

TP3 : DOCUMENT B (APRES SEANCE DE TP)

Certaines parties (au début seulement) ressemblent à la partie A (juste pour révision), le reste est différent.....alors je vous demande de voir tous le document SVP.

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 droit)



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



- **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'à :

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).
- Dans une procédure stockée (PSM : Persistent Stored Module) appelé à partir d'un programme extérieur.

Donc 3 manières d'exécuter un programme PL/SQL.

Trigger (ou déclencheur) :

Définition 1 : Un Trigger (ou déclencheur en français) est un scripte stocké dans le SGBDR permettant d'exécuter une action **avant** (**BEFORE**) ou **après** (**AFTER**) avoir reçu un événement (une commande). Trois événements déclencheurs peuvent exister : la commande **INSERT**, la commande **UPDATE** et la commande **DELETE**. Par exemple, il est possible d'indiquer à MySQL d'exécuter certaines actions (tâches) avant de réaliser un **INSERT** ou un **UPDATE** sur une table. Tout Trigger est lié à une seule table et ne peut pas être déclenché sur les événements **INSERT/UPDATE/DELETE** quand ces derniers se produisent sur d'autres tables.

Définition 2 : Un Trigger (ou déclencheur en français) est un programme stocké dans le SGBDR qui s'exécute automatiquement lorsqu'un événement spécifique se produit sur une table précise. Ces **événements** peuvent être une instruction d'insertion sur une table (**INSERT**), une instruction de modification sur une table (**UPDATE**) ou une instruction de suppression (**DELETE**) exécutée sur une table. Le Trigger peut être programmé pour s'exécuter (se déclencher) juste **Avant** (**BEFORE**) l'événement déclencheur ou juste **Après** (**AFTER**) l'événement déclencheur. Tout Trigger est attaché à une seule table précise et ne peut pas être déclenché sur les événements **INSERT/UPDATE/DELETE** quand ces derniers se produisent sur d'autres tables.

Il faut aussi savoir qu'un Trigger exécute un traitement pour chaque ligne (chaque tuple, chaque instances) insérée (par la commande **INSERT**), supprimée (par la commande **DELETE**) ou modifiée (par la commande **UPDATE**). Ainsi si l'on traite dix lignes à la fois dans une même commande, le traitement sera effectué dix fois (le trigger se déclenchera 10 fois). De plus, le traitement ne peut être exécuté que sur la table avec laquelle le Trigger est lié.

Gestion des triggers

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 ;
```

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 trouver dans d'autres SGBDR tel que SQL Server mais pas dans MySQL). Il faut donc supprimer le Trigger puis le recréer de nouveau pour mettre un contenu différent. Notez également que suppression d'une table engendre automatiquement la suppression de tous les triggers qui y sont attachés.

Chronologie et Type de déclenchement d'un trigger :

BEFORE / AFTER **INSERT/UPDATE/DELETE** **ON** **NomTable**

BEFORE : Un trigger **BEFORE** est un trigger qui se déclenche (s'exécute) **avant** une action « **INSERT** ou **UPDATE** ou **DELETE** ».

AFTER : Un trigger **AFTER** est un trigger qui se déclenche (s'exécute) **après** une action « **INSERT** ou **UPDATE** ou **DELETE** ».

Les deux chronologies (**BEFORE / AFTER**) croisés à trois types de déclenchement (**INSERT / UPDATE / DELETE**) permettent d'obtenir 6 types de triggers :

- **TRIGGER BEFORE INSERT**

- TRIGGER AFTER INSERT
- TRIGGER BEFORE UPDATE
- TRIGGER AFTER UPDATE
- TRIGGER BEFORE DELETE
- TRIGGER AFTER DELETE

L'usage de Trigger permet d'exécuter des traitements d'une manière automatique.

Notez que **pour une table donnée**, il n'est possible d'avoir qu'un seul Trigger BEFORE INSERT, un seul Trigger AFTER INSERT, un seul Trigger BEFORE UPDATE, etc. Par contre un même trigger peut englober plusieurs actions à faire.

En d'autres termes, on ne peut pas avoir plusieurs déclencheurs pour le même événement sur la même table. Si on veut faire plusieurs traitements différents à un même moment, il faut les regrouper dans le même déclencheur tel que montré dans cette exemple ou nous voulons avoir deux Triggers BEFORE UPDATE sur la même table « table1 » :

- CREATE TRIGGER nom_trigger_1 BEFORE UPDATE ON table1 FOR EACH ROW SET action1;
- CREATE TRIGGER nom_trigger_2 BEFORE UPDATE ON table1 FOR EACH ROW SET action2;

On ne peut pas définir deux Triggers BEFORE UPDATE ON sur table1.

Solution :

- CREATE TRIGGER nom_trigger_12 BEFORE UPDATE ON table1 FOR EACH ROW BEGIN action1; action2; END ;

Rappelons la syntaxe des trois commandes 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 stipulé** 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.

Nombre de déclenchement :

L'instruction **FOR EACH ROW** permet au trigger de traiter toutes les lignes concernées par l'action **INSERT** / **UPDATE** / **DELETE** (un insert qui met en jeu l'insertion de 5 tuples à la fois, implique par l'effet de l'instruction **FOR EACH ROW**, l'exécution du trigger pour chaque ligne concernée par le INSERT, et donc 5 exécution du trigger).

Récupération de valeurs (NEW/OLD) :

- **INSERT** : **NEW**.attribut (dans une insertion il n'y a pas d'anciennes valeurs)
- **UPDATE** : **NEW**.attribut / **OLD**.attribut
- **DELETE** : **OLD**.attribut (dans une suppression il n'y a pas de nouvelles valeurs)

Exemple d'utilisation des triggers dans les opérations « INSERT / UPDATE / DELETE » :

Dans toutes conceptions certaines contraintes peuvent être soulevées (permettre par exemple une insertion sur une table **que si une condition spécifique est respectée** et **empêcher cette insertion si cette condition** n'est pas respectée dans la commande d'insertion). Ces contraintes peuvent se traduire par des Trigger (programmés des contraintes à l'aide de Trigger à pour but de bloquer ou autoriser l'exécution d'une commande en fonction du respect ou du non respect d'une condition).

Exemple 1 d'utilisation de Trigger lors d'insertions/modifications de valeurs :

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 tous 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 inscrits pour un stage (les enseignants qui sont stagiaires).

Nous avons la contrainte suivante : un enseignant qui est stagiaire dans un stage ne peut pas être formateur dans ce même stage (**un enseignant ne peut pas être stagiaire et formateur à la fois dans le même stage**).

Cette contrainte se traduit en langage SQL par l'interdiction de la présence d'un même couple (**Numéro enseignant /numéro stage**) à la fois dans la table « Formateur » et dans la table « Stagiaire ».

On va donc programmer le SGBDR pour que lors de l'insertion d'un nouveau formateur dans la table « Formateur » pour un stage, il faut contrôler que ce formateur n'est pas stagiaire dans cette formation. Cette vérification doit se faire **avant l'insertion dans la table « Formateur »** (autrement, si le contrôle se produit après l'insertion sa ne servira à rien et sa sera trop tard).

Ainsi nous avons un premier Trigger associé a cette contrainte :

```
DROP TRIGGER IF EXISTS Avant_insertion_Formateur ; -- voir explication 1
DELIMITER // -- le délimiteur du bloc d'instructions devient //. Voir explication 2
CREATE TRIGGER Avant_insertion_Formateur -- voir explication 3
BEFORE INSERT -- avant toute insertion dans la table Formateur. Voir explication 4
ON Formateur
FOR EACH ROW -- voir explication 5
BEGIN
    DECLARE nb INTEGER;
    SELECT COUNT(*) INTO nb /*le résultat de la requête est récupéré dans la variable nb*/
    FROM Stagiaire
    WHERE NumEnsStg = NEW.NumEnsF
    AND NumStageStg= NEW.NumStageF ; -- voir explication 6
    IF (nb=1) THEN /* le couple existe déjà dans la table Stagiaire, on bloque l'insertion et
on provoque une erreur*/
        SIGNAL SQLSTATE "45000" SET MESSAGE_TEXT ="opération d'insertion impossible" ;
        /* dans le cas contraire l'insertion va se produire après la fin du trigger*/
    END IF ;
END // -- le TRIGGER se termine grâce au nouveau délimiteur.
DELIMITER ; -- on restaure l'ancien délimiteur.
```

Explications :

- **Explication 1** : le trigger est d'abord supprimé s'il existe déjà (on ne peut pas mettre à jour la structure d'un trigger. Il faut le supprimer puis le redéfinir de nouveau. Certain SGBDR permettent de modifier la structure d'un Trigger comme c'est le cas du SGBDR SQL-Server qui par la commande **ALTER TRIGGER** peut modifier la structure initiale du Trigger, cependant **MySQL ne permet pas** de faire ce genre de manipulation. Notez bien qu'on ne peut pas avoir plus d'un trigger **Avant_insertion** sur la même table, mais on peut avoir un Trigger qui fait plusieurs actions à la fois.
- **Explication 2** : Avant la création du trigger, il faut changer de délimiteur pour le Trigger par l'instruction : **DELIMITER //**. Ceci vient du fait que le code du trigger utilise des « ; » comme délimiteur de **bloc d'instructions** alors même que le « ; »

annonce la fin (délimiteur) d'instruction standard du SQL. Il est donc important de changer de délimiteur du **bloc d'instructions** du Trigger pour que la rencontre du premier « ; » à l'intérieur du trigger ne se soit pas interpréter comme la fin du trigger mais plutôt comme la fin d'une d'instruction SQL.

- **Explication 3 :** Avant toute insertion dans la table Formateur il faut contrôler que le couple (**Numéro enseignant /numéro stage**) n'existe pas déjà dans la table stagiaire. Ainsi, dès que le SGBDR reçoit un ordre **INSERT** sur la table « Formateur », il va exécuter le trigger avant même de faire l'opération **INSERT**. Il est possible que la commande d'INSERTION soit bloquée par le SGBDR si le trigger génère une erreur due à la non vérification de la condition. Si le trigger ne génère aucune erreur, l'opération d'insertion dans la table Formateur se déroulera à la fin du Trigger. Il faut toujours penser à donner un nom significatif au Trigger qui permet de comprendre le moment de déclenchement du Trigger, l'événement de déclenchement du trigger et la table à la quelle il est liée (dans notre cas, la lecture nom du trigger **Avant_insertion_Formateur** nous permet de comprendre directement que c'est un Trigger liée à la table Formateur et que ce Trigger se déclenche automatiquement avant une opération d'insertion dans la table Formateur).
- **Explication 4 :** On doit spécifier le moment ou le trigger s'exécute (ici c'est avant une opération d'insertion sur la table Formateur).
- **Explication 5 :** c'est pour dire que le trigger va s'exécuter sur toutes les lignes insérées par la commande **INSERT** (il est possible que plusieurs insertions de lignes se fasse dans la même commande **INSERT**).
- **Explication 6 :** cette requête retourne soit 1 s'il existe un stagiaire qui à le même couple (**Numéro enseignant /numéro stage**) que le formateur qu'on veut insérer, et il faut à ce moment la bloqué la commande d'insertion. Soit retourne 0 et dans ce cas la l'insertion est permise et se produira juste après la fin du trigger.
- **Explication 7 :** **NEW** représente le nouveau tuple qu'on essaie d'insérer dans la table «Formateur». **NEW** est un mot-clé qui permet d'accéder aux nouvelles valeurs du tuple qu'on est en train d'ajouter ou de modifier. **NEW** s'utilise uniquement dans les trigger **INSERT** et **UPDATE**. En cas de **DELETE**, il n'y a pas de nouvelles valeurs, il n'ya que d'anciennes valeurs référencés par le mot clé **OLD**. **OLD** est un mot-clé qui permet d'accéder aux anciennes valeurs du tuple qu'on est en train de modifier ou de supprimer. **OLD** s'utilise dans les trigger **UPDATE** et **DELETE** uniquement. En cas d'**INSERT**, il n'y a pas d'anciennes valeurs.

Test :

```
INSERT INTO Stagiaire (NumEnsStg, NumStageStg) VALUES ('1', '1') ;
```

Tentative d'insertion dans « Formateur » :

```
INSERT INTO Formateur (NumEnsF, NumStageF) VALUES ('2', '2') ;
```

Cette insertion se passe bien puisque la contrainte est vérifiée.

```
INSERT INTO Formateur (NumEnsF, NumStageF) VALUES ('1', '2') ;
```

Cette insertion se passe bien puisque la contrainte est vérifiée.

```
INSERT INTO Formateur (NumEnsF, NumStageF) VALUES ('1', '1') ;
```

Une erreur se produit car le couple (1,1) est déjà présent dans **Stagiaire** : l'insertion dans la table **Formateur** est bloqué par un message d'erreur.

Dans le cas de modification d'un formateur, cette vérification doit être également faite :

```
DROP TRIGGER IF EXISTS Avant_Modification_Formateur ;
DELIMITER // -- le délimiteur du bloc d'instructions devient //.
CREATE TRIGGER Avant_Modification_Formateur
BEFORE UPDATE -- avant toute modification de tuples sur la table Formateur.
ON Formateur
FOR EACH ROW
BEGIN
    DECLARE nb INTEGER;
    SELECT COUNT(*) INTO nb /*le résultat de la requête est récupéré dans la variable nb*/
    FROM Stagiaire
    WHERE NumEnsStg = NEW.NumEnsF
    AND NumStageStg= NEW.NumStageF;
    IF (nb=1) THEN /* le couple existe déjà dans la table Stagiaire, on bloque la
modification et on provoque une erreur*/
        SIGNAL SQLSTATE "45000" SET MESSAGE_TEXT ="opération de modification
impossible" ;
        /* dans le cas contraire la modification va se produire après la fin du trigger*/
    END IF ;
END // -- le TRIGGER se termine grâce au nouveau délimiteur.
DELIMITER ; -- on restaure l'ancien délimiteur.
```

Test :

```
INSERT INTO Stagiaire (NumEnsStg, NumStageStg) VALUES ('3', '6') ;
```

Tentative d'insertion dans « Formateur » :

```
INSERT INTO Formateur (NumEnsF, NumStageF) VALUES ('3', '4') ;
```

Cette insertion se passe bien puisque la contrainte est vérifiée.

```
UPDATE Formateur
```

```
SET NumStageF = '5'
```

```
WHERE NumEnsF = '3' AND NumStageF = '4';
```

Cette modification se passe bien puisque la contrainte est vérifiée.

```
UPDATE Formateur
```

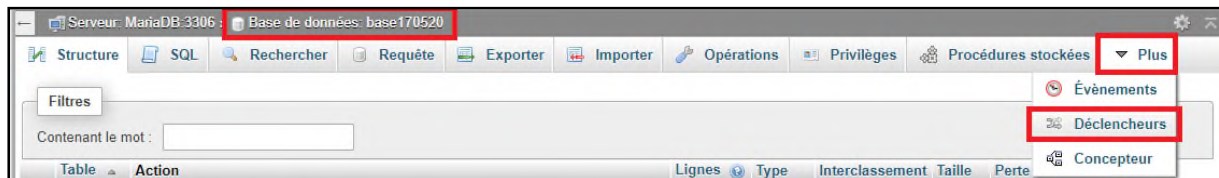
```
SET NumStageF = '6'
```

```
WHERE NumEnsF = '3' AND NumStageF = '5';
```

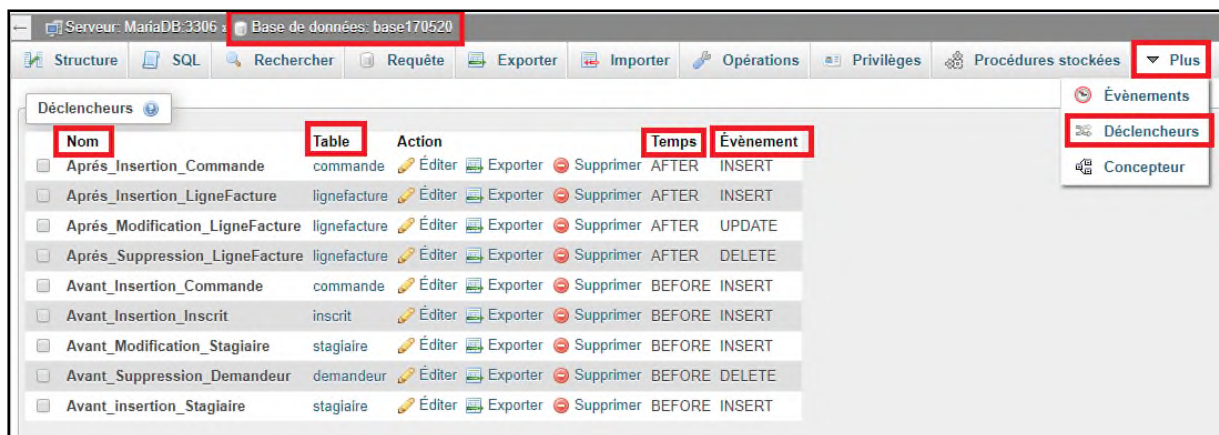
Une erreur se produit car le couple (3,6) est déjà présent dans **Stagiaire** : la modification dans la table **Formateur** est bloqué par un message d'erreur.

Du coté de la table « **Stagiaire** », on va également programmer le SGBDR pour que lors de l'insertion d'un nouveau stagiaire pour un stage (ou modification de stagiaire), il faut

contrôler que ce stagiaire n'est pas formateur dans ce stage. Cette vérification doit se faire avant l'insertion d'un nouveau stagiaire dans la table « Stagiaire » et également avant la modification d'un tuple dans la table « Stagiaire ». PhpMyAdmin propose une interface graphique pour la création d'un Trigger (pour les opérations d'insertion et de modification dans la table Stagiaire, nous allons créer des Triggers à l'aide de cette interface). Pour y accéder, allez à la vue globale de la base de donnée, volet « Plus », option « Déclencheur » :



En cliquant dessus (option « Déclencheur »), vous avez la liste des Triggers (déclencheurs) établis. Pour chaque Trigger (déclencheur) nous avons le nom du déclencheur, la table à laquelle il est lié, quand le Trigger s'exécute et l'événement déclencheur du Trigger :



Trigger **Avant_Insertion_Stagiaire** dans la table « Stagiaire » : ce Trigger a pour objectif de contrôler si un couple (**NumEns, NumStage**) n'existe pas déjà dans la table « Formateur » avant de faire l'opération d'insertion de ce couple dans la table « Stagiaire ». Nous allons exploiter cette fois si l'interface proposée par PHPMyAdmin pour MySQL. Cette interface propose des options spécifiques qui facilitent la création du trigger.

Le code du Trigger **Avant_Insertion_Stagiaire** sans usage d'interface est le suivant :

```
DROP TRIGGER IF EXISTS Avant_insertion_Stagiaire ;
DELIMITER // -- le délimiteur du bloc d'instructions devient //.
CREATE TRIGGER Avant_insertion_Stagiaire
BEFORE INSERT -- avant toute insertion dans la table Stagiaire.
```

```

ON Stagiaire
FOR EACH ROW
BEGIN
    DECLARE nb INTEGER;
    SELECT COUNT(*) INTO nb /* le résultat de la requête est récupéré dans la variable nb*/
    FROM Formateur
    WHERE NumEnsF = NEW.NumEnsStg
    AND NumStageF = NEW.NumStageStg ;
    IF (nb=1) THEN /* le couple existe déjà dans la table Formateur, on bloque l'insertion
dans la table Stagiaire et on provoque une erreur*/
        SIGNAL SQLSTATE "45000" SET MESSAGE_TEXT ="opération d'insertion impossible" ;
        /* dans le cas contraire l'insertion va se produire après la fin du trigger*/
    END IF ;
END // -- le TRIGGER se termine grâce au nouveau délimiteur.
DELIMITER ; -- on restaure l'ancien délimiteur.

```

L'usage de l'interface de PHPMyAdmin permet de réduire ce code et d'écrire seulement le corps du Trigger (les autres informations peuvent être définies à travers l'interface). Dans PHPMyAdmin, vue globale de la base de données, le volet « plus », option « déclencheur », nous allons suivre les étapes suivantes sur l'interface :

- Assurer vous que le trigger n'existe pas déjà, dans le cas ou le trigger existe l'interface PHPMyAdmin vous permet de le modifier (contrairement à la manipulation en code SQL ou il faut toujours supprimer le Trigger puis le recréer de nouveau pour personnaliser le Trigger ou modifier son contenu).
- Cliquez sur « Ajouter un déclencheur ». Une interface apparaît.

Remplissez les différents champs : le Nom du déclencheur (dans notre cas le nom du déclencheur est **Avant_insertion_Stagiaire**), Table (la table à laquelle ce déclencheur est lié : **Stagiaire**), Moment (choisissez **BEFORE** dans notre cas), Événement (Mettez l'événement d'insertion **INSERT**). Dans le champ « définition » mettez le corps du Trigger (ses instructions) qui commence par « **BEGIN** » et se termine par « **END** » (remarque : dans le cas de la création du trigger à travers l'interface, il ne faut pas définir de délimiteur pour le bloc d'instructions). Dans notre cas le champ définition sera rempli avec :

```
BEGIN
DECLARE nb INTEGER;
SELECT COUNT(*) INTO nb
FROM Formateur
WHERE NumEnsF = NEW.NumEnsStg
AND NumStageF = NEW.NumStageStg ;
IF (nb=1) THEN /* le couple existe déjà dans la table Formateur, on
bloque l'insertion dans la table Stagiaire et on provoque une erreur*/
SIGNAL SQLSTATE "45000" SET MESSAGE_TEXT ="opération d'insertion
impossible" ;
/* dans le cas contraire l'insertion va se produire après la fin du trigger*/
END IF ;
END
```

Éditer le déclencheur

Détails

Nom du déclencheur: Avant_insertion_Stagiaire

Table: stagiaire

Moment: BEFORE

Évènement: INSERT

Définition:

```
1 BEGIN
2   DECLARE nb INTEGER;
3   SELECT COUNT(*) INTO nb
4   FROM Formateur
5   WHERE NumEnsF = NEW.NumEnsStg
6   AND NumStageF = NEW.NumStageStg ;
7   IF (nb=1) THEN
8     SIGNAL SQLSTATE "45000" SET MESSAGE_TEXT ="opération insertion impossible";
9   END IF ;
10  END
```

Créateur: root@localhost

Exécuter Fermer

Puis cliquez sur « exécuter » pour créer le trigger.

Test :

```
INSERT INTO Formateur (NumEnsF, NumStageF) VALUES ('72', '3');
```

Tentative d'insertion dans « Stagiaire » :

```
INSERT INTO Stagiaire (NumEnsStg, NumStageStg) VALUES ('72', '4');
```

Cette insertion se passe bien puisque la contrainte est vérifiée.

```
INSERT INTO Stagiaire (NumEnsStg, NumStageStg) VALUES ('73', '2');
```

Cette insertion se passe bien puisque la contrainte est vérifiée.

```
INSERT INTO Stagiaire (NumEnsStg, NumStageStg) VALUES ('72', '3');
```

Une erreur se produit car le couple (72,3) est déjà présent dans Formateur : l'insertion dans la table Stagiaire est bloqué par un message d'erreur.

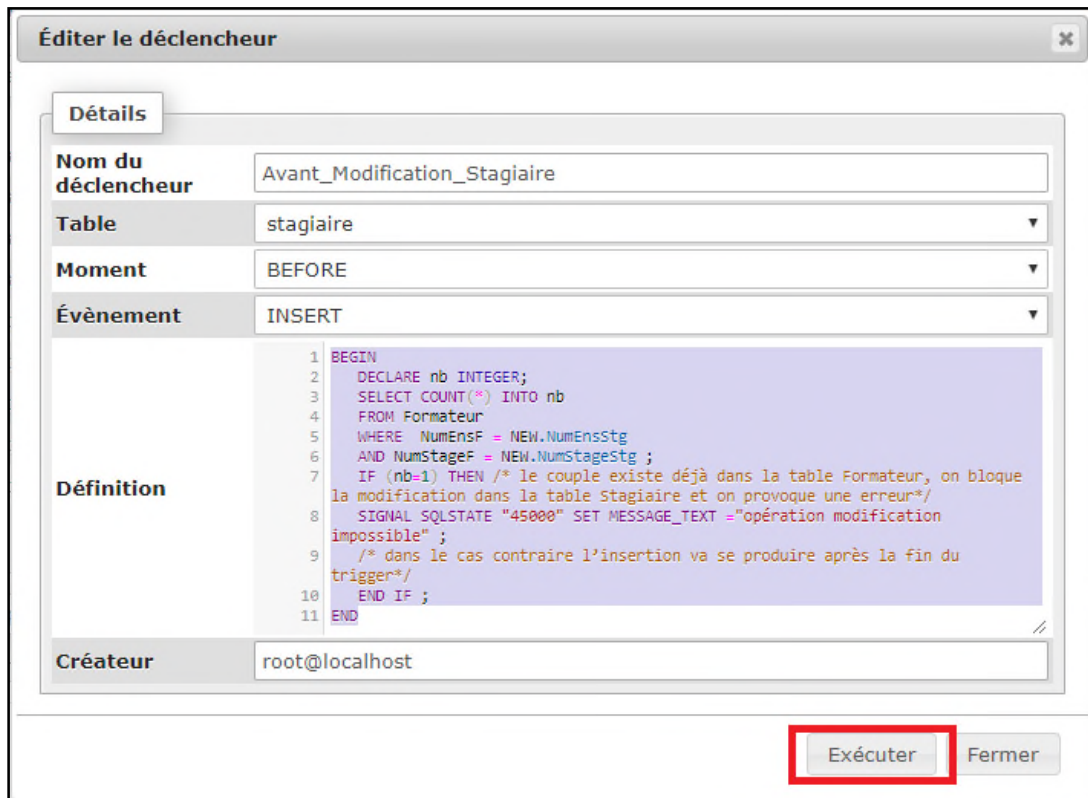
Trigger Avant_Modification_Stagiaire dans la table « Stagiaire » : ce Trigger à pour objectif de contrôler si un couples (NumEns, NumStage) n'existe pas déjà dans la table « Formateur » avant de faire l'opération de modification de ce couple dans la table « Stagiaire ». Nous allons également exploiter l'interface proposé par PHPMyAdmin pour MySQL. Cette interface propose des options spécifiques qui facilitent la création du trigger.

Le code du Trigger Avant_Modification_Stagiaire sans usage d'interface est le suivant :

```
DROP TRIGGER IF EXISTS Avant_Modification_Stagiaire ;
DELIMITER // -- le délimiteur du bloc d'instructions devient //.
CREATE TRIGGER Avant_Modification_Stagiaire
BEFORE UPDATE -- avant toute modification dans la table Stagiaire.
ON Stagiaire
FOR EACH ROW
BEGIN
    DECLARE nb INTEGER;
    SELECT COUNT(*) INTO nb /* le résultat de la requête est récupéré dans la variable nb*/
    FROM Formateur
    WHERE NumEnsF = NEW.NumEnsStg
    AND NumStageF = NEW.NumStageStg ;
    IF (nb=1) THEN /* le couple existe déjà dans la table Formateur, on bloque la
modification dans la table Stagiaire et on provoque une erreur*/
        SIGNAL SQLSTATE "45000" SET MESSAGE_TEXT ="opération modification
impossible" ;
        /* dans le cas contraire la modification va se produire après la fin du trigger*/
    END IF ;
END // -- le TRIGGER se termine grâce au nouveau délimiteur.
DELIMITER ; -- on restaure l'ancien délimiteur.
```

Si on veut créer le Trigger par la console de PHPMyAdmin, vous devez remplir les champs du nom du Trigger, sa chronologie (BEFORE/AFTER), type de déclenchement

d'un trigger (**INSERT** / **UPDATE** / **DELETE**) ainsi que la table à laquelle le trigger est lié. Il ne reste qu'à écrire le corps du trigger entre un **BEGIN** et un **END**.



Test :

```
INSERT INTO Formateur (NumEnsF, NumStageF) VALUES ('62', '4');  
INSERT INTO Formateur (NumEnsF, NumStageF) VALUES ('62', '5');
```

Tentative d'insertion dans « **Stagiaire** » :

```
INSERT INTO Stagiaire (NumEnsStg, NumStageStg) VALUES ('62', '6');
```

Cette insertion se passe bien puisque la contrainte est vérifiée.

UPDATE Stagiaire

```
SET NumStageStg = '3'  
WHERE NumEnsStg = '62' AND NumStageStg = '6';
```

Cette modification se passe bien puisque la contrainte est vérifiée.

UPDATE Stagiaire

```
SET NumStageStg = '4'  
WHERE NumEnsStg = '62' AND NumStageStg = '3';
```

Cette modification ne se passe pas bien puisque la contrainte n'est pas vérifiée.

Une erreur se produit car le couple (62,4) est déjà présent dans **Formateur**: la modification dans la table **Stagiaire** est bloqué par un message d'erreur.

Exemple2 d'utilisation de Trigger dans les insertions/modifications/suppressions de valeurs :

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».

La traduction de cette contrainte en langage SQL implique l'interdiction de l'insertion (**INSERT**) d'une personne dans un stage si cette personne n'est pas demandeur de se stage (ceci est valable également dans le cas de la modification **UPDATE** dans la table «Inscrit» : on ne peut pas modifier un inscrit si cette modification abouti à un inscrit qui n'est pas demandeur de stage). Ainsi lors de l'insertion d'un couple (Numéro Personne, Numéro Stage) dans la table «Inscrit», il faut s'assurer que ce couple existe dans la table «Demandeur» (ceci est également valable dans le cas de la modification **UPDATE** dans la table «Inscrit» : il faut s'assurer que la modification d'un inscrit abouti à un nouveau inscrit qui est déjà demandeur de stage). J'attire également votre attention sur le fait que lors de la suppression (**DELETE**) d'un demandeur de formation, il faut s'assurer que ce demandeur de formation n'est pas inscrit à la formation dans laquelle il sera supprimer (sinon l'inclusion de la relation «Inscrit» dans la relation «Demandeur» n'est plus respectée). Ceci est valable également pour le cas de modification (**UPDATE**) sur la table «Demandeur», ou il est n'est pas possible de modifier un demandeur qui est inscrit dans la table «Inscrit».

Les événements reliés à cette contrainte, nous mène à quatre triggers :

- 1/- Trigger avant **insertion** ou **modification** sur la table «Inscrit» qui contrôle si le couple (Numéro Personne, Numéro Stage) est présent dans la table «Demandeur».
- 2/- Trigger avant **suppression** ou **modification** sur la table «Demandeur» qui contrôle si le couple (Numéro Personne, Numéro Stage) n'existe pas dans la table «Inscrit».

Code trigger « Avant_Insertion_Inscrit » :

```
DROP TRIGGER IF EXISTS Avant_Insertion_Inscrit ;
DELIMITER // -- le délimiteur du bloc d'instructions devient //.
CREATE TRIGGER Avant_Insertion_Inscrit
BEFORE INSERT -- avant toute insertion dans la table Inscrit.
ON Inscrit
FOR EACH ROW
BEGIN
    DECLARE nb INT ;
    SELECT COUNT(*) INTO nb /*le résultat de la requête est récupéré dans la variable nb*/
    FROM Demandeur
    WHERE NumPersD = NEW.NumPersI
    AND NumStageD= NEW.NumStageI;
    IF (nb=0) THEN /* Si le couple n'existe pas déjà dans la table Demandeur, on bloque
l'opération d'insertion et on provoque une erreur*/
        SIGNAL SQLSTATE "45000" SET MESSAGE_TEXT ="opération d'insertion impossible" ;
        /* Dans le cas contraire, ou nb=1, le couple est déjà présent dans la table Demandeur et
l'opération d'insertion dans la table Inscrit se produira après la fin du trigger*/
    END IF ;
END // -- le TRIGGER se termine grâce au nouveau délimiteur.
DELIMITER ; -- on restaure l'ancien délimiteur.
```

Test :

```
INSERT INTO Demandeur (NumPersD, NumStageD) VALUES ('101', '1') ;
INSERT INTO Demandeur (NumPersD, NumStageD) VALUES ('125', '2') ;
```

Tentative d'insertion dans « Inscrit » :

```
INSERT INTO Inscrit (NumPersI, NumStageI) VALUES ('101', '1') ;
```

Cette insertion se passe bien puisque la contrainte est vérifiée.

```
INSERT INTO Inscrit (NumPersI, NumStageI) VALUES ('125', '2') ;
```

Cette insertion se passe bien puisque la contrainte est vérifiée.

```
INSERT INTO Inscrit (NumPersI, NumStageI) VALUES ('125', '6') ;
```

Une erreur se produit car le couple (125,6) n'est pas présent dans la table Demandeur :
l'insertion est bloqué dans la table Inscrit par un message d'erreur.

Code trigger « Avant_Modification_Inscrit » :

```
DROP TRIGGER IF EXISTS Avant_Modification_Inscrit ;
DELIMITER // -- le délimiteur du bloc d'instructions devient //.
CREATE TRIGGER Avant_Modification_Inscrit
BEFORE UPDATE -- avant toute modification dans la table Inscrit.
ON Inscrit
FOR EACH ROW
BEGIN
```

```

DECLARE nb INT ;
SELECT COUNT(*) INTO nb /*le résultat de la requête est récupéré dans la variable nb*/
FROM Demandeur
WHERE NumPersD = NEW.NumPersI
AND NumStageD= NEW.NumStageI;
IF (nb=0) THEN /* Si le couple n'existe pas déjà dans la table Demandeur, on bloque
l'opération de modification et on provoque une erreur*/
    SIGNAL SQLSTATE "45000" SET MESSAGE_TEXT ="opération de modification
impossible" ;
    /* Dans le cas contraire, ou nb=1, le couple est déjà présent dans la table Demandeur et
l'opération de modification dans la table Inscrit se produira après la fin du trigger*/
END IF ;
END // -- le TRIGGER se termine grâce au nouveau délimiteur.
DELIMITER ; -- on restaure l'ancien délimiteur.

```

Test :

```

INSERT INTO Demandeur (NumPersD, NumStageD) VALUES ('135', '9') ;
INSERT INTO Demandeur (NumPersD, NumStageD) VALUES ('135', '10') ;
INSERT INTO Inscrit (NumPersI, NumStageI) VALUES ('135', '9') ;

```

Tentative de modification dans « Inscrit » :

```

UPDATE Inscrit
SET NumStageI = '10'
WHERE NumPersI = '135' AND NumStageI = '9';

```

Cette modification se passe bien puisque la contrainte est vérifiée.

```

UPDATE Inscrit
SET NumStageI = '11'
WHERE NumPersI = '135' AND NumStageI = '10';

```

Une erreur se produit car le couple (135,11) n'est pas demandeur de stage (il n'existe pas dans la table Demandeur) : la modification dans la table Inscrit est bloqué par un message d'erreur.

Code trigger « Avant_Suppression_Demandeur » :

```

DROP TRIGGER IF EXISTS Avant_Suppression_Demandeur ;
DELIMITER // -- le délimiteur du bloc d'instructions devient //.
CREATE TRIGGER Avant_Suppression_Demandeur
BEFORE DELETE -- avant toute suppression dans la table Demandeur.
ON Demandeur
FOR EACH ROW
BEGIN
    DECLARE nb INT ;
    SELECT COUNT(*) INTO nb /*le résultat de la requête est récupéré dans la variable nb*/
    FROM Inscrit
    WHERE NumPersI = OLD.NumPersD

```

```

AND NumStageI= OLD.NumStageD; -- quand on fait un DELETE il ya juste des valeurs anciennes.
IF (nb=1) THEN /* il existe un inscrit qui porte le même numéro de personne et le
même numéro de stage que le demandeur qu'on cherche à supprimer, le couple existe
dans la table Inscrit, on bloque donc l'opération de suppression dans la table Demandeur
et on provoque une erreur*/
    SIGNAL SQLSTATE "45000" SET MESSAGE_TEXT ="opération de suppression
impossible" ;
    /* Dans le cas contraire, ou nb=0, le couple n'est pas dans la table Inscrit et l'opération
de suppression dans la table Demandeur se produira après la fin du trigger*/
END IF ;
END // -- le TRIGGER se termine grâce au nouveau délimiteur.
DELIMITER ; -- on restaure l'ancien délimiteur.

```

Test :

```

INSERT INTO Demandeur (NumPersD, NumStageD) VALUES ('121', '11') ;
INSERT INTO Demandeur (NumPersD, NumStageD) VALUES ('145', '11') ;
INSERT INTO Inscrit (NumPersI, NumStageI) VALUES ('145', '11') ;

```

Tentative de suppression dans « Inscrit » :

```
DELETE FROM Demandeur
```

```
WHERE NumPersD = '121' AND NumStageD= '11' ;
```

Cette suppression se passe bien puisque la contrainte est vérifiée.

```
DELETE FROM Demandeur
```

```
WHERE NumPersD = '145' AND NumStageD= '11' ;
```

Une erreur se produit car le couple (145,11) est déjà présent dans la table Inscrit : la suppression dans la table Demandeur est bloqué par un message d'erreur.

Code trigger « Avant_Modification_Demandeur » :

```
DROP TRIGGER IF EXISTS Avant_Modification_Demandeur ;
```

```
DELIMITER // -- le délimiteur du bloc d'instructions devient //.
```

```
CREATE TRIGGER Avant_Modification_Demandeur
```

```
BEFORE UPDATE -- avant toute modification dans la table Demandeur.
```

```
ON Demandeur
```

```
FOR EACH ROW
```

```
BEGIN
```

```
    DECLARE nb INT ;
```

```
    SELECT COUNT(*) INTO nb /*le résultat de la requête est récupéré dans la variable nb*/
```

```
    FROM Inscrit
```

```
    WHERE NumPersI = OLD.NumPersD
```

```
    AND NumStageI= OLD.NumStageD; /*on cherche si l'ancien demandeur qu'on veut
modifier est inscrit ou pas. Si l'ancien demandeur est inscrit alors la modification est
impossible. Ainsi nous testons les anciennes valeurs et non pas les nouvelles valeurs
qu'on veut mettre */
```

IF (nb=1) THEN /* il existe inscrit qui porte le même numéro de personne et le même numéro de stage que le demandeur qu'on cherche à modifier, le couple existe dans la table Inscrit, on bloque donc l'opération de modification dans la table Demandeur et on provoque une erreur*/

SIGNAL SQLSTATE "45000" SET MESSAGE_TEXT ="opération de modification impossible" ;

/* Dans le cas contraire, ou nb=0, le couple n'est pas dans la table Inscrit et l'opération de modification dans la table Demandeur se produira après la fin du trigger*/

END IF ;

END // -- le TRIGGER se termine grâce au nouveau délimiteur.

DELIMITER ; -- on restaure l'ancien délimiteur.

Test :

INSERT INTO Demandeur (NumPersD, NumStageD) VALUES ('162', '2') ;

INSERT INTO Demandeur (NumPersD, NumStageD) VALUES ('69', '7') ;

INSERT INTO Inscrit (NumPersI, NumStageI) VALUES ('69', '7') ;

Tentative de modification dans « Demandeur » :

UPDATE Demandeur

SET NumStageD = '3'

WHERE NumPersD = '162' AND NumStageD = '2';

Cette modification se passe bien puisque la contrainte est vérifiée.

UPDATE Demandeur

SET NumStageD = '8'

WHERE NumPersD = '69' AND NumStageD = '7';

Une erreur se produit car le couple (69,7) est déjà inscrit donc on ne peut pas modifier ce couple dans la table demandeur (sinon nous obtenons un inscrit qui n'est pas demandeur de stage) : la modification dans la table Demandeur est bloqué par un message d'erreur.

D'autres usage de triggers:

Les Triggers ne servent pas uniquement à bloquer ou à autoriser des opérations, ils peuvent servir d'autres objectifs. Ci-dessus, on donne quelques exemples d'utilisation des Triggers dans d'autres tâches :

Exemple1 : de mise à jour d'un totale par rapport à des insertions de valeurs :

Soit les relations suivantes :

Produit (**NomProduit**, QuantitéProdEnStock)

Commande (**NumCom**, DateCom, NomProduitCom, QuantitéCom)

Un fournisseur de produits dispose de ces deux tables 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 de la quantité en stock de ce produit. Il faut ainsi définir un Trigger qui soit à l'écoute d'un événement d'insertion sur la table Commande et qui va permettre après cette insertion de diminuer la quantité du produit commandé de la valeur de quantité présente en stock pour ce même produit (nous mettons ainsi à jour la quantité réelle du produit présente en stock).

```
DROP TRIGGER IF EXISTS Après_Insertion_Commande;
DELIMITER // -- le délimiteur du bloc d'instructions devient //.
CREATE TRIGGER Après_Insertion_Commande
AFTER INSERT -- après toute insertion dans la table Commande ce trigger se déclenche.
ON Commande
FOR EACH ROW
BEGIN
    UPDATE Produit /*Modifié dans la table Produit*/
    SET QuantitéProdEnStock = QuantitéProdEnStock - NEW.QuantitéCom
    WHERE NomProduit = NEW.NomProduitCom ; /* cette mise à jour concerne seulement
un produit et non pas tous les produits de la table Produit*/
/* la valeur de la quantité de produit commandé est réduite de la quantité en stock. La
quantité en stock du produit commandé va diminuée de la quantité commandée*/
/* le Trigger ne bloque jamais l'insertion mais utilise cette insertion comme
événement pour mettre à la quantité du produit en stock*/
END // -- le TRIGGER se termine grâce au nouveau délimiteur.
DELIMITER ; -- on restaure l'ancien délimiteur.
```

Test :

```
INSERT INTO Produit (NomProduit, QuantitéProdEnStock)
VALUES
('Masque protection tissu',100000),
('Masque de protection type FFP1',15000),
('Masque de protection type FFP2',30000);
```

Table: produit

Affichage des lignes 0 - 2 (total de 3, traitement en 0,0004 seconde(s).)

SELECT * FROM `produit`

Profilage [Éditer en ligne] [Éditer en ligne]

Tout afficher | Nombre de lignes : 25 | Filtrer les lignes: Chercher dans cette table | Trier sur l'index:

+ Options

	NomProduit	QuantitéProdEnStock
Éditer Copier Supprimer	Masque de protection type FFP1	15000
Éditer Copier Supprimer	Masque de protection type FFP2	30000
Éditer Copier Supprimer	Masque protection tissu	100000

INSERT INTO Commande (NumCom, DateCom, NomProduitCom, QuantitéCom)**VALUES** (1,'2020-05-18','Masque de protection type FFP1',5000) ;

Après exécution de cette insertion le nombre de marque de protection de type FFP1 disponible en stock est diminué de 5000 marques (ce qui veut dire que la quantité disponible en stock est de 10000 masques) :

Table: produit

Affichage des lignes 0 - 2 (total de 3, traitement en 0,0004 seconde(s).)

SELECT * FROM `produit`

Profilage [Éditer en ligne] [Éditer en ligne]

Tout afficher | Nombre de lignes : 25 | Filtrer les lignes: Chercher dans cette table | Trier s

+ Options

	NomProduit	QuantitéProdEnStock
Éditer Copier Supprimer	Masque de protection type FFP1	10000
Éditer Copier Supprimer	Masque de protection type FFP2	30000
Éditer Copier Supprimer	Masque protection tissu	100000

Notons au passage qu'il existe un autre point auquel il faut faire attention, c'est le fait de ne pas pouvoir enregistrer une commande dont la quantité est supérieure à la quantité en stock. Il faut définir un Trigger qui s'exécute automatiquement lorsqu'un événement d'insertion de commande (avant insertion) se produit sur la table commande et qui permet d'éviter d'insérer une commande si la quantité commandée est supérieure à la quantité en stock (notons que ce Trigger a pour objectif de bloquer ou de permettre l'exécution d'une commande d'insertion sur la table Commande. Malgré que cette partie n'est pas réservée à ce genre de Triggers, il fallait attirer votre attention comme même sur ce cas de figure qui rejoint l'ensemble des triggers ayant à pour but de vérifier le respect d'une contrainte avant d'exécuter une opération) :


```

DROP TRIGGER IF EXISTS Avant_Insertion_Commande;
DELIMITER // -- le délimiteur du bloc d'instructions devient //.
CREATE TRIGGER Avant_Insertion_Commande
BEFORE INSERT -- avant toute insertion dans la table Commande ce trigger se déclenche.
ON Commande
FOR EACH ROW
BEGIN
    DECLARE QT INT ;
    SET QT = (SELECT QuantitéProdEnStock
              FROM Produit
              WHERE NomProduit = NEW.NomProduitCom) ;
    IF (QT < NEW.QuantitéCom) THEN
        SIGNAL SQLSTATE "45000" SET MESSAGE_TEXT = "opération impossible : quantité
en stock insuffisante";
        /* Dans le cas contraire, l'opération d'insertion dans la table Commande se produira
après la fin du trigger*/
    END IF ;
END // -- le TRIGGER se termine grâce au nouveau délimiteur.
DELIMITER ; -- on restaure l'ancien délimiteur.

```

Test :

Le nombre de masque de protection type FFP2 disponible en stock est de 30000 masques. Nous exécutons la commande suivante :

```

INSERT INTO Commande (NumCom, DateCom, NomProduitCom, QuantitéCom) VALUES
(2,'2020-05-18','Masque de protection type FFP2',70000) ;

```

Cette insertion génère une erreur car la quantité de masque de protection type FFP2 commandée est supérieure à la quantité de ce type de masque disponible en stock.

Nous avons donc réalisé deux Triggers, le premier s'assure que la quantité du produit en stock est suffisante pour passer commande (vérification avant insertion dans la table commande) et le second permettras de mettre à jour la quantité en stock après insertion dans la table commande.

Challenge (BONUS EN JEU pour les 5 premières bonnes réponses) :

Nous avons deux nouvelles contraintes que nous voulons prendre en considération :

- 1/- la suppression d'une commande (cette suppression engendre la mise à jour de la quantité en stock du produit)..... Créez un trigger qui prend en charge cette situation.
- 2/- la modification d'une commande (la modification de la quantité d'un produit engendre la mise à jour de la quantité en stock du produit et la vérification de la quantité du produit commandé)..... Créez un trigger qui prend en charge cette situation.

Exemple2 : utilisation des triggers pour le calcul du cumul

Soit les relations :

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 après chaque :

- **Insertion de nouvelles lignes** (donc de nouvel article) dans la facture (**ici il ne s'agit pas de bloquer ou de permettre l'insertion mais de mettre à jour le total de la facture après chaque insertion d'une nouvelle ligne d'article dans la facture**).
- **Modification d'une ligne** (modifié la quantité d'un article acheté dans une ligne de la facture implique la mise à jour du total de la facture. **Ici il ne s'agit pas de bloquer ou de permettre la modification mais de mettre à jour le total de la facture après la modification de la quantité d'un article dans une ligne d'article de la facture**)
- **Suppression d'une ligne de la facture** (supprimer un article de la facture implique la suppression d'une ligne de la facture et la mise à jour du total de la facture. Il ne s'agit pas de bloquer ou de permettre l'opération de suppression mais **de mettre à jour le total de la facture après chaque suppression d'une ligne d'article dans la facture**).

Remarquez que dans la table « **Article** » nous avons le prix unitaire de chaque article, dans la table « **LigneFacture** » nous avons la Quantité achetée pour cet article et dans la table « **Facture** » nous avons le total de tous les articles achetés (le total de toutes les lignes de la factures). Remarquez également que la table « **LigneFacture** » comporte le numéro de facture, le numéro de l'article et sa quantité et peut obtenir le prix unitaire de chaque article à partir de la table « **Article** ».

Les Triggers seront établies sur la table « LigneFacture » pour les trois cas d'insertion (**INSERT**), de modification (**UPDATE**) et de suppression (**DELETE**) à fin de mettre à jour le total d'une facture.

Code du Trigger **Après_Insertion_LigneFacture** :

```
DROP TRIGGER IF EXISTS Après_Insertion_LigneFacture;
DELIMITER // -- le délimiteur du bloc d'instructions devient //.
CREATE TRIGGER Après_Insertion_LigneFacture
AFTER INSERT -- après toute insertion dans la table LigneFacture.
ON LigneFacture
FOR EACH ROW
```

```

BEGIN
UPDATE Facture /*Modifié table Facture*/
SET TotalFac = TotalFac + ((SELECT PrixUnitaireArt
                             FROM Article
                             WHERE NumArt = NEW.NumArtL) * NEW.Quantité)
WHERE NumFac = NEW.NumFacL ; /* cette mise à jour concerne une seule facture et
non pas toute les factures*/
/* le Trigger ne bloque pas l'insertion mais utilise cette insertion comme événement
pour mettre à jour le total de la facture*/
END // -- le TRIGGER se termine grâce au nouveau délimiteur.
DELIMITER ; -- on restaure l'ancien délimiteur.

```

Dans un trigger, on peut faire référence **soit** aux nouvelles valeurs insérés (commande d'insertion) ou modifiés (commande de modification) avec **NEW.Nom_champ**, **soit** faire référence aux anciennes valeurs supprimés (commande de suppression) ou modifiés (commande de modification) avec **OLD.Nom_champ**. La manipulation des objets **NEW** et **OLD** doit être faite avec le plus grand soin.

Test :

```

INSERT INTO Article (NumArt, NomArt, PrixUnitaireArt)
VALUES
(1, 'Sèche-cheveux',3400),
(2, 'Lisseur-cheveux',1500),
(3, 'Tondeuses-cheveux-barbe',6200),
(4, 'Mixeur',1850),
(5, 'Machine à café',7200),
(6, 'Fer à repasser simple',3000),
(7, 'Fer à repasser vapeur',5400),
(8, 'Micro-ondes',8300),
(9, 'Hachoir à viande électrique',12000),
(10, 'Machine à coudre à main',1700),
(11, 'Brosse a cheveux électrique',1950),
(12, 'Hachoir à viande manuel',4000);

```

```

INSERT INTO Facture (NumFac, DateFac, TotalFac)
VALUES
(1,'2020-05-17', 0),
(2,'2020-05-20', 0),
(3,'2020-05-23', 0) ;

```

Exécutant la commande d'insertion suivante :

```
INSERT INTO LigneFacture (NumFacL, NumArtL, Quantité)
VALUES
```

```
(1,1, 2),
(1,2, 3),
(1,3, 1),
(1,11, 2);
```

/*Nous avons ajouté à la facture 1 : 2 Sèche-cheveux à 3400 DA chacun (un pour monsieur et un pour madame) + 3 Lisseur-cheveux à 1500 DA chacun (pour madame)+ 1 Tondeuses-cheveux-barbe à 6200 DA (pour monsieur)+ 2 Brosse a cheveux électrique à 1950 chacune (une pour monsieur et une pour madame) pour un totale de 21400 DA*/

Après exécution de la commande d'insertion le Total de la facture Numéro 1 devient automatiquement 21400 DA :



NumFac	DateFac	TotalFac
1	2020-05-17	21400
2	2020-05-20	0

Nous avons automatiquement changé le total de la facture 1 avec l'aide du Trigger. Lorsqu'on fait une requête d'insertion sur la table « LigneFacture », le Trigger s'exécute automatiquement pour mettre à jour le total de la facture.

Code du Trigger **Après_Modification_LigneFacture** :

```
DROP TRIGGER IF EXISTS Après_Modification_LigneFacture;
DELIMITER // -- le délimiteur du bloc d'instructions devient //.
CREATE TRIGGER Après_Modification_LigneFacture
AFTER UPDATE -- après toute modification dans la table LigneFacture.
ON LigneFacture
FOR EACH ROW
BEGIN
```

```
UPDATE Facture /*Modifié table Facture*/
SET TotalFac = TotalFac - ((SELECT PrixUnitaireArt
FROM Article
WHERE NumArt = OLD.NumArtL) * OLD.Quantité)
+ ((SELECT PrixUnitaireArt
```

```

FROM Article
WHERE NumArt = NEW.NumArtL) * NEW.Quantité)
WHERE NumFac = NEW.NumFacL ; /* cette mise à jour concerne une facture et non pas
toute les factures*/
/* le Trigger ne bloque pas la modification mais utilise cette modification comme
événement pour mettre à jour le total de la facture*/
END // -- le TRIGGER se termine grâce au nouveau délimiteur.
DELIMITER ; -- on restaure l'ancien délimiteur.

```

Test :

Exécutant la commande d'insertion suivante :

```

INSERT INTO LigneFacture (NumFacL, NumArtL, Quantité)
VALUES
(2,4, 1),
(2,5, 3),
(2,6, 1),
(2,8, 2) ;
/*Nous avons ajouté à la facture 2 : 1 Mixeur à 1850 DA chacun + 3 Machines à café à
7200 DA chacun + 1 Fer à repasser simple à 3000 DA + 2 Micro-ondes à 8300 chacun
pour un totale de 43050 DA*/

```

Table: facture

Affichage des lignes 0 - 1 (total de 2, traitement en 0,0004 seconde(s).)

SELECT * FROM `facture`

Options: NumFac, DateFac, TotalFac

	NumFac	DateFac	TotalFac
1	1	2020-05-17	21400
2	2	2020-05-20	43050

Supposons que monsieur change d'avis avant sa livraison et qu'il décide de prendre 1 Machine à café au lieu de 3 :

```

UPDATE LigneFacture
SET Quantité = 1
WHERE NumFacL = 2 AND NumArtL = 5;

```

Après exécution de cette requête le total de la facture 2 passe à = $43050 - (3 \times 7200) + (1 \times 7200) = 28650$.

←

Serveur: MariaDB:3306 » Base de données: base170520 » Table: facture

Parcourir

Structure

SQL

Rechercher

Insérer

Exporter

Importer

Affichage des lignes 0 - 1 (total de 2, traitement en 0,0003 seconde(s).)

SELECT * FROM `facture`

☐ Profilage

[Éditer en ligne]

☐ Tout afficher

Nombre de lignes : 25

Filtrer les lignes: Chercher dans cette table

Trier sur l

+ Options

← T →

▼

NumFac

DateFac

TotalFac

☐ Éditer

☐ Copier

☐ Supprimer

1

2020-05-17

21400

☐ Éditer

☐ Copier

☐ Supprimer

2

2020-05-20

28650

↑

☐ Tout cocher

Avec la sélection :

☐ Éditer

☐ Copier

☐ Supprimer

Exporter

Puisque monsieur à fait des économies sur les machines à café, madame à revue ces exigences en hausse et décider de changer son fer à repasser simple (code produit 6) à 3000DA par un fer à repasser à vapeur (code produit 7) à 5400DA avant la livraison de leurs achats :

UPDATE LigneFacture

SET Quantité = 1, NumArtL = 7

WHERE NumFacL = 2 AND NumArtL = 6;

Après exécution de cette requête le total de la facture 2 passe à = $28650 - (1 \times 3000) + (1 \times 5400) = 31050$.

←

Serveur: MariaDB:3306 » Base de données: base170520 » **Table: facture**

Parcourir

Structure

SQL

Rechercher

Insérer

Exporter

Importer

Affichage des lignes 0 - 1 (total de 2, traitement en 0,0006 seconde(s).)

SELECT * FROM `facture`

☐ Profilage

[Éditer en ligne]

☐ Tout afficher

Nombre de lignes : 25

Filtrer les lignes: Chercher dans cette table

Trier sur

+ Options

←T→

▼

NumFac

DateFac

TotalFac

☐

Éditer

Copier

Supprimer

1

2020-05-17

21400

☐

Éditer

Copier

Supprimer

2

2020-05-20

31050

↑

☐ Tout cocher

Avec la sélection :

Éditer

Copier

Supprimer

Exporter

← Serveur: MariaDB:3306 » Base de données: base170520 » Table: lignefacture

Parcourir Structure SQL Rechercher Insérer Exporter Importer

✓ Affichage des lignes 0 - 7 (total de 8, traitement en 0,0004 seconde(s).)

SELECT * FROM `lignefacture`

Profilage [Éditer en ligne]

Tout afficher | Nombre de lignes : 25 | Filtrer les lignes: Chercher dans cette table | Trier sur

+ Options

		NumFacL	NumArtL	Quantité
<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer		1	1	2
<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer		1	2	3
<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer		1	3	1
<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer		1	11	2
<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer		2	4	1
<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer		2	5	1
<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer		2	7	1
<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer		2	8	2

Vous remarquez que le produit de code 6 (fer à repasser simple) n'existe plus dans la table « LigneFacture » et à été remplacé par le produit de code 7 (fer à repasser à vapeur) dans la même table.

Lorsqu'on fait une requête de modification sur la table « LigneFacture », le Trigger s'exécute automatiquement pour mettre à jour le total de la facture.

Code du Trigger **Après_Suppression_LigneFacture** :

```

DROP TRIGGER IF EXISTS Après_Suppression_LigneFacture;
DELIMITER // -- le délimiteur du bloc d'instructions devient //.
CREATE TRIGGER Après_Suppression_LigneFacture
AFTER DELETE -- après toute suppression dans la table LigneFacture.
ON LigneFacture
FOR EACH ROW
BEGIN
    UPDATE Facture /*Modifié table Facture*/
    SET TotalFac = TotalFac - ((SELECT PrixUnitaireArt
                                FROM Article
                                WHERE NumArt = OLD.NumArtL) * OLD.Quantité)
    WHERE NumFac = OLD.NumFacL ; /* cette mise à jour concerne une facture et non pas
toute les factures*/
    /* le Trigger ne bloque pas la suppression mais utilise cette suppression comme
événement pour mettre à jour le total de la facture*/
END // -- le TRIGGER se termine grâce au nouveau délimiteur.
DELIMITER ; -- on restaure l'ancien délimiteur.

```


Test :

Exécutant la commande d'insertion suivante :

```
INSERT INTO LigneFacture (NumFacL, NumArtL, Quantité)
VALUES
(3,9, 2),
(3,12, 1) ;
```

/*Nous avons ajouté à la facture 3 : 2 Hachoir à viande électrique à 12000 DA chacun + 1 Hachoir à viande manuel à 4000 DA pour un totale de 28000 DA*/

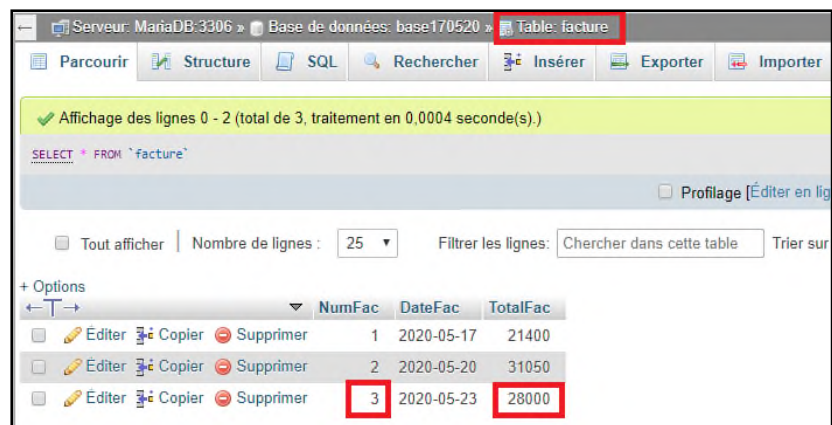


Table: facture

Affichage des lignes 0 - 2 (total de 3, traitement en 0,0004 seconde(s).)

SELECT * FROM `facture`

Options: NumFac, DateFac, TotalFac

	NumFac	DateFac	TotalFac
<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer	1	2020-05-17	21400
<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer	2	2020-05-20	31050
<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer	3	2020-05-23	28000

Nous décidons maintenant de supprimer la commande Hachoir à viande manuel de la facture 3 pour ne garder que celui qui est électrique :

```
DELETE FROM LigneFacture
WHERE NumFacL = 3 AND NumArtL = 12 ;
```

Après l'exécution de cette commande de suppression le total de la facture 3 passe à = 28000 - (1*4000) = 24000 DA.

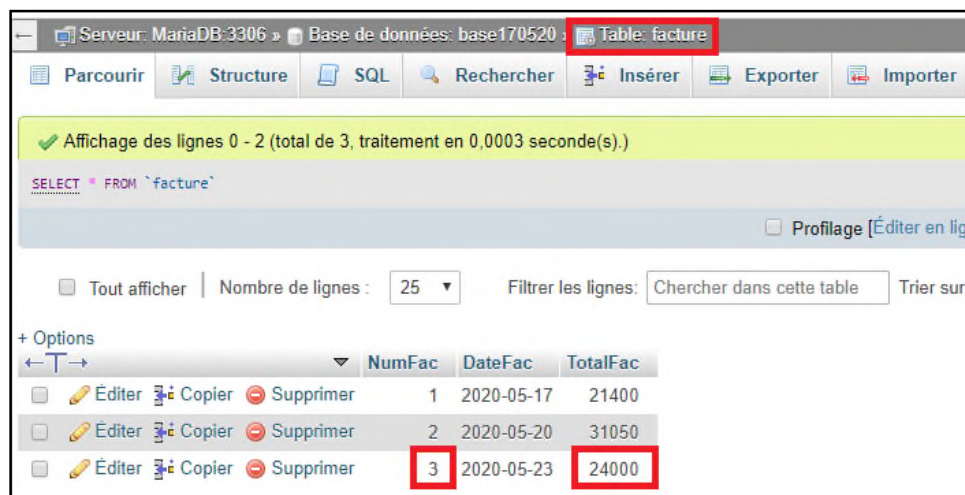


Table: facture

Affichage des lignes 0 - 2 (total de 3, traitement en 0,0003 seconde(s).)

SELECT * FROM `facture`

Options: NumFac, DateFac, TotalFac

	NumFac	DateFac	TotalFac
<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer	1	2020-05-17	21400
<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer	2	2020-05-20	31050
<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer	3	2020-05-23	24000

Lorsqu'on fait une requête de suppression sur la table « **LigneFacture** », le Trigger s'exécute automatiquement pour mettre à jour le total de la facture.

Procédure stockée et fonction stockée:

Une procédure stockée est un programme SQL intégré dans une base de données mais qui doit être appelé pour être exécuter (ce n'est pas la même logique avec un Trigger qui se déclenche suite à un événement).

Intérêt d'une procédure stockée : une procédure stockée permet de mémoriser des traitements directement dans la base de données, ce qui permet de :

- Limiter les échanges de flux entre serveur de traitement et le serveur de données.
- Offrir la possibilité de réutilisation des programmes (éviter les répétitions de code et optimiser le code).

Une procédure stockée peut être appelée par :

- Une autre procédure stockée (pour éviter les répétitions de code).
- Un trigger (pour optimiser les triggers offrant certains traitements similaires).
- Un programme extérieur (un programme PHP qui appelle une procédure stockée).

Nous reprenons l'exemple du Trigger qui permet d'éviter qu'un enseignant pour un stage donné soit à la fois formateur et stagiaire. Nous devons écrire deux Triggers par table :

- Un Trigger avant insertion et avant modification dans la table « Formateur »
- Un Trigger avant insertion et avant modification dans la table « Stagiaire »

Avec comme relations :

Enseignant (**NumEns**, NomEns, TélEns)

Stage (**NumStage**, LibelléStg, DuréeStg)

Formateur (**NumEnsF**, **NumStageF**)

Stagiaire (**NumEnsStg**, **NumStageStg**)

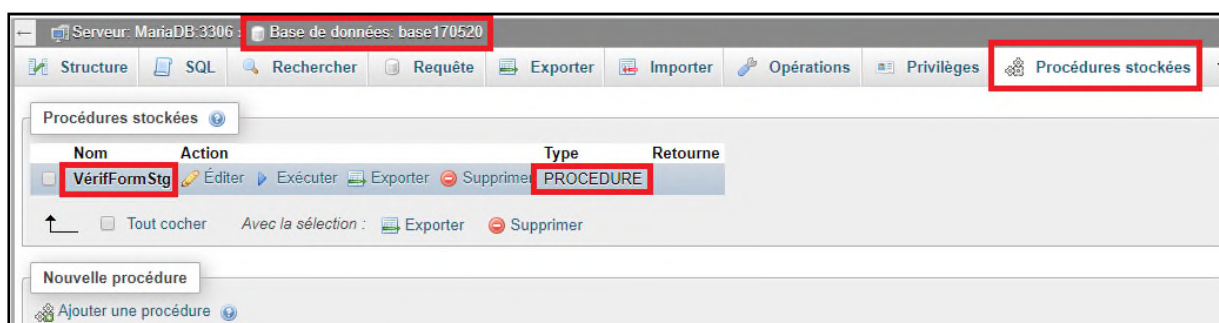
Comme nous l'avons vu précédemment pour la table « Formateur », le Trigger avant insertion et avant modification comporte le même code. Dans une optique d'optimisation, écrire deux Triggers avec des codes qui se ressemblent n'est pas optimal. La solution est d'écrire une procédure stockée unique qui sera appelée par les deux Triggers.

Attention : une procédure stockée ne peut pas accéder aux objets **NEW** et **OLD** contrairement à un Trigger (chose qui peut être faite par un trigger). Ceci implique l'obligation d'utiliser des paramètres dans une procédure stockée qui vont remplacer les objets **NEW** et **OLD** accessibles que par un Trigger.

La procédure stockée qui va contenir la portion de code commune entre le Trigger « **avant insertion Formateur** » et le Trigger « **avant modification Formateur** » est créée comme suit :

```
DELIMITER | -- le délimiteur du bloc d'instructions devient |.
CREATE PROCEDURE VérifFormStg (IN NumEnsProc INT, IN NumStagProc INT)
/* cette procédure comporte deux paramètres en entrée qui sont les deux objets NEW
présent dans les deux Triggers et qui ne peuvent pas être utilisés directement dans une
procédure stockée. IN représente une valeur en entrée dont il faut spécifier le nom et le
type*/
BEGIN
/* le corps du programme est assez similaire à celui des deux Triggers*/
DECLARE nb INT ;
SELECT COUNT(*) INTO nb /*le résultat de la requête est récupéré dans la variable nb*/
FROM Stagiaire
WHERE NumEnsStg= NumEnsProc
AND NumStageStg= NumStagProc;
/* nous avons remplacé les deux objets NEW.NumEnsF et NEW.NumStageF par les deux
paramètres NumEnsProc et NumStagProc qui sont envoyés à la procédure stockée par
le Trigger */
IF (nb=1) THEN /*le couple existe déjà dans la table Stagiaire, il ne faut pas réaliser
une opération INSERT ou une opération UPDATE dans la table Formateur*/
SIGNAL SQLSTATE "45000" SET MESSAGE_TEXT ="opération impossible" ;
/* Dans le cas contraire, ou nb=0, le couple n'est pas dans la table Stagiaire et
l'opération INSERT ou l'opération UPDATE dans la table Formateur se produira
normalement*/
END IF ;
END |
DELIMITER ;
```

Pour vérifier que la procédure stockée a été réellement créée, allez au volet « Procédures stockées » de la vue global de la base de données :



Les deux Triggers « **avant insertion Formateur** » et « **avant modification Formateur** » sont écrits comme suit :

```

DROP TRIGGER IF EXISTS Avant_insertion_Formateur ;
DELIMITER // -- le délimiteur du bloc d'instructions devient //.
CREATE TRIGGER Avant_insertion_Formateur
BEFORE INSERT -- avant toute insertion dans la table Formateur.
ON Formateur
FOR EACH ROW
BEGIN
CALL VérifFormStg (NEW.NumEnsF, NEW.NumStageF) ;
END // -- le TRIGGER se termine grâce au nouveau délimiteur.
DELIMITER ; -- on restaure l'ancien délimiteur.

DROP TRIGGER IF EXISTS Avant_Modification_Formateur ;
DELIMITER // -- le délimiteur du bloc d'instructions devient //.
CREATE TRIGGER Avant_Modification_Formateur
BEFORE UPDATE -- avant toute modification dans la table Formateur.
ON Formateur
FOR EACH ROW
BEGIN
CALL VérifFormStg (NEW.NumEnsF, NEW.NumStageF) ;
END // -- le TRIGGER se termine grâce au nouveau délimiteur.
DELIMITER ; -- on restaure l'ancien délimiteur.

```

Nous avons ainsi optimisé le code des deux triggers (ceci est faisable aussi pour les vérifications de contraintes lors des opérations d'insertion ou de modification sur la table Stagiaire).

La procédure stockée sera appelé depuis le Trigger, et dans le cas où la procédure stockée génère une erreur (déclenche une erreur), le Trigger va bloquer l'opération d'insertion (INSERT) ou de modification (UPDATE) (tous dépend du Trigger appelant de la procédure stockée).

Test :

```

INSERT INTO Stagiaire (NumEnsStg, NumStageStg) VALUES ('33', '6') ;
INSERT INTO Stagiaire (NumEnsStg, NumStageStg) VALUES ('33', '7') ;
INSERT INTO Stagiaire (NumEnsStg, NumStageStg) VALUES ('21', '7') ;

```

Tentative d'insertion dans « Formateur » :

```

INSERT INTO Formateur (NumEnsF, NumStageF) VALUES ('33', '4') ;

```

Cette insertion se passe bien puisque la contrainte est vérifiée.

```

INSERT INTO Formateur (NumEnsF, NumStageF) VALUES ('21', '6') ;

```

Cette insertion se passe bien puisque la contrainte est vérifiée.

```

INSERT INTO Formateur (NumEnsF, NumStageF) VALUES ('33', '6') ;

```

Cette insertion provoque une erreur car le couple (33,6) est déjà présent dans Stagiaire : l'insertion dans la table Formateur est bloqué par un message d'erreur provoqué par la procédure stockée.

```
UPDATE Formateur
SET NumStageF = '5'
WHERE NumEnsF = '33' AND NumStageF = '4';
```

Cette modification se passe bien puisque la contrainte est vérifiée.

```
UPDATE Formateur
SET NumStageF = '7'
WHERE NumEnsF = '21' AND NumStageF = '6';
```

Cette modification provoque une erreur car le couple (21,7) est déjà présent dans la table **Stagiaire** : la modification dans la table Formateur est bloqué par un message d'erreur provoqué par la procédure stockée.

Vous remarquez que le message d'erreur pour les deux Triggers est standard pour les deux commandes d'insertion (**INSERT**) et de modification (**UPDATE**) sur la table « Formateur » (puisque le message d'erreur se trouve dans la procédure stockée et la même procédure stockée est utilisée par les deux Triggers). Pour pouvoir personnaliser les messages d'erreurs, on peut toujours utiliser une fonction stockée à la place d'une procédure stockée (une fonction stockée ressemble à une procédure stockée sauf qu'elle retourne une valeur de sortie). C'est-à-dire au lieu de générer un message d'erreur dans la procédure stockée, la fonction stockée va renvoyer une valeur de sortie au Trigger qui va en fonction de cette valeur soit générer un message d'erreur personnalisé (à la place du message standard) soit permettre la commande à laquelle il est lié (commande d'insertion (**INSERT**) ou commande de modification (**UPDATE**)).

Code de la fonction stockée et des Triggers:

```
DELIMITER | -- le délimiteur du bloc d'instructions devient |.
CREATE FUNCTION VérifFormStgFUN (IN NumEnsFun INT, IN NumStagFun INT)
RETURNS INT
/* cette fonction comporte deux paramètres en entrée qui sont les deux objets NEW
présent dans les Triggers et qui ne peuvent pas être utilisés directement dans une
fonction stockée. La fonction stockée retourne une valeur entière*/
/* il existe un petit détail à ne pas oublier dans une fonction stockée et qui n'est pas
spécifiée dans une procédure stockée, c'est de préciser le type d'actions appliqués par la
fonction par rapport aux données. Si l'on veut exécuter des requêtes SQL de type
sélection il faut spécifier READ SQL DATA. Si cette instruction est oubliée, les requêtes
SQL SELECT ne seront pas exécutées dans la fonction stockée*/
READ SQL DATA
BEGIN
/* le corps de la fonction stockée est assez similaire à celui du Trigger*/
DECLARE nb INT ;
SELECT COUNT(*) INTO nb /* le résultat de la requête est récupéré dans la variable nb*/
FROM Stagiaire
WHERE NumEnsStg = NumEnsFun
AND NumStageStg= NumStagFun;
```

/* nous avons remplacé les deux objets NEW.NumEnsF et NEW.NumStageF par les deux paramètres NumEnsFun et NumStagFun qui sont envoyés à la fonction stockée par le Trigger */

RETURN nb; /*on ne génère pas d'erreur dans fonction stockée, on retourne seulement une valeur au Trigger et c'est au Trigger de tester cette valeur et de faire une action en fonction de cette valeur */

END |

DELIMITER ;

DROP TRIGGER IF EXISTS Avant_insertion_Formateur ;

DELIMITER // -- le délimiteur du bloc d'instructions devient //.

CREATE TRIGGER Avant_insertion_Formateur

BEFORE INSERT -- avant toute insertion dans la table Formateur.

ON Formateur

FOR EACH ROW

BEGIN

DECLARE valretour INT;

SET valretour := CALL VérifFormStgFUN (NEW.NumEnsF, NEW.NumStageF);

IF (valretour=1) THEN /*le couple existe déjà dans la table Stagiaire, il ne faut pas réaliser une opération INSERT dans la table Formateur*/

SIGNAL SQLSTATE "45000" SET MESSAGE_TEXT="opération d'insertion impossible" ;

/* Dans le cas contraire, ou valretour =0, le couple n'est pas dans la table Stagiaire et l'opération INSERT dans la table Formateur se produira après la fin du Trigger*/

END IF ;

END // -- le TRIGGER se termine grâce au nouveau délimiteur.

DELIMITER ; -- on restaure l'ancien délimiteur.

DROP TRIGGER IF EXISTS Avant_Modification_Formateur ;

DELIMITER // -- le délimiteur du bloc d'instructions devient //.

CREATE TRIGGER Avant_Modification_Formateur

BEFORE UPDATE -- avant toute modification dans la table Formateur.

ON Formateur

FOR EACH ROW

BEGIN

DECLARE valretour INT;

SET valretour := CALL VérifFormStgFUN (NEW.NumEnsF, NEW.NumStageF);

IF (valretour=1) THEN /*le couple existe déjà dans la table Stagiaire, il ne faut pas réaliser une opération UPDATE dans la table Formateur*/

SIGNAL SQLSTATE "45000" SET MESSAGE_TEXT="opération de modification impossible" ;

/* Dans le cas contraire, ou valretour =0, le couple n'est pas dans la table Stagiaire et l'opération UPDATE dans la table Formateur se produira après la fin du Trigger*/

END IF ;

END // -- le TRIGGER se termine grâce au nouveau délimiteur.

DELIMITER ; -- on restaure l'ancien délimiteur.

Test :

INSERT INTO **Stagiaire** (NumEnsStg, NumStageStg) VALUES ('18', '6') ;

INSERT INTO **Stagiaire** (NumEnsStg, NumStageStg) VALUES ('18', '7') ;

INSERT INTO **Stagiaire** (NumEnsStg, NumStageStg) VALUES ('19', '7') ;

Tentative d'insertion dans « Formateur » :

INSERT INTO **Formateur** (NumEnsF, NumStageF) VALUES ('18', '8') ;

Cette insertion se passe bien puisque la contrainte est vérifiée.

INSERT INTO **Formateur** (NumEnsF, NumStageF) VALUES ('19', '8') ;

Cette insertion se passe bien puisque la contrainte est vérifiée.

INSERT INTO **Formateur** (NumEnsF, NumStageF) VALUES ('19', '7') ;

Cette insertion provoque une erreur car le couple (19,7) est déjà présent dans **Stagiaire** : l'insertion est bloqué dans la table Formateur par un message d'erreur provoqué par la fonction stockée.

UPDATE **Formateur**

SET NumStageF = '5'

WHERE NumEnsF = '18' AND NumStageF = '8';

Cette modification se passe bien puisque la contrainte est vérifiée.

UPDATE **Formateur**

SET NumStageF = '7'

WHERE NumEnsF = '18' AND NumStageF = '5';

Cette modification provoque une erreur car le couple (18,7) est déjà présent dans la table **Stagiaire** : la modification dans la table Formateur est bloqué par un message d'erreur provoqué par la fonction stockée.

L'utilisation de fonctions stockées est également faisable pour la vérification de contraintes lors de l'insertion ou de la modification dans la table Stagiaire.