.Payement_frais_inscription!= 'FALSE'
  THEN  /* générer une erreur à travers une insertion d'une description d'erreur déjà existante*/
    INSERT INTO Erreur (id_erreur, Description_erreur)
    VALUES
    (NULL, 'Erreur : Payement_frais_inscription doit valoir TRUE (1) ou FALSE (0)');
  END IF;
  IF NEW.Date_inscription < (SELECT Date_Création
                             FROM SPECIALITE
                             WHERE NEW.SpécialitéId_E = SpécialitéId)
  /* Date_inscription < Date_Création veut dire que la date de création précède la date d'inscription*/
  THEN  /* générer une erreur à travers une insertion d'une description d'erreur déjà existante*/
  INSERT INTO Erreur (id_erreur, Description_erreur)
  VALUES
  (NULL, 'Erreur : la date d'inscription de l'étudiant doit être après date création spécialité.');
  END IF;
END |
-- Ne oublier de remettre votre délimiteur à sa valeur initiale « ; »
DELIMITER ;

-- Trigger 2 déclenché par la modification sur la table ETUDIANT
DROP TRIGGER IF EXISTS Avant_Modification_Etudiant ;
-- N'oubliez pas que nous avons déjà définie un trigger Avant_Modification_Etudiant
-- comme on ne peut pas modifier un trigger dans MySQL, il faut le supprimer d'abord
-- puis le définir de nouveau
-- Définir le délimiteur
DELIMITER |
CREATE TRIGGER Avant_Modification_Etudiant
BEFORE UPDATE
ON ETUDIANT
FOR EACH ROW
BEGIN
  IF NEW.Payement_frais_inscription != 'TRUE'
  AND NEW.Payement_frais_inscription!= 'FALSE'
  THEN  /* générer une erreur à travers une insertion d'une description d'erreur déjà existante*/
    INSERT INTO Erreur (id_erreur, Description_erreur)
    VALUES
    (NULL, 'Erreur : Payement_frais_inscription doit valoir TRUE (1) ou FALSE (0)');
```

```
  END IF;
  IF NEW.Date_inscription < (SELECT Date_Création
                             FROM SPECIALITE
                             WHERE NEW.SpécialitéId_E = SpécialitéId)
  /* Date_inscription < Date_Création veut dire que la date de création précède la date d'inscription*/
  THEN  /* générer une erreur à travers une insertion d'une description d'erreur déjà existante*/
  INSERT INTO Erreur (id_erreur, Description_erreur)
  VALUES
  (NULL, 'Erreur : la date d'inscription de l'étudiant doit être après date création spécialité.');
  END IF;
END |
-- Ne oublier de remettre votre délimiteur à sa valeur initiale « ; »
DELIMITER ;
-- Le trigger ne pourra déclencher qu'une erreur à la fois.
-- car le déclanchement d'une erreur provoque l'arrêt du script.
```

Test du bon fonctionnement des deux triggers :

```
INSERT INTO SPECIALITE (SpécialitéId, Désignation, Date_Création)
VALUES (1, 'ISIL', '2009-11-21');

INSERT INTO ETUDIANT (EtudiantId, SpécialitéId_E, Date_inscription, Frais_inscription, Payement_frais_inscription)
VALUES (125, '1', '2009-11-22', '200', 'TRUE');
```

Cette première insertion dans la table ETUDIANT se passe bien et ne génère aucune erreur car l'étudiant c'est inscrit un jour après la date de création de la spécialité.

```
INSERT INTO ETUDIANT (EtudiantId, SpécialitéId_E, Date_inscription, Frais_inscription, Payement_frais_inscription)
VALUES (126, '1', '2009-11-20', '200', 'TRUE');
```

Cette deuxième insertion dans la table ETUDIANT génère une erreur car l'étudiant c'est inscrit un jour avant la date de création de la spécialité (Dans la table spécialité, la spécialité « SpécialitéId=1 » à été créée le '2009-11-21'). Ainsi la requête INSERT génère l'erreur :

#1062 - Duplicata du champ 'Erreur : la date d'inscription de l'étudiant doit être après date création spécialité.' pour la clef 'Description_erreur'

```
INSERT INTO ETUDIANT (EtudiantId, SpécialitéId_E, Date_inscription, Frais_inscription, Payement_frais_inscription)
VALUES (127, '1', '2011-01-13', '200', 'TRUE');
```

Cette insertion dans la table ETUDIANT se passe bien et ne génère aucune erreur car Payement_frais_inscription vaut TRUE.

INSERT INTO ETUDIANT (EtudiantId, SpécialitéId_E, Date_inscription, Frais_inscription, Payement_frais_inscription)
VALUES (128, '1', '2011-01-13', '200', 'T');
Cette insertion dans la table ETUDIANT génère une erreur car Payement_frais_inscription vaut T. La requête INSERT génère l'erreur :
#1062 - Duplicata du champ 'Erreur : Payement_frais_inscription doit valoir TRUE (1) ou FALSE (0)' pour la clef 'Description_erreur'

UPDATE ETUDIANT
SET Date_inscription = '2012-01-01'
WHERE EtudiantId = 127 ; -- l'étudiant 127 doit exister dans la table ETUDIANT
Cette requête de modification (UPDATE) s'exécute normalement et modifie la date d'inscription de l'étudiant 127 de '2011-01-13' à '2012-01-01'. La requête ne génère pas d'erreur car même après modification la date d'inscription de l'étudiant est après la date création spécialité.

UPDATE ETUDIANT
SET Date_inscription = '2008-03-13'
WHERE EtudiantId = 127 ; -- l'étudiant 127 doit exister dans la table ETUDIANT
Cette requête de modification UPDATE génère une erreur car la date d'inscription de l'étudiant à la spécialité précède la date de création de la spécialité. Cette requête de modification génère l'erreur :
#1062 - Duplicata du champ 'Erreur : la date d'inscription de l'étudiant doit être après date création spécialité.' pour la clef 'Description_erreur'

UPDATE ETUDIANT
SET Payement_frais_inscription = 'FALSE'
WHERE EtudiantId = 127 ; -- l'étudiant 127 doit exister dans la table ETUDIANT
Cette requête de modification UPDATE s'exécute correctement car nous avons modifié l'attribut Payement_frais_inscription qui été à TRUE et nous l'avons mis à FALSE.
UPDATE ETUDIANT
SET Payement_frais_inscription = 'F'
WHERE EtudiantId = 127 ; -- l'étudiant 127 doit exister dans la table ETUDIANT
Cette requête de modification UPDATE génère une erreur :
Duplicate entry 'Erreur : Payement_frais_inscription doit valoir TRUE (1) ou FALSE (0).' pour la clef 'Description_erreur'.
Car nous avons voulu modifier l'attribut Payement_frais_inscription qui été à FALSE par une valeur aberrante (F).
Les deux vérifications fonctionnent correctement.

Mise à jour d'informations dépendant d'autres données par usage de Triggers

Pour comprendre se volet sur lequel les triggers peuvent agir, on prend comme exemple l'ajoute la colonne « Nbr_place » à la table Spécialité par la requête suivante (on met la valeur 0 comme valeur par défaut de cet attribut):

```
ALTER TABLE SPECIALITE ADD Nbr_place INT NOT NULL DEFAULT '0' AFTER Date_Création;
```

Nous obtenons les deux relations :

ETUDIANT (**EtudiantId**, SpécialitéId_E, Date_inscription, Frais_inscription, Payement_frais_inscription)

SPECIALITE (**SpécialitéId**, Désignation, Date_Création, Nbr_place)

Cette nouvelle colonne (Nbr_place) représente le nombre de place pédagogique réellement occupées dans la spécialité et qui aura pour valeur initiale (à la création de la spécialité), la valeur par défaut 0. La colonne « Nbr_place » pour une spécialité donnée doit être mise à jour à chaque fois qu'un nouveau étudiant s'inscrit dans la spécialité. La tache de mise à jour de nombre de place pédagogique réellement occupées dans une spécialité peut être réalisée grâce à trois triggers sur la table ETUDIANT.

- À l'insertion d'un nouvel étudiant dans la table ETUDIANT, il faut augmenter le nombre de places pédagogiques réellement occupées dans la spécialité à laquelle l'étudiant compte s'inscrire.
- En cas de suppression d'un étudiant, il faut faire le contraire (diminuer le nombre de nombre de place pédagogique de spécialité dans laquelle l'étudiant été inscrit).
- En cas de modification d'un tuple étudiant et plus particulièrement dans le cas ou se dernier change de spécialité, il faut diminuer le nombre de place pédagogique de l'ancienne spécialité dans laquelle l'étudiant été inscrit et augmenter celle de la nouvelle spécialité à laquelle l'étudiant sera inscrit.

Remarque : dans ce cas de figure il n'ya pas de message d'erreur car on ne cherche pas à bloquer une opération d'insertion (INSERT), de suppression (DELETE) ou de modification (UPDATE) si une condition n'est pas vérifiée. Il s'agit tous simple de faire la mise à jour de la valeur d'un attribut (nombre de place pédagogique réellement occupées dans la spécialité) **après** (AFTER) une opération sur une table (les événements déclencheur sont INSERT/ DELETE/ UPDATE).

```
-- Ajouter de la colonne Nbr_place à la table Spécialité
ALTER TABLE Spécialité ADD COLUMN Nbr_place INT NOT NULL DEFAULT '0';

-- Création des trois triggers
-- Premier trigger :
DROP TRIGGER IF EXISTS Aprés_Insertion_ETUDIANT;
```

```
-- Création Trigger Aprés_Insertion_ETUDIANT
DELIMITER |
CREATE TRIGGER Aprés_Insertion_ETUDIANT
AFTER INSERT
ON ETUDIANT
FOR EACH ROW
BEGIN
    UPDATE SPECIALITE
    SET Nbr_place  = Nbr_place + 1
    WHERE SpécialitéId= NEW.SpécialitéId_E;
END |
DELIMITER ;

-- Second trigger :
DROP TRIGGER IF EXISTS Aprés_Suppression_ETUDIANT;
-- Création Trigger Aprés_Suppression_ETUDIANT
DELIMITER |
CREATE TRIGGER Aprés_Suppression_ETUDIANT
AFTER DELETE
ON ETUDIANT
FOR EACH ROW
BEGIN
    UPDATE SPECIALITE
    SET Nbr_place  = Nbr_place - 1
    WHERE SpécialitéId=OLD.SpécialitéId_E;
END |
DELIMITER ;

-- Troisième trigger :
DROP TRIGGER IF EXISTS Aprés_Modification_ETUDIANT;
-- Création Trigger Aprés_Modification_ETUDIANT
DELIMITER |
CREATE TRIGGER Aprés_Modification_ETUDIANT
AFTER UPDATE
ON ETUDIANT
FOR EACH ROW
BEGIN
IF OLD.SpécialitéId_E != NEW.SpécialitéId_E
THEN
/*Diminuer le nombre de place de l'ancienne spécialité de l'étudiant*/
    UPDATE SPECIALITE
    SET Nbr_place  = Nbr_place - 1
    WHERE SpécialitéId= OLD.SpécialitéId_E;
```

```
/*Augmenter le nombre de place de la nouvelle spécialité de l'étudiant*/
    UPDATE SPECIALITE
    SET Nbr_place  = Nbr_place + 1
    WHERE SpécialitéId= NEW.SpécialitéId_E;
END IF;
END |
DELIMITER ;
```

Tester les trois triggers :

```
INSERT INTO SPECIALITE (SpécialitéId, Désignation, Date_Création, Nbr_place )
VALUES (6, 'GTR', '2007-01-01',0);

INSERT INTO SPECIALITE (SpécialitéId, Désignation, Date_Création, Nbr_place )
VALUES (7, 'GL', '2010-12-31',0);

INSERT INTO ETUDIANT (EtudiantId, SpécialitéId_E, Date_inscription, Frais_inscription,
Payement_frais_inscription)
VALUES
(520, '6', '2016-08-27', '200', 'TRUE'),
(521, '6', '2016-08-27', '200', 'TRUE'),
(522, '6', '2016-08-27', '200', 'TRUE');
```

Après l'exécution de cette requête, le nombre de place pédagogique réellement occupées dans la spécialité 6 passe de 0 à 3 (vérifiez les tuples de la table et vous allez voir que c'est vrai).

```
DELETE FROM ETUDIANT  -- l'étudiant 520  doit exister dans la table ETUDIANT
WHERE EtudiantId = 520; -- voir le Nbr_place  de la spécialité de l'étudiant 520
```

Après suppression de l'étudiant 520, le nombre de place pédagogique réellement occupées dans la spécialité 6 diminue d'une place et devient 2 (vérifiez les tuples de la table et vous allez voir que c'est vrai).

```
UPDATE ETUDIANT
SET SpécialitéId_E = '7'-- Spécialité_id_E de l'étudiant 521 est devenu 7 et non pas 6
WHERE EtudiantId = 521;  -- l'étudiant 521  doit exister dans la table ETUDIANT
```

Après modification de la spécialité de l'étudiant 521, le nombre de place pédagogique réellement occupées dans la spécialité 6 devient 1 et le nombre de place pédagogique réellement occupées dans la spécialité 7 devient 1 (vérifiez les tuples de la table et vous allez voir que c'est vrai).

Imposer des restrictions sur les valeurs de tuples par usage de Triggers :

Les triggers sont souvent utilisés pour imposer des restrictions sur les valeurs acceptés par certaines colonnes d'une table. Pour comprendre ce point on se propose les relations :

Cinéma (**numC**, nomCinéma, numéro, rue, ville)
Salle (**NumSalle, #numC**, capacité, climatisée)
Film (**idFilm**, titre, année, genre, résumé)
Séance (**#idFilm, #numC, #NumSalle**, tarif)

Nous avons les contraintes suivantes sur le schéma relationnel :

Un **Cinéma** doit faire partie de la liste des **villes** {Alger, Oran, Constantine, Adrar} :

Lors de l'insertion ou la modification d'un tuple **Cinéma** il faut vérifier que la valeur de la ville est prise de l'ensemble {Alger, Oran, Constantine, Adrar}. Dans le cas ou la valeur de la ville n'est pas prise de cet ensemble de valeurs, elle est mise directement à NULL. Ainsi nous avons deux triggers pour cette contrainte :

1/- Trigger **avant** insertion (INSERT) sur la table «**Cinéma** » qui contrôle la valeur introduite pour la colonne Ville (dans le cas ou la valeur n'est pas correct, le trigger met la valeur NULL pour cette colonne).
2/- Trigger **avant** modification (UPDATE) sur la table «**Cinéma** » qui contrôle si la valeur de la Ville change. Et dans le cas ou la valeur de la ville est modifiée, le trigger contrôle la nouvelle valeur introduite dans cette colonne (quand la valeur introduite n'est pas correct, le trigger met la valeur NULL pour cette colonne).

```
-- Code trigger Avant_Insertion_Cinéma :
DROP TRIGGER IF EXISTS Avant_Insertion_Cinéma ;
DELIMITER |
CREATE TRIGGER Avant_Insertion_Cinéma
BEFORE  INSERT -- avant toute insertion dans la table Cinéma.
ON Cinéma
FOR EACH ROW
BEGIN
  IF NEW.ville NOT IN ('Alger', 'Oran', 'Constantine', 'Adrar')
  THEN
     SET NEW.ville = NULL;
 END IF ;
END | -- le TRIGGER se termine grâce au nouveau délimitateur.
DELIMITER ; -- on restaure l'ancien délimitateur.
```

```
-- Code trigger Avant_Modification_Cinéma :
DROP TRIGGER IF EXISTS Avant_Modification_Cinéma ;
DELIMITER |
CREATE TRIGGER Avant_Modification_Cinéma
BEFORE  UPDATE -- avant toute modification dans la table Cinéma.
ON Cinéma
FOR EACH ROW
BEGIN
  IF NEW.ville NOT IN ('Alger', 'Oran', 'Constantine', 'Adrar')
  THEN
     SET NEW.ville = NULL;
 END IF ;
END | -- le TRIGGER se termine grâce au nouveau délimitateur.
DELIMITER ; -- on restaure l'ancien délimitateur.
```

Remarque : à la déclaration de la table Cinéma ne pas oublier de spécifier que le champ ville de la table Cinéma peut être NULL.

Test :

```
INSERT INTO Cinéma (numC, nomCinéma, numéro, rue, ville)
VALUES (1,'El Djazair',12,'Ali Khoudja', 'Alger');
```

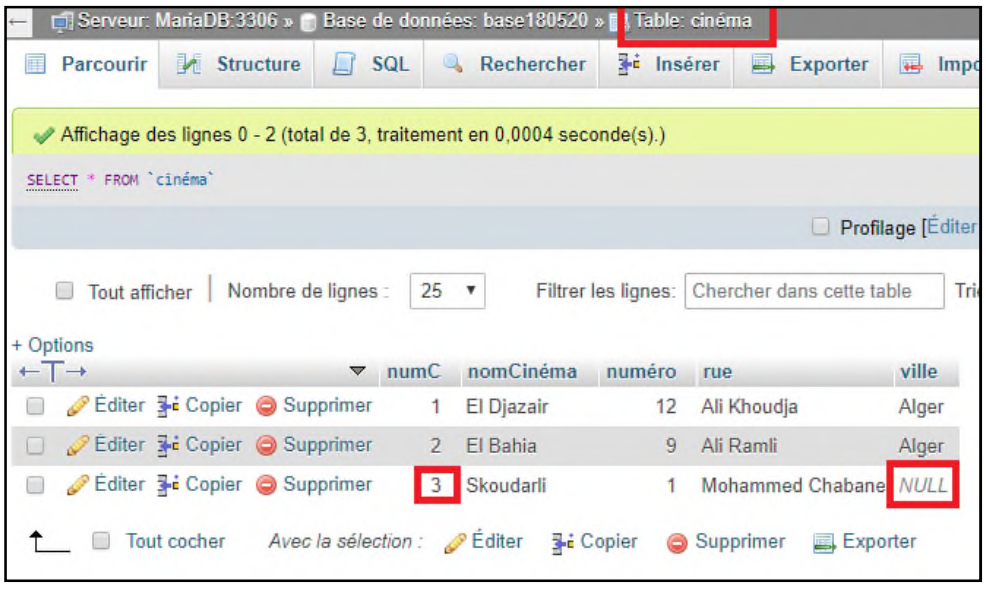
Pour cette insertion le champ ville prend la valeur Alger.

```
INSERT INTO Cinéma (numC, nomCinéma, numéro, rue, ville)
VALUES (2,'El Bahia', 09,'Ali Ramli', 'Alger');
```

Pour cette insertion le champ ville prend la valeur Alger.

```
INSERT INTO Cinéma (numC, nomCinéma, numéro, rue, ville)
VALUES (3,'Skoudarli', 01,'Mohammed Chabane', 'A');
```

Pour cette insertion le champ ville prend la valeur NULL.



Pour la modification dans « Cinéma » :

```
UPDATE Cinéma
SET  ville = 'Oran'
WHERE numC = 1;
```

Cette modification change la ville du cinéma 1 à Oran (à la place d'Alger).

```
UPDATE Cinéma
SET  ville = 'Bejaia'
WHERE numC = 2;
```

Cette requête de modification fait passer la ville du cinéma 2 à NULL car la ville proposée dans l'UPDATE n'existe pas dans la liste des villes admises.

<u>L'année de parution d'un film est toujours connue :</u>

Pour satisfaire cette contrainte il suffit de spécifier lors de la création de la table Film que le champ « année » n'est pas NULL (NOT NULL).....on la fait dans le TP1+TP2 (avec interface et avec le langage SQL).

<u>Le tarif d'une séance est supérieur à 700DA (il est au minimum =700 DA) :</u>

Ainsi nous avons deux triggers pour cette contrainte :

1/- Trigger **avant** insertion (INSERT) sur la table «**Séance**» qui contrôle la valeur introduite pour la colonne tarif (dans le cas ou la valeur est inferieur à 700DA, le trigger met la valeur 700 pour cette colonne).

2/- Trigger **avant** modification (UPDATE) sur la table «**Séance**» qui contrôle si la valeur du champ tarif change. Et dans le cas ou la valeur du champ tarif est modifié, le trigger contrôle la nouvelle valeur introduite dans cette colonne (quand la valeur introduite n'est pas correct, le trigger met la valeur 700 pour cette colonne).

```
-- Code trigger Avant_Insertion_Séance :
DROP TRIGGER IF EXISTS Avant_Insertion_Séance ;
DELIMITER |
CREATE TRIGGER Avant_Insertion_Séance
```

```
BEFORE  INSERT -- avant toute insertion dans la table Séance.
ON Séance
FOR EACH ROW
BEGIN
  IF NEW.tarif < 700
  THEN
     SET NEW.tarif = 700;
 END IF ;
END | -- le TRIGGER se termine grâce au nouveau délimitateur.
DELIMITER ; -- on restaure l'ancien délimitateur.

-- Code trigger Avant_Modification_Séance :
DROP TRIGGER IF EXISTS Avant_Modification_Séance ;
DELIMITER |
CREATE TRIGGER Avant_Modification_Séance
BEFORE  UPDATE -- avant toute modification dans la table Séance.
ON Séance
FOR EACH ROW
BEGIN
  IF NEW.tarif < 700
  THEN
     SET NEW.tarif = 700;
 END IF ;
END | -- le TRIGGER se termine grâce au nouveau délimitateur.
DELIMITER ; -- on restaure l'ancien délimitateur.
```

Pour les testes je vous laisse le soin de le faire (faite des insertions dans la table Séance avec des séances de tarif inferieurs à 700DA, avec des séances de tarifs=700DA et avec des séances de tarif supérieurs à 700 DA. Vous allez voir que toutes les séances avec des tarifs totalement inferieurs à 700DA seront mis à un tarif de 700DA par le trigger. Pour les modifications, essayez de prendre une séance avec un tarif supérieur à 700DA et fait un UPDATE sur ce tuple en mettant un tarif de 900DA, vous allez remarquer que la modification sera prise en considération. Maintenant si vous faite un UPDATE d'une séance et que vous mettez le tarif à 200 DA, le trigger va directement changer cette valeur et va mettre 700 à la place…… vérifiez).

<u>Restreindre les valeurs possibles des attributs capacité <300 et climatisée (soit OUI ou NON) dans la table Salle :</u>

Pour cette contrainte il faut d'une part :

- Il faut définir un Trigger Avant_Insertion_Salle (BEFORE INSERT ON Salle) permettant de vérifier la valeur du champ capacité d'une salle (on suppose que toute les salles de cinéma ont au minimum une capacité de 300 personnes, sinon ce n'est plus considérer comme une salle de cinéma). La valeur introduite de le

champ capacité est vérifiée et dans le cas ou cette valeur est inferieur à 300, le Trigger se charge de la mettre à sa plus petite valeur possible qui est égale à 300.

- Il faut définir un Trigger Avant_Insertion_Salle (BEFORE INSERT ON Salle) permettant de vérifier la valeur du champ climatisée d'une salle. La valeur introduite de le champ climatisée est vérifiée et dans le cas ou cette valeur est erroné (différente de OUI/NON), le Trigger se charge de la mettre à NON (on suppose que si la valeur de champ climatisée n'est pas bien spécifier pour une salle, cela veut automatiquement dire qu'elle ne l'est pas).
- J'attire votre attention sur un point très important. Il n'est **pas possible** d'avoir deux triggers BEFORE INSERT pour **la même table**. Ainsi on ne peut pas avoir un trigger Avant_Insertion_Salle pour vérifier la valeur du champ capacité et un autre un trigger Avant_Insertion_Salle pour vérifier la valeur du champ climatisée. Solution on va définir un seul trigger Avant_Insertion_Salle qui permet de faire les deux vérifications au même temps.

D'autre part :

- Il faut définir un Trigger Avant_Modification_Salle (BEFORE UPDATE ON Salle) permettant de vérifier la valeur du champ capacité d'une salle lors de l'application d'un UPDATE sur un tuple de la table Salle. Si la valeur du champ capacité change par l'effet de la commande UPDATE, cette nouvelle valeur doit être vérifiée avant de remplacer l'ancienne valeur. Dans le cas ou cette valeur est erroné, le trigger se charge de mettre la valeur 300 (valeur par défaut) à sa place (à la place de la valeur spécifiée dans le UPDATE).
- Il faut définir également un Trigger Avant_Modification_Salle (BEFORE UPDATE ON Salle) permettant de vérifier la valeur du champ climatisée lors d'un UPDATE sur la table salle. Si la valeur introduite pour le champ climatisée est différente de OUI/NON), le Trigger se charge de la mettre à NON à la place de la valeur spécifiée dans le UPDATE.
- Même remarque pour ce cas de figure, Il n'est **pas possible** d'avoir deux triggers BEFORE UPDATE pour **la même table**. Par consequent on ne peut pas avoir un trigger Avant_Modification_Salle pour vérifier la valeur du champ capacité et un autre un trigger Avant_Modification_Salle pour vérifier la valeur du champ climatisée. Solution on va définir un seul trigger Avant_Modification_Salle qui permet de faire les deux vérifications au même temps.

```
-- Création des triggers
-- Trigger 1 déclenché par l'insertion sur la table Salle
DROP TRIGGER IF EXISTS Avant_Insertion_Salle ;
-- comme on ne peut pas modifier un trigger dans MySQL, il faut le supprimer d'abord
-- puis le définir de nouveau
-- Définir le délimiteur
DELIMITER |
```

```
CREATE TRIGGER Avant_Insertion_Salle
BEFORE INSERT
ON Salle
FOR EACH ROW
BEGIN
  IF NEW.climatisée != 'OUI'
  AND NEW.climatisée != 'NON'  /* on peut écrire NEW.climatisée  NOT IN  ('OUI',
'NON')*/
  THEN
    SET NEW.climatisée = 'NON' ;
  END IF;
  IF NEW.capacité < 300
  THEN
   SET NEW.capacité = 300 ;
  END IF;
END |
-- Ne oublier de remettre votre délimiteur à sa valeur initiale « ; »
DELIMITER ;

-- Trigger 2 déclenché par la modification sur la table Salle
DROP TRIGGER IF EXISTS Avant_Modification_Salle ;
-- comme on ne peut pas modifier un trigger dans MySQL, il faut le supprimer d'abord
-- puis le définir de nouveau
-- Définir le délimiteur
DELIMITER |
CREATE TRIGGER Avant_Modification_Salle
BEFORE UPDATE
ON Salle
FOR EACH ROW
BEGIN
  IF NEW.climatisée != 'OUI'
  AND NEW.climatisée != 'NON'  /* on peut écrire NEW.climatisée  NOT IN  ('OUI',
'NON')*/
  THEN
    SET NEW.climatisée = 'NON' ;
  END IF;
  IF NEW.capacité < 300
  THEN
   SET NEW.capacité = 300 ;
  END IF;
END |
-- Ne oublier de remettre votre délimiteur à sa valeur initiale « ; »
DELIMITER ;
```

-- Le trigger ne pourra déclencher qu'une erreur à la fois.
-- car le déclanchement d'une erreur provoque l'arrêt du script.

Pour les testes je vous laisse le soin de faire les testes nécessaires pour la validation du fonctionnement de ces deux Triggers…..

Commandes interdites dans un trigger :

Il existe certaines restrictions sur les trigger. En Voici un exemple de ces restrictions :
Une suppression ou modification de données déclenchée par une clé étrangère ne provoquera pas l'exécution du trigger correspondant.
Par exemple, la colonne ETUDIANT.SpécialitéId_E est une clé étrangère qui référence la colonne SpécialitéId de la table spécialité. Cette clé étrangère a été définie avec l'option ON DELETE SET NULL. Donc en cas de suppression d'une spécialité, tous les étudiants de cette spécialité seront modifiés et leur spécialité sera changée et mise à la valeur NULL. Il s'agit donc d'une modification de données dans la table ETUDIANT. Mais comme cette modification a été déclenchée par une contrainte de clé étrangère, les éventuels triggers BEFORE UPDATE et AFTER UPDATE définiessur la table ETUDIANT 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 ces options (options de modification déclenchées par une contrainte de clé étrangère : ON DELETE CASCADE, ON UPDATE CASCADE, ON DELETE SET NULL, ON UPDATE SET NULL, etc.) et déplacer ce comportement dans des triggers.
Une autre solution est de ne pas utiliser les triggers sur les tables concernées (tables ayant des clés étrangères).

En résumé du trigger (déclencheur)

- Un trigger est un objet stocké dans la base de données.
- Un trigger est lié à une table (un déclencheur associée à une table), donc en cas de suppression d'une table, les triggers liés à celle-ci sont supprimés également.
- Pour créer un trigger (un déclencheur) ou supprimer un trigger (un déclencheur) l'instruction CREATE TRIGGER et DROP TRIGGER sont utilisées.
- 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 (BEFORE/AFTER).
- Une table ne peut posséder qu'un seul trigger par combinaison événement/moment (BEFORE UPDATE, AFTER DELETE…).
- Les triggers sous MySQL sont soumis à certaines restrictions.

Pour la prochaine séance de TP (celle de ce jeudi) d'autres exemples :

Exemple1 : Soit les relations suivantes :

Enseignant (**NumEns**, NomEns, TélEns)
Stage (**NumStage**, LibelléStg, DuréeStg)
Formateur (**NumEnsF, NumStageF**)
Stagiaire (**NumEnsStg, NumStageStg**)

Chaque enseignant a le droit d'être un formateur dans un stage. La relation Formateur permet d'enregistrer les enseignants qui sont formateurs dans ces stages.
Chaque enseignant peut être également stagiaire dans un stage (chaque enseignant peut s'inscrire dans un stage pour améliorer ces connaissances). La relation stagiaire récence les enseignants inscrit pour un stage (les enseignants qui sont stagiaires). Contrainte : un enseignant ne peut pas être formateur et stagiaire à la fois.

Exemple2 : Soit les relations suivantes :

Personne (**NumPers**, NomPers, TélPers)
Stage (**NumStage**, LibelléStg, DuréeStg)
Demandeur (**NumPersD, NumStageD**)
Inscrit (**NumPersI, NumStageI**)

Toute personne peut être demandeur de stage. Cependant, toutes les demandes de stage ne peuvent pas être satisfaites. Il ya juste une partie de demandeur de stage qui seront inscrit dans des stages (seulement une partie des demandes de stage sont satisfaites). A partir des relations ci-dessus, nous avons la contrainte suivante :

Une personne ne peut pas être inscrite dans un stage que si elle en fait une demande au préalable. La relation «Inscrit » regroupe tous les demandeurs de stage qui sont acceptés. De ce fait, la relation «Inscrit » est un sous ensemble de la relation « Demandeur ». Cette contrainte implique l'interdiction de la présence d'un couple (Numéro Personne, Numéro Stage) dans la table «Inscrit », s'il n'est pas présent de la table « Demandeur ».

Exemple3 : Soit les relations suivantes

Facture (**NumFac**, DateFac, TotalFac)
Article (**NumArt**, NomArt, PrixUnitaireArt)
LigneFacture (**NumFacL**, **NumArtL**, Quantité)

Une facture comporte plusieurs lignes (un article par ligne). Pour chaque ligne, un article peut être acheté en Quantité >=1 (c'est la quantité achetée pour chaque article).
L'attribut TotalFac représente le total de toutes les lignes d'une facture (le total des articles présent dans la facture). Ce total doit être mis à jour automatiquement.

Exemple4 : Soit les relations suivantes

Produit (**NomProduit**, QuantitéProdEnStock)
Commande (**NumCom**, DateCom, NomProduitCom, QuantitéCom)

Un fournisseur de produit dispose de ces deux tabes pour la gestion de son stock et la gestion de ces commandes.
Lors d'une insertion d'une nouvelle commande (insertion d'une nouvelle ligne dans la table commande), la quantité d'un produit commandé dans une commande doit être diminuée automatiquement de la quantité en stock de ce produit.