Cours : Triggers et événements

Thème : conception de déclencheurs et mise en place d'événements

Notions abordées :
- Le respect de contraintes d'intégrité avancées à l'aide de déclencheurs ;
- La conception de déclencheurs, alias triggers ;
- La conception de routines ou de tâches planifiées ;
- Pour aller plus loin : la programmation événementielle et le design pattern observer appliqué aux bases de données.

1. Introduction

1.1. SQL et limites

Le langage SQL usuel permet de nombreuses opérations sur les bases de données mais ne suffit malgré tout pas à implémenter toutes les règles de gestion inhérentes à celles-ci. En effet, il est par exemple impossible d'assurer l'intégrité des données dès lors que le modèle de données nécessite la mise en œuvre de contraintes d'intégrité avancées.

En particulier, les contraintes d'entités (l'héritage) ou encore les contraintes d'associations (inclusion, exclusion, etc.) ne peuvent être mises en œuvre au moyen de simples clefs étrangères ou encore de contraintes de domaine (CHECK). Pareillement, l'historisation ou encore la stabilité constituent des contraintes tout à fait modélisables en Merise 2 mais dont la mise en œuvre s'avère inenvisageable avec du SQL habituel. On ne pourra pas non plus envisager la mise en place de champs calculés, pourtant si pratiques aux fins d'optimiser les requêtes statistiques en outre.

Ainsi le langage SQL a-t 'il fait l'objet d'enrichissements successifs de sorte que de nouvelles notions sont apparues de manière à pouvoir solutionner des problématiques telles que celles évoquées ci-avants. Ces notions font même désormais partie de la norme SQL : trigger, événements, procédures et fonctions stockées.

C'est dans ce contexte que l'on étudie ci-après les notions d'événements et de triggers. Procédures et fonctions stockées font l'objet d'un cours venant compléter celui-ci.

Au demeurant, le présent cours se veut être une introduction. Il ne s'agit pas d'explorer toutes les facettes du SQL, chose qui ne présente d'ailleurs ni un grand intérêt théorique, ni un grand intérêt pratique.

Pour ce faire, nous étudierons ces notions sous MySQL, facile d'utilisation et surtout gratuit. On notera cependant que MySQL, très largement en retard il y a encore quelques années, est à ce jour l'un des SGBDR à respecter le mieux la norme SQL.

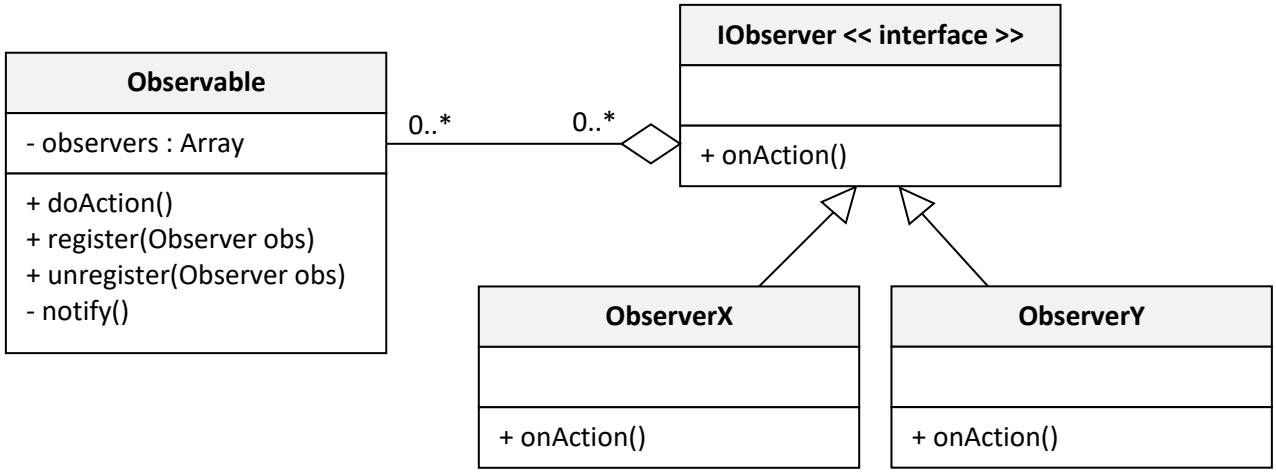
1.2. Evénement et triggers

Avertissement ! Cette partie est à l'attention des curieux et n'est pas forcément nécessaire à la compréhension de ce qui suit.

Tâchons, à titre accessoire, de comprendre la notion d'événement en programmation orientée objets (POO). En POO, il existe ce que l'on a coutume d'appeler les *designs patterns*, soit en français les patrons de conceptions. Un patron de conception admet en règle générale une représentation sous forme de diagramme de classes car un patron de conception est un modèle type, un procédé type, permettant de répondre à une problématique classique. Or, une problématique classique est celle de la programmation événementielle.

- Objectif : l'on souhaite que zéro à plusieurs algorithmes puissent être exécutés au moment où une action ou une série de traitements se produisent. On qualifie volontiers ce moment d'événement. Ainsi, l'on désire que des algorithmes personnalisés puissent dynamiquement compléter une série de traitements en cours. En d'autres termes, il s'agit de pouvoir exécuter un code, une fonction ou encore une méthode d'objet quelconque au moment ou un événement est émis.
- Idée : l'idée, c'est d'émettre un événement au moment où l'on souhaite que des algorithmes puissent être exécutés. Il faut encore offrir la possibilité à un algorithme de s'enregistrer et/ou de se désenregistrer en tant qu'observateur de l'événement de sorte que, lorsque l'événement est levé, les algorithmes enregistrés soient parcourus et exécutés par celui-là même qui a levé l'évènement.
- Solution : le *design pattern* observer. On décide qu'une classe est l'observable ou observé (soit *observable* ou *observed* en anglais) et une autre l'observateur (soit *observer* en anglais). Les observateurs vont s'inscrire auprès de l'observable pour pouvoir l'observer. Et lorsque l'observable émet un événement, il va notifier ses observateurs. Par notifier, on entend que l'observable va parcourir ses observateurs et appeler une méthode commune à ces derniers. Ça y est ! On a créé un système événementiel.

Le *pattern observer* se traduit finalement par le diagramme de classes générique suivant :



Dans le cas des bases données, le problème se traduit de la manière suivante :

- Objectif : on souhaite pouvoir exécuter un algorithme quelconque, qu'on appellera **trigger**, avant ou

après une insertion, une modification ou une suppression sur une table.

- Solution : on considère alors que la table est l'observable et le trigger l'observé. Imaginons que Table et Trigger soient deux classes. Une implémentation est proposée en Annexe 1 : **implémentation des triggers, la classe Trigger** et Annexe 2 : **implémentation des triggers, la classe Table**.

2. Déclencheurs (alias triggers)

2.1. Principe

2.1.1. Le principe général

Un trigger, soit en français un déclencheur, est un algorithme exécuté à l'occasion d'un événement se produisant sur une table d'une base de données. Les événements sont typiquement les suivants :

- **BEFORE INSERT** : pour exécuter un trigger avant l'insertion d'une nouvelle ligne au sein d'une table. La nouvelle ligne n'est donc pas encore dans la table au moment de l'exécution du trigger.
- **AFTER INSERT** : pour exécuter un trigger après l'insertion d'une nouvelle ligne au sein d'une table. La nouvelle ligne existe donc effectivement dans la table au moment de l'exécution du trigger.
- **BEFORE UPDATE** : pour exécuter un trigger avant la modification d'une ligne existant dans une table. L'ancienne ligne n'a pas encore été affectée par la modification au moment où le trigger est exécuté.
- **AFTER UPDATE** : pour exécuter un trigger après la modification d'une ligne qui existait dans une table. L'ancienne ligne a par conséquent déjà été affectée par le modification au moment où le trigger est exécuté.
- **BEFORE DELETE** : pour exécuter un trigger avant la suppression d'une ligne. L'ancienne ligne existe par conséquent encore au moment où le trigger est exécuté.
- **AFTER DELETE** : pour exécuter un trigger après la suppression d'une ligne. L'ancienne ligne n'existe par conséquent plus au moment où le trigger est exécuté.

Un trigger peut, au besoin être attaché à plusieurs événements (exemple : **BEFORE INSERT, UPDATE**).

Quoiqu'il en soit, un déclencheur est un réel algorithme. En cela, dès lors que l'on s'attaque à l'implémentation de trigger, le SQL devient un réel langage de programmation.

2.1.2. Les mots-clefs NEW et OLD

Que représente l'opérateur NEW ? Dans le code d'un trigger, le mot-clef NEW permet d'utiliser la ligne en cours d'insertion ou de modification. Syntaxe : `NEW.nomChamp`.

Que représente l'opérateur OLD ? Dans le code d'un trigger, le mot-clef OLD permet d'utiliser la ligne qui a été ou va être modifiée ou supprimée. Syntaxe : `OLD.nomChamp`.

Quand peut-on les utiliser ?

Evénement (BEFORE ou AFTER)	**NEW**	**OLD**
INSERT	OUI	NON
UPDATE	OUI	OUI
DELETE	NON	OUI

Explication :

- À l'insertion, l'on insère une nouvelle ligne. Il n'y a pas d'ancienne ligne (OLD) mais bien une nouvelle (NEW). De ce fait, l'opérateur OLD est inutilisable.
- À la mise à jour, l'on vient modifier une ligne préexistante. Il y a donc une ancienne ligne (OLD) et une

nouvelle (NEW), à savoir la même mais modifiée. NEW et OLD sont dès lors utilisables.

- À la suppression, l'on souhaite retirer une ligne préexistante. Il n'est pas question de nouvelle ligne (NEW) mais l'ancienne va être ou a été supprimée (OLD). On peut par conséquent utiliser OLD mais on ne peut utiliser NEW.

2.1.3. La notion de transaction

L'exécution d'un trigger est encapsulée dans ce qu'on appelle une transaction. Une transaction, c'est un ensemble de traitements qui doivent s'exécuter en bloc. Si l'un des traitements échoue, toute la transaction échoue et tous les traitements qui ont été exécutés sont annulés.

L'on peut valider une transaction. C'est ce qu'on appelle un COMMIT. Au contraire, l'on peut faire échouer une transaction. C'est ce qu'on qualifie de ROLLBACK.

En MySQL, conformément à la norme, on fait échouer une transaction en levant une erreur grâce à l'instruction SIGNAL. Exemple :

```
SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'An error occurred', MYSQL_ERRNO = 1001;
```

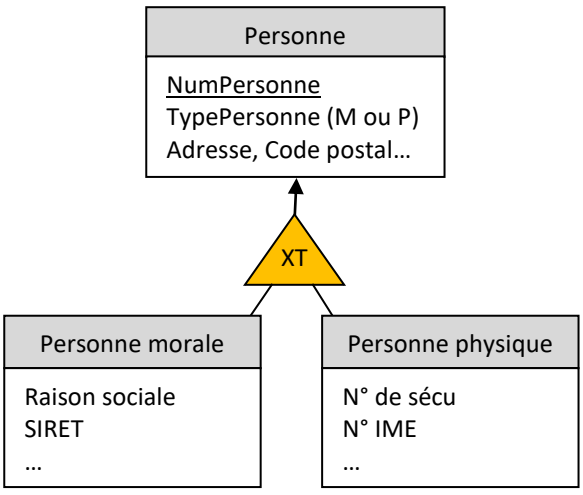
Quel est l'intérêt de faire échouer une transaction ? Très simplement, la logique veut que, lors de l'exécution d'une requête, si une contrainte d'intégrité n'est pas vérifiée, la transaction en question échoue.

2.1.4. L'intérêt

Si nous pouvons exécuter un programme lorsqu'un événement se produit, nous pouvons désormais, entre autres choses, vérifier des contraintes d'intégrité avancées. Et c'est ce que nous allons tâcher de montrer au travers de quelques exemples.

2.2. Conception

2.2.1. Vérifier une contrainte d'entités



L'on ne cesse de parler d'héritage ou de spécialisation. Pour autant, nous n'étions pas capables de nous assurer qu'une telle contrainte soit vérifiée.

Quels problèmes ?

- La contrainte XT (partition) impose qu'une personne soit forcément morale ou physique ;
- Rien n'empêche pourtant qu'on ait une personne ne figurant ni parmi les personnes morales ni parmi les personnes physiques.
- Rien n'empêche non plus qu'on ait une personne figurant tout à la fois parmi les personnes morales et parmi les personnes physiques.

Quelle solution possible ?

- Si l'on insère une personne, en fonction de son type, on insère une personne morale ou physique par défaut, ce qui impose que les champs propres à la personne morale ou physique puissent être NULL ;
- On empêche l'insertion d'une personne morale ou physique à partir du moment où la personne est déjà associée à une personne morale ou physique.

Quelle implémentation ?

Les traitements ci-avant évoqués se traduisent par l'ajout des deux triggers suivants.

1er trigger *:* création d'une personne morale ou physique par défaut

```
DROP TRIGGER IF EXISTS create_default_personne ;
DELIMITER $$
CREATE TRIGGER create_default_personne AFTER INSERT ON Personne FOR EACH ROW
BEGIN
    -- insertion d'une personne morale
    IF NEW.TypePersonne = 'M' THEN
        INSERT INTO PersonneMorale(NumPersonne) VALUES (NEW.NumPersonne) ;
    -- sinon insertion d'une personne physique
    ELSE
        INSERT INTO PersonnePhysique(NumPersonne) VALUES (NEW.NumPersonne) ;
    END IF ;
END $$
DELIMITER ;
```

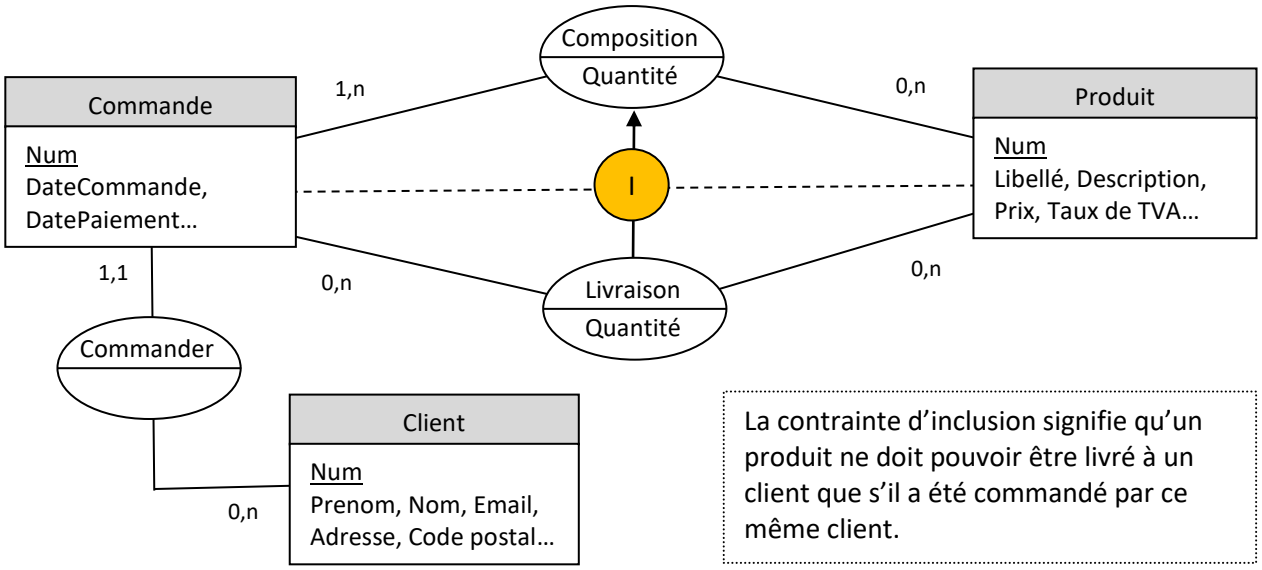
2ème trigger : on empêche l'insertion d'une personne physique ou morale lorsque la personne correspondante en a déjà une associée. *N.B. : ce trigger doit encore être appliqué à la table PersonnePhysique.*

```
DROP TRIGGER IF EXISTS prevent_moral_insert ;
DELIMITER $$
CREATE TRIGGER create_default_personne BEFORE INSERT ON PersonneMorale FOR EACH ROW
BEGIN
    -- on récupère l'information : combien de personnes morales ou physiques sont associées
    -- à la personne morale insérée ?
    SELECT COUNT(*) INTO @nombre
    FROM Personne P
    LEFT JOIN PersonneMorale PM ON P.NumPersonne = PM.NumPersonne
    LEFT JOIN PersonnePhysique PP ON P.NumPersonne = PP.NumPersonne ;
    WHERE P.NumPersonne = NEW.NumPersonne ;
    -- si la personne est déjà associée à quelque chose, alors rollback
    IF @nombre > 0 THEN
        SIGNAL sqlstate '45000' SET message_text = 'Personne déjà affectée' ;
    END IF ;
END $$
DELIMITER ;
```

2.2.2. Vérifier une contrainte d'associations

Important ! Toute contrainte d'associations nécessite des contrôles qui peuvent être mis en œuvre de manière applicative ou au moyen de triggers.

1er exemple : trigger pour vérifier une contrainte d'inclusion (*gestion de commandes*).



La contrainte d'inclusion signifie qu'un produit ne doit pouvoir être livré à un client que s'il a été commandé par ce même client.

Le sens de la contrainte d'inclusion n'est pas même suffisant pour décrire la contrainte d'intégrité à respecter. En effet, il convient non seulement, pour une commande donnée, qu'un client ne se voit livrer que des produits qu'il a commandés, mais encore que la quantité livrée de chaque produit commandé ne n'excède pas celle commandée.

Quels problèmes ?

- Rien ne nous assure que les livraisons portent sur des produits commandés ;
- Rien ne nous assure que les quantités livrées ne soient pas supérieures à celles commandées.

Quelle solution possible ?

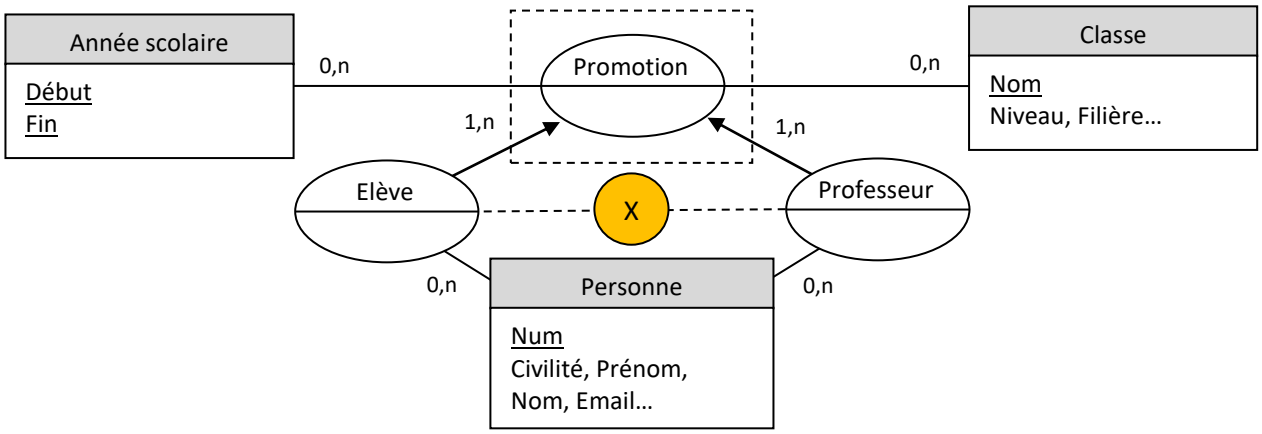
- A l'insertion comme à la modification d'une livraison, on vérifie que le produit inséré ait bien été commandé par le client ;
- Toujours à l'insertion et à la modification d'une livraison, on vérifie que la quantité totale livrée du produit qu'on souhaite livrer soit au plus égal à la quantité commandée concernant ce même produit.

Quelle implémentation ?

```
DROP TRIGGER IF EXISTS check_livraison ;
DELIMITER $$
CREATE TRIGGER check_livraison AFTER INSERT, UPDATE ON Livraison FOR EACH ROW
BEGIN
```

```
    DECLARE QTE INT
    -- quantité totale commandée du produit livre ou à livrer
    SELECT SUM(Composition.Quantite) INTO QTE
    FROM Composition C1
    WHERE C1.NumCommande = NEW.NumCommande
    AND C1.NumProduit = NEW.NumProduit ;
    -- erreur si produit non commandé
    IF QTE = 0 THEN
        SIGNAL sqlstate '45000' SET message_text = 'Livraison d'un produit non commandé' ;
    END IF ;
    -- erreur si quantité livrée supérieure strictement à quantité commandée
    IF NEW.Quantité > QTE THEN
        SIGNAL sqlstate '45000' SET message_text = 'Livraison en quantités excédentaires' ;
    END IF ;
END $$
DELIMITER ;
```

2ème exemple : trigger pour vérifier une contrainte d'exclusion (*gestion de classes*).



Le couple (Année scolaire, Classe) constitue une promotion. C'est un agrégat. Une telle promotion dispose de plusieurs professeurs et élèves. En l'espèce, la contrainte d'exclusion figurant ci-dessus traduit une règle de gestion des plus simples : pour une classe et une année scolaire données, l'on ne peut être à la fois professeur et élève.

Quels problèmes ? Rien ne nous assure qu'un professeur d'une promotion ne participe pas également à la relation élève et réciproquement.

Quelle solution possible ? Lorsque l'on ajoute un professeur, il convient qu'il ne figure pas parmi les élèves. Inversement, lorsque l'on ajoute un élève, il convient qu'il ne soit pas professeur. S'il participe aux deux relations il importe d'interdire l'ajout, à savoir l'insertion, voire la modification également.

Quelle implémentation ?

1er trigger *:* aucun professeur ne peut être élève.

```
DROP TRIGGER IF EXISTS check_exclusion_prof ;
DELIMITER $$
CREATE TRIGGER check_exclusion_prof BEFORE INSERT, UPDATE ON Professeur FOR EACH ROW
BEGIN
    DECLARE QTE INT
    -- si le professeur à ajouter/modifier existe en tant qu'élève, c'est mal !
    IF ( SELECT COUNT(*) FROM Eleve E WHERE E.NumPersonne = NEW.NumPersonne ) > 0 THEN
        SIGNAL sqlstate '45000'
            SET message_text = 'Un professeur d'une promo ne saurait en être un élève' ;
    END IF ;
END $$
DELIMITER ;
```

2ème trigger *:* aucun élève ne peut être professeur.

L'on crée le trigger check_exclusion_mome dont la principale différence avec le précédent trigger est la requête SQL, laquelle devient :

```
SELECT COUNT(*) FROM Professeur P WHERE P.NumPersonne = NEW.NumPersonne
```

2.2.3. Utiliser des champs calculés

Bien que les champs calculés constituent une forme de redondance de données, il est parfois commode d'en utiliser afin d'optimiser les performances de certaines requêtes. En effet, l'usage de champs calculés peut servir aux fins par exemple d'éviter des jointures.

1er Exemple : dans le MCD de *gestion de commandes* précédemment représenté, afin de faciliter les calculs concernant les ventes réalisées, on souhaite :

- Ajouter les attributs MontantHT, MontantTVA et MontantTTC à l'entité Composition. Ces propriétés doivent être calculées.
- Ajouter les attribut MontantHT, MontantTVA et MontantTTC à l'entité Commande. Ces propriétés doivent être calculées automatiquement.

Quelle solution possible ?

- A l'insertion ainsi qu'à la modification de la composition d'une commande, calculer ou recalculer les champs MontantHT, MontantTVA et MontantTTC ;
- A l'insertion ainsi qu'à la modification et à la suppression de la composition d'une commande, calculer ou recalculer les champs MontantHT, MontantTVA et MontantTTC de la commande montant de la commande.

Quelle implémentation ?

1er trigger : calcule ou recalcule les champs MontantHT, MontantTVA et MontantTTC des composants d'une commande, c'est-à-dire des lignes de commande.

```
DROP TRIGGER IF EXISTS calculate_composition ;
DELIMITER $$
CREATE TRIGGER calculate _composition BEFORE INSERT, UPDATE ON Livraison FOR EACH ROW
BEGIN
    -- récupération du prix et du taux de TVA applicable au produit
    SELECT Prix, TauxTVA INTO @prix, @taux
    FROM Produit
    WHERE Num = NEW.Num ;
    -- (re)calcule des totaux
    SET NEW.MontantHT = @prix * NEW.Quantite ;
    SET NEW.MontantTVA = NEW.MontantHT * @taux ;
    SET NEW.MontantTTC = NEW.MontantHT + NEW.MontantTVA ;
END $$
DELIMITER ;
```

2ème trigger : calcule ou recalcule les champs MontantHT, MontantTVA et MontantTTC d'une commande dès lors que sa composition est modifiée.

```
DROP TRIGGER IF EXISTS calculate_commande ;
DELIMITER $$
CREATE TRIGGER calculate_commande AFTER INSERT, UPDATE ON Livraison FOR EACH ROW
BEGIN
    -- calcule le montant HT et le montant de TVA de la commande à partir de sa composition
    SELECT SUM(MontantHT), SUM(MontantTVA) INTO @montantHT, @montantTVA
    FROM Composition
    WHERE NumCommande = NEW.NumCommande ;
    -- (re)calcule des totaux
    SET NEW.MontantHT = @montantHT ;
    SET NEW.MontantTVA = @montantTVA ;
    SET NEW.MontantTTC = @montantHT + @montantTVA ;
END $$
DELIMITER ;
```

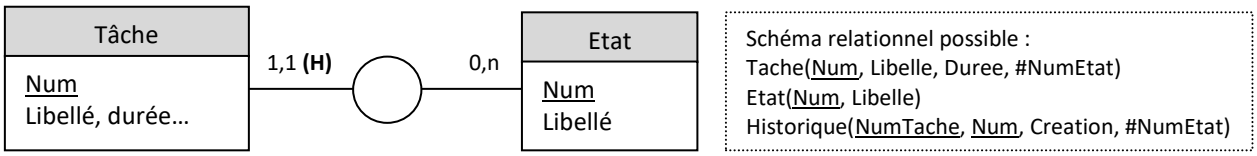
On place un trigger similaire AFTER DELETE, avec pour principale différence :

```
    WHERE NumCommande = OLD.NumCommande ;
```

2.2.4. Historiser des données

L'historisation telles que modélisée en Merise 2 constitue un autre cas d'espèce où l'on peut avoir recours aux triggers. On rappelle que l'historisation consiste à conserver, au travers du temps, l'historique des différentes modifications effectuées sur tout ou partie des champs d'une entité.

Exemple : historisation de l'état d'une tâche.



Quelle solution possible ? A l'insertion ainsi qu'à la modification d'une tâche, on historise son état, à savoir qu'on crée une ligne d'historique.

Quelle implémentation ?

```
DROP TRIGGER IF EXISTS historicize_tache ;
DELIMITER $$
CREATE TRIGGER historicize_tache BEFORE INSERT, UPDATE ON Tache FOR EACH ROW
BEGIN
    -- crée une nouvelle ligne d'historique
    INSERT INTO Historique (NumTache, Num, Creation, NumEtat)
    VALUES (
        NEW.Num,
        SELECT COUNT(*) + 1 FROM Historique WHERE NumTache = NEW.NUM,
        NOW(),
        NEW.NumEtat
    ) ;
END $$
DELIMITER ;
```

Les fonctions de date MySQL :

- La fonction NOW() retourne la date et l'heure courante au format : AAAA-MM-DD hh:mm:ss.
- La fonction CURDATE() retourne la date courante au format : AAAA-MM-DD.
- La fonction CURTIME() retourne l'heure courante au format : hh:mm:ss.

2.2.5. Conclusion

On retiendra essentiellement que, en pratique, la logique métier d'un logiciel implique en général la conception ou l'évolution d'une base de données dont l'intégrité ne peut être assurée qu'au prix de nombreux contrôles. Il importe d'être capable d'identifier les contrôles à mettre en œuvre et d'établir les algorithmes nécessaires à leur mise en œuvre, qu'il s'agisse de les concevoir sous forme applicative ou sous forme de triggers.

2.3. Limites

2.3.1. Avantages

Les triggers permettent de mettre en œuvre des contrôles d'intégrité avancées. A l'inverse des contrôles applicatifs, lesquels nécessitent des va-et-vient entre le logiciel et le SGBD, les triggers permettent un accès direct à la base de données. Plus encore, l'exécution du code d'un trigger étant réalisée au sein même du SGBD, l'algorithme développé peut profiter des optimisations effectuées par le SGBD (tables

d'index, fonctions de calcul optimisées, etc.). En somme, l'avantage du recours aux triggers réside avant tout dans les performances qu'ils peuvent procurer.

2.3.2. Inconvénients

Quoique des normes SQL existent et que les implémentations du langage SQL tendent à s'harmoniser ces dernières années, la syntaxe utilisée demeure relativement propre au SGBDR utilisé. Par exemple, MS SQL Server dispose de sa propre implémentation, appelée Transact-SQL et souvent abrégée TSQL. De même, Oracle dispose d'une implémentation appelée PL-SQL.

Ainsi, la première limite des triggers réside dans le fait que le SQL avancé, celui que nous étudions dans ce cours, ne soit pas interopérable. De plus, la syntaxe SQL employée peut sembler relativement lourde en comparaison de celles des langages les plus courants (PHP, Python, Java/JEE, etc.). Finalement, ces derniers langages offrent une syntaxe souvent plus étoffée que celle du SQL.

2.3.3. Alternative

Un trigger n'est finalement qu'un simple programme. Et il est finalement très fréquent de vérifier les contraintes d'intégrité de façon logicielle. On parle de contrôle logiciel ou de contrôle applicatif. Autrement dit, c'est souvent le logiciel ou les logiciels interagissant avec une base de données qui procèderont à la vérification des règles de gestion et des contraintes d'intégrité en particulier. On considérera communément que ces vérifications font partie de la logique métier.

Ce choix offre souvent l'avantage de la portabilité, de l'interopérabilité. Effectivement, une migration de base de données ou une duplication de la base de données vers un autre SGBD sera sans effet sur la vérification des contraintes. De surcroît, les langages de programmations actuels disposent en général d'une syntaxe plus aboutie et plus facile d'utilisation que celle proposée par la norme SQL. Plus encore, ces derniers sont souvent accompagnés de *frameworks* de type ORM (*Object Relational Mapping*) permettant de vérifier de manière élégante des contraintes d'intégrités avancées.

Malgré tout, la programmation au sein des SGBD n'est pas en soi un phénomène isolé. Après tout, de nos jours encore, il existe en outre des développeurs et experts Oracle.

3. Tâches planifiées

3.1. Principe

Certains SGBD, dont MySQL, disposent d'un planificateur de tâches (*scheduler*) et permettent ainsi de mettre en place des tâches planifiées, qu'on peut encore qualifier de routines. En matière de SGBD, le terme approprié semble être celui « d'événements » (*event*).

Vous le savez sans doute, il est possible de planifier des tâches sur un ordinateur. Pour ce faire :
- Windows met par exemple à disposition l'utilitaire shtask.
- Unix met quant à lui à disposition l'utilitaire chrontab.

Dans tous les cas, les tâches planifiées sont soit :
- Des traitements exécutés en différé (*timer*).
- Des traitements exécutés à intervalle régulier (*routine*).

L'on conviendra qu'il est souvent plus commode d'utiliser shtask ou encore chrontab. Cependant, un SGBD tel que MySQL permet de mettre en place de telles tâches planifiées et routines.

3.2. Intérêt

Dans le cadre d'une base de données, certaines règles de gestion peuvent nécessiter la mise en place de traitements ponctuels ou récurrents. On ne voudra pour le montrer que l'exemple typique de l'archivage. En effet, pour des raisons de performances, on peut souhaiter procéder à l'archivage de données à intervalle régulier de manière à ne pas surcharger la base de données. Bref, on peut être amené à vouloir effectuer des tâches en différer ou à intervalles réguliers. Et certains SGBDR le permettent.

3.3. Conception

3.3.1. Créer et supprimer une tâche différée

```
DROP EVENT IF EXISTS ma_tache_differee ;
DELIMITER $$
CREATE EVENT ma_tache_differee ON SCHEDULE
    AT CURRENT_TIMESTAMP + INTERVAL 1 DAY + INTERVAL 12 HOUR
DO BEGIN
    -- le code de la tâche
END $$
DELIMITER ;
```

- Dans l'exemple ci-dessus, l'on crée une tâche qui s'exécutera dans un jour et demi à compter de la création de la tâche.
- Bien entendu, il est possible d'utiliser d'autres unités de temps : YEAR, QUARTER, MONTH, WEEK, MINUTE, SECOND.
- CURRENT_TIMESTAMP représente la date et l'heure courante. Rien n'empêche de préciser un quelconque autre *timestamp*, par exemple : '2052-04-27 23:59:59'.

3.3.2. Créer et supprimer une tâche récurrente

Aucune différence majeure lorsque l'on crée une routine. Il faut préciser les éléments suivants :

- L'intervalle de temps : fréquence d'exécution de la tâche.
- Le début (optionnel) : date et heure de la première exécution de la tâche. Si le début est omis, la première exécution de la tâche, c'est maintenant !
- La fin (optionnel) : date et heure à compter desquels la routine cessera d'être exécutée.

```
DROP EVENT IF EXISTS ma_tache_recurrente ;
DELIMITER $$
CREATE EVENT ma_tache_recurrente ON SCHEDULE
    EVERY 1 DAY + 12 HOUR
    STARTS CURRENT_TIMESTAMP + INTERVAL 1 WEEK
    ENDS CURRENT_TIMESTAMP + INTERVAL 6 MONTH
DO BEGIN
    -- le code de la routine
END $$
DELIMITER ;
```

Dans l'exemple ci-dessous, l'on crée une tâche qui sera exécutée tous les un jour et demi (EVERY), à compter de la semaine prochaine (STARTS) et ce jusqu'à dans six mois (ENDS).

3.3.3. Activer et désactiver une tâche planifiée

On notera qu'une tâche planifiée MySQL n'est pas active par défaut. Une tâche n'est exécutée que si elle est active.

1er cas : activer une tâche dès sa création.

```
[…] CREATE EVENT ma_tache ON SCHEDULE ENABLE […]
```

2ème cas : activer une tâche existante.

```
ALTER EVENT ma_tache ENABLE ;
```

3ème cas : désactiver une tâche existante.

```
ALTER EVENT ma_tache DISABLE ;
```

3.4. Limites

La mise en place de tâches planifiées pose le même problème que celui des triggers : la portabilité. Et il est souvent plus simple de mettre en place une tâche planifiée au moyen de *chrontab* ou de *shtask*. En

effet, grâce à ces utilitaires, l'on pourra par exemple exécuter des routines avancées. De fait, ces routines pourront certes être des fichiers de commandes, c'est-à-dire des *batchs* .bat (MSDOS) ou .sh (UNIX), mais encore des programmes avancés rédigées en C++, en Java ou encore en PHP. Or, ces langages offriront de nombreux avantages en matière de maintenabilité, d'évolutivité ou encore de réutilisabilité.

Annexe 1 : implémentation des triggers, la classe Trigger

```
Class Trigger{
	// Code du trigger
	private $code ;
	// Evénement sur lequel le trigger doit être enregistrer
	private $event ;
	/**
	 * Permet de créer un trigger (constructeur).
	 * @param String $on
	 *	Nom de la table sur laquelle doit être enregistré le trigger.
	 * @param String $event
	 *	Evenement sur lequel doit être enregistré le trigger, soit :
	 *	- "BEFORE INSERT" ou "AFTER INSERT"
	 *	- "BEFORE UPDATE" ou "AFTER UPDATE"
	 *	- "BEFORE DELETE" ou "AFTER DELETE"
	 **/
	public function __construct($on, $event , $code){
		$this->event = $event;
		$this->code = $code;
		// Récupération de la table auprès de laquelle le trigger doit être enregistré
		$table = Table::table($on) ;
		// Enregistrement du trigger en tant qu'observateur de la table $table
		$table->register($this) ;
	}
	/**
	 * Retourne l'événement sur lequel le trigger doit être enregistré.
	 * @return String
	 **/
	public function event(){
		return $this->event ;
	}
	/**
	 * Interprète le code du trigger
	 * @Exception
	 *		L'exécution du trigger peut lever une exception.
	 **/
	public function execute(){
		…
	}
}
```

Annexe 2 : implémentation des triggers, la classe Table

```
Class Table{
	// Toutes les tables enregistrées.
	private static $tables = array() ;
	// Tous les triggers enregistrés en tant qu'observateurs de la table courante
	private $triggers = array() ;
	// Nom de la table courante
	private $name ;
	// Lignes de la table courante
	private $rows = array() ;
	/**
	 * Permet de créer une nouvelle table
	 **/
	public function __construct($name){
		$this->name = $name ;
		self::$tables[$name] = $this ;
	}
	/**
	 * Retourne, à partir de son nom, l'une des tables enregistrées
	 * @param String $name
	 *	Nom de la table à retourner
	 **/
	public static function table($trigger){
		return self::$tables[$name] ;
	}
	/**
	 * Permet d'enregistrer un trigger
	 * @param Trigger $trigger
	 *	Le trigger à enregistrer en tant qu'observateur de la table courante
	 **/
	public function register($trigger){
		$this->triggers[$trigger->event()] = $trigger ;
	}
	/**
	 *  Exécute les triggers enregistrés sur l'événement passé en paramètre.
	 * @param String $event
	 * @param Array $old
	 *	Ligne avant opération
	 * @param Array $new
	 *	Ligne après opération
	 **/
	private notify($event, $old, $new){
		$triggers = $this->triggers[$event] ;
		foreach($triggers as $trigger){
			$trigger->execute() ;
```

```
            }
        }
        /**
         * Insertion d'une ligne dans la table courante.
         * @param Array $new
         *     Ligne à insérer
         **/
        public function insert($new){
            $this->notify("BEFORE INSERT", null, $new) ;
            …
            try{
                $this->notify("AFTER INSERT", null, $new) ;
            }catch(Exception $e){
                $this->delete($new) ;
            }
        }
        /**
         * Suppression d'une ligne de la table courante.
         * @param Array $old
         *     Ligne à supprimer
         **/
        public function delete($old){
            $this->notify("BEFORE DELETE", $old, null) ;
            …
            try{
                $this->notify("AFTER DELETE", $new, null) ;
            }catch(Exception $e){
                $this->insert($old) ;
            }
        }
        /**
         * Mise à jour d'une ligne dans la table courante
         * @param Array $old
         *     Ligne avant mise à jour
         * @param Array $new
         *     Ligne après mise à jour
         **/
        public function update($old, $new){
            $this->notify("BEFORE UPDATE", $old, $new) ;
            …
            try{
                $this->notify("AFTER DELETE", $old, $new) ;
            }catch(Exception $e){
                $this->update($new, $old) ;
            }
        }
    }
```