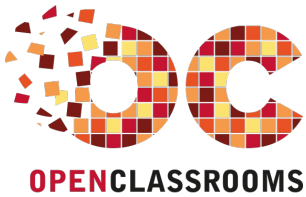
Triggers et vues matérialisées (MySQL)

Par Lord Casque Noir



www.openclassrooms.com

Dernière mise à jour le 6/01/2012

Sommaire

Triggers et vues matérialisées (MySQL)

Par  Lord Casque Noir

Mise à jour : 06/01/2012
Difficulté : Intermédiaire Durée d'étude : 2 heures

Ce tutoriel a pour but de vous familiariser avec les différents concepts suivants :

- les triggers ;
- les vues ;
- le maintien à jour de tables via des *triggers* ;
- l'utilisation de ces tables comme un cache pour accélérer grandement certaines requêtes.

Nous utiliserons MySQL 5 ; bien sûr, tout cela fonctionne aussi sous PostgreSQL à condition de changer un peu la syntaxe des exemples.

Prérequis

- Jointures.
- Relations.
- Transactions et différences MyISAM/InnoDB.
- Savoir ce qu'est une procédure stockée.

Sommaire du tutoriel :

Un exemple simple

Supposons que nous ayons une équipe de vendeurs, que nous rentrons dans la base de données sous la forme suivante :

Code : SQL

```
CREATE TABLE vendeurs (
  vd_id    INTEGER PRIMARY KEY AUTO_INCREMENT,
  vd_name  TEXT NOT NULL
) ENGINE=InnoDB;
```

Le patron veut que chaque vendeur entre dans la base de données, chaque jour, le total de ses recettes. Nous créons donc une table pour stocker ces informations :

Code : SQL

```
CREATE TABLE recettes_vendeurs (
  vd_id        INTEGER NOT NULL REFERENCES vendeurs( vd_id ) ON
DELETE RESTRICT,
  rc_date      DATE NOT NULL,
  rc_montant   NUMERIC( 12, 2 ),
  PRIMARY KEY( vd_id, rc_date ),
  KEY( rc_date, vd_id )
) ENGINE=InnoDB;
```

Quelques explications s'imposent.

Nous créons une **relation** entre la table « recettes_vendeurs » et la table « vendeurs » par « vd_id REFERENCES vendeurs(vd_id) ». Chaque ligne de « recettes_vendeurs » doit donc contenir l'*id* d'un vendeur existant. Comme nous n'avons par défini de contrainte d'unicité sur « recettes_vendeurs.vd_id », chaque vendeur peut avoir plusieurs lignes lui faisant référence dans « recettes_vendeurs ».

« ON DELETE RESTRICT » empêche la suppression d'un vendeur si des lignes de « recettes_vendeurs » lui font référence. En effet ici, on souhaite garder l'historique des recettes en comptabilité. Pour gérer le *turnover*, on ajoutera par exemple dans la table « vendeurs » des colonnes permettant de savoir si le vendeur est encore employé par la société, sa date d'embauche et éventuellement de départ, etc.

Pour d'autres applications où l'on souhaite supprimer automatiquement les lignes faisant référence au parent lorsque celui-ci est supprimé, on utilisera « ON DELETE CASCADE ».

Pour plus d'explications sur les relations : voir ce tutoriel.

La clé primaire naturelle de la table est (vd_id, rc_date) : elle est unique, une ligne par date et par vendeur. On aurait tout aussi bien pu utiliser (rc_date, vd_id). Il n'est pas nécessaire de créer une autre clé primaire. On crée aussi un index sur la date.

Pour la suite, nous allons remplir les tables avec ce petit script PHP très simple :

Code : PHP

```
<?php

// On place la connexion à MySQL dans un include pour éviter de
// diffuser par inadvertance des mots de passe sur le Net
// par copier-coller, ou dans Subversion/Git...
require_once "dbconn.php";

// Si les triggers sont actifs lors du remplissage de la table et
que le
// but est de les tester, mieux vaut réduire ces valeurs.
$max_vendeurs = 100;
$max_jours = 5000;

function query( $sql )
{
 $t = microtime( true );
 $q = mysql_query( $sql );
 if( $q  )
 {
  printf( "<p>%.02f ms : %s</p>", (microtime(true) - $t)*1000, $sql
);
  return $q;
 }
 echo "<pre>";
 var_dump( debug_backtrace() );
```

```
 echo "\n";
 exit( mysql_error() );
}

query( "BEGIN" );
query( "TRUNCATE recettes_vendeurs" );
query( "TRUNCATE vendeurs" );
query( "TRUNCATE recettes_jour_mat" );
query( "TRUNCATE recettes_mois_mat" );
query( "TRUNCATE recettes_vendeur_mois_mat" );

$values = array();
for( $vendeur_id=1; $vendeur_id<=$max_vendeurs; $vendeur_id++ )
 $values[] = "($vendeur_id,'vendeur $vendeur_id')";

query( "INSERT INTO vendeurs (vd_id, vd_name) VALUES
".implode(',',$values) );

for( $jour=0; $jour<$max_jours; $jour++ )
 query( "INSERT INTO recettes_vendeurs (vd_id, rc_date, rc_montant)
SELECT vd_id, DATE_SUB( now(), INTERVAL $jour DAY ), pow(rand(),4) *
10000
FROM vendeurs" );

query( "COMMIT" );
```

Cela nous crée une base de données de test avec 500 000 lignes, ce qui est relativement petit, mais permettra de voir rapidement si une requête est optimisée ou non.

Création de vues (VIEW)

Nous voulons récupérer les informations suivantes :
1) total des recettes par jour ;
2) total des recettes par vendeur et par mois ;
2) total des recettes par mois.

Pour simplifier nos autres requêtes, nous allons utiliser des vues représentant ces trois requêtes.
Une vue (VIEW) est une requête SQL stockée sur le serveur que l'on peut appeler par son nom, et utiliser exactement comme une table.
Cela permet de stocker une tartine de SQL et de l'utiliser simplement.

Code : SQL

```
CREATE VIEW recettes_jour AS
  SELECT rc_date, sum( rc_montant ) AS rc_montant
  FROM recettes_vendeurs
  GROUP BY rc_date;

CREATE VIEW recettes_mois AS
  SELECT YEAR( rc_date )   AS rc_year,
         MONTH( rc_date )  AS rc_month,
         sum( rc_montant ) AS rc_montant
  FROM recettes_vendeurs
  GROUP BY rc_year, rc_month;

CREATE VIEW recettes_vendeur_mois AS
  SELECT YEAR( rc_date )   AS rc_year,
         MONTH( rc_date )  AS rc_month,
         vd_id,
         sum( rc_montant ) AS rc_montant
  FROM recettes_vendeurs
  GROUP BY rc_year, rc_month, vd_id;
```

Une vue se comporte exactement comme une table. Par exemple, on peut la faire apparaître dans un SELECT :

Code : SQL

```
SELECT * FROM recettes_vendeur_mois WHERE rc_year=2009 AND vd_id
BETWEEN 1 AND 2;
+---------+----------+-------+------------+
| rc_year | rc_month | vd_id | rc_montant |
+---------+----------+-------+------------+
|    2009 |        1 |     1 |   56534.90 |
|    2009 |        1 |     2 |   60471.68 |
|    2009 |        2 |     1 |   74446.72 |
|    2009 |        2 |     2 |   53958.03 |
|    2009 |        3 |     1 |   46140.94 |
|    2009 |        3 |     2 |   81320.11 |
|    2009 |        4 |     1 |   34460.82 |
|    2009 |        4 |     2 |   55439.18 |
|    2009 |        5 |     1 |   52062.94 |
|    2009 |        5 |     2 |   70186.89 |
|    2009 |        6 |     1 |   79167.22 |
|    2009 |        6 |     2 |   71725.05 |
|    2009 |        7 |     1 |   75596.33 |
|    2009 |        7 |     2 |   68066.56 |
|    2009 |        8 |     1 |   49466.99 |
|    2009 |        8 |     2 |   71283.26 |
|    2009 |        9 |     1 |   41135.17 |
|    2009 |        9 |     2 |   89311.28 |
|    2009 |       10 |     1 |   63664.04 |
|    2009 |       10 |     2 |   69149.33 |
|    2009 |       11 |     1 |   48181.47 |
|    2009 |       11 |     2 |   67718.38 |
|    2009 |       12 |     1 |   75128.19 |
|    2009 |       12 |     2 |   37845.85 |
+---------+----------+-------+------------+
```

Quels ont été nos meilleurs vendeurs, sur un mois, en 2009 ?

Code : SQL

```
SELECT * FROM recettes_vendeur_mois WHERE rc_year=2009 ORDER BY
rc_montant DESC LIMIT 5;
+---------+----------+-------+------------+
| rc_year | rc_month | vd_id | rc_montant |
+---------+----------+-------+------------+
|    2009 |        7 |    39 |  110352.47 |
|    2009 |        4 |    31 |  109560.18 |
|    2009 |        7 |    56 |  106520.38 |
|    2009 |       12 |    87 |  103366.81 |
|    2009 |        7 |    66 |  100009.22 |
+---------+----------+-------+------------+
```

Pas besoin de réécrire les clauses GROUP BY, etc., on utilise simplement la VIEW.

Faisons maintenant un test de performances. Pour mesurer de manière fiable la durée d'une requête avec MySQL, il faut l'exécuter par exemple en PHP, et mesurer le temps passé avec `microtime()` . Ou, plus simplement, exécuter la requête dans phpMyAdmin.

Cependant, si la même requête est exécutée plusieurs fois de suite, ce qui risque de se produire ici, MySQL va garder le résultat en cache et le chronométrage sera biaisé : à partir de la deuxième exécution, la requête semblera très rapide, en effet elle n'est pas traitée du tout.

Le mot-clé SQL_NO_CACHE, spécifique à MySQL, désactive le cache pour la requête qui le contient. Il doit être placé juste après SELECT. Il faut toujours l'utiliser pour chronométrer une requête.

Nous apercevons ici une limitation de MySQL, particulièrement sur la requête suivante :

Code : SQL

```
SELECT SQL_NO_CACHE * FROM recettes_vendeur_mois WHERE rc_year=2009
AND vd_id BETWEEN 1 AND 2;
```

qui correspond en fait à :

Code : SQL

```
SELECT SQL_NO_CACHE YEAR( rc_date )   AS rc_year,
         MONTH( rc_date )  AS rc_month,
         vd_id,
         sum( rc_montant ) AS rc_montant
  FROM recettes_vendeurs
  WHERE YEAR( rc_date )=2009 AND vd_id BETWEEN 1 AND 2
  GROUP BY rc_year, rc_month, vd_id;
```

En écrivant la requête complète, l'index sur la table « recettes_vendeurs » est utilisé et la requête prend moins de 5 ms.
En utilisant la VIEW, l'index n'est pas utilisé, et par conséquent la requête prend plus de 0.58 seconde, ce qui est énorme.

Une base de données plus évoluée, comme PostgreSQL, est bien sûr capable de transformer la première requête sous la forme de la deuxième, pour utiliser tous les index possibles. Ce n'est pas toujours le cas de MySQL. Par conséquent, les VIEW sont d'une utilité limitée sous MySQL.

Qu'est-ce qu'une vue matérialisée ?

Les vues sont des outils très pratiques, mais leur utilisation peut causer certains problèmes de performances. En effet, on ne peut pas créer d'index sur une VIEW. Par conséquent, sur cette requête :

Code : SQL

```
SELECT * FROM recettes_vendeur_mois WHERE rc_year=2009 ORDER BY
rc_montant DESC LIMIT 5;
```

l'agrégat sum() doit être calculé pour toutes les lignes de l'année 2009, puis les résultats doivent être triés.

Souvent, nous aimerions avoir accès directement aux résultats du calcul, sans devoir effectuer ce calcul. Par exemple, dans un forum, il faut stocker dans la table « topics » ces informations :

- nombre de *posts* dans le *topic* ;
- date du dernier *post* dans le *topic*.

Cela nous permet de poser un index sur la date du dernier *post*, par exemple.

Reprenons notre exemple de comptabilité. Ici, nous n'avons pas de tables existantes (comme la table « topics ») où stocker les résultats de nos calculs ; nous allons donc en créer.

Nous allons matérialiser les vues créées plus haut :

Code : SQL

```
CREATE TABLE recettes_jour_mat (
  rc_date         DATE NOT NULL,
  rc_montant      NUMERIC( 12, 2 ),
  PRIMARY KEY ( rc_date )
) ENGINE=InnoDB;

CREATE TABLE recettes_mois_mat (
  rc_year         INTEGER NOT NULL,
  rc_month        INTEGER NOT NULL,
  rc_montant      NUMERIC( 12, 2 ),
  PRIMARY KEY( rc_year, rc_month )
) ENGINE=InnoDB;

CREATE TABLE recettes_vendeur_mois_mat (
  rc_year         INTEGER NOT NULL,
  rc_month        INTEGER NOT NULL,
  vd_id           INTEGER NOT NULL REFERENCES vendeurs( vd_id ) ON
DELETE RESTRICT,
  rc_montant      NUMERIC( 12, 2 ),
  PRIMARY KEY( rc_year, rc_month, vd_id ),
  KEY( vd_id )
) ENGINE=InnoDB;
```

Après avoir créé ces tables, il va falloir les remplir, puis les tenir à jour en fonction des modifications de la table « recettes_vendeurs ».

Ces opérations peuvent se faire dans l'application, mais cela présente de nombreux inconvénients :

- il faut penser à faire les requêtes de mise à jour à chaque fois qu'on touche à la table « recettes » ;
- l'ajout d'une autre vue matérialisée oblige à ajouter des requêtes de mise à jour un peu partout dans l'application ;
- les performances ne sont pas les meilleures possibles.

La solution correcte pour ce genre de problème est de laisser la base de données s'en occuper, en créant un TRIGGER.

Qu'est-ce qu'un trigger ?

Citation : Documentation PostgreSQL

> Un déclencheur (TRIGGER) spécifie que la base de données doit exécuter automatiquement une fonction donnée chaque fois qu'un certain type d'opération est exécuté. Les fonctions déclencheurs peuvent être définies pour s'exécuter avant ou après une commande INSERT, UPDATE ou DELETE, soit une fois par ligne modifiée, soit une fois par expression SQL.

Nous voulons maintenir à jour nos vues matérialisées en fonction des modifications sur la table « recettes_vendeurs ». Par conséquent, nous allons créer trois *triggers* sur cette table.

La documentation complète se trouve ici.

La commande « delimiter » permet au client MySQL en ligne de commande de ne pas couper les lignes sur les « ; » qui sont dans le code des *triggers*. Si les commandes sont collées dans un phpMyAdmin, il faut l'inclure.

Code : SQL

```
delimiter //

CREATE TRIGGER recettes_vendeurs_insert_t
BEFORE INSERT ON recettes_vendeurs
FOR EACH ROW
BEGIN
    INSERT INTO recettes_jour_mat (rc_date, rc_montant) VALUES
(NEW.rc_date, NEW.rc_montant)
```

```
        ON DUPLICATE KEY UPDATE rc_montant = rc_montant +
NEW.rc_montant;

    INSERT INTO recettes_mois_mat (rc_year, rc_month, rc_montant)
        VALUES (YEAR( NEW.rc_date ), MONTH( NEW.rc_date ),
NEW.rc_montant)
        ON DUPLICATE KEY UPDATE rc_montant = rc_montant +
NEW.rc_montant;

    INSERT INTO recettes_vendeur_mois_mat (rc_year, rc_month, vd_id,
rc_montant)
        VALUES (YEAR( NEW.rc_date ), MONTH( NEW.rc_date ),
NEW.vd_id, NEW.rc_montant)
        ON DUPLICATE KEY UPDATE rc_montant = rc_montant +
NEW.rc_montant;
END//

CREATE TRIGGER recettes_vendeurs_update_t
BEFORE UPDATE ON recettes_vendeurs
FOR EACH ROW
BEGIN
    UPDATE recettes_jour_mat
        SET     rc_montant  = rc_montant - OLD.rc_montant
        WHERE   rc_date     = OLD.rc_date;

    UPDATE recettes_jour_mat
        SET     rc_montant  = rc_montant + NEW.rc_montant
        WHERE   rc_date     = NEW.rc_date;

    UPDATE recettes_mois_mat
        SET     rc_montant  = rc_montant - OLD.rc_montant
        WHERE   rc_year     = YEAR( OLD.rc_date )
            AND rc_month    = MONTH( OLD.rc_date );

    UPDATE recettes_mois_mat
        SET     rc_montant  = rc_montant + NEW.rc_montant
        WHERE   rc_year     = YEAR( NEW.rc_date )
            AND rc_month    = MONTH( NEW.rc_date );

    UPDATE recettes_vendeur_mois_mat
        SET     rc_montant  = rc_montant - OLD.rc_montant
        WHERE   rc_year     = YEAR( OLD.rc_date )
            AND rc_month    = MONTH( OLD.rc_date )
            AND vd_id       = OLD.vd_id;

    UPDATE recettes_vendeur_mois_mat
        SET     rc_montant  = rc_montant + NEW.rc_montant
        WHERE   rc_year     = YEAR( NEW.rc_date )
            AND rc_month    = MONTH( NEW.rc_date )
            AND vd_id       = NEW.vd_id;
END//

CREATE TRIGGER recettes_vendeurs_delete_t
BEFORE DELETE ON recettes_vendeurs
FOR EACH ROW
BEGIN
    UPDATE recettes_jour_mat
        SET     rc_montant  = rc_montant - OLD.rc_montant
        WHERE   rc_date     = OLD.rc_date;

    UPDATE recettes_mois_mat
        SET     rc_montant  = rc_montant - OLD.rc_montant
        WHERE   rc_year     = YEAR( OLD.rc_date )
            AND rc_month    = MONTH( OLD.rc_date );

    UPDATE recettes_vendeur_mois_mat
        SET     rc_montant  = rc_montant - OLD.rc_montant
        WHERE   rc_year     = YEAR( OLD.rc_date )
            AND rc_month    = MONTH( OLD.rc_date )
            AND vd_id       = OLD.vd_id;
```

```
END//

delimiter ;
```

Voilà, nos *triggers* sont créés.

Lors de l'insertion, on utilise INSERT ... ON DUPLICATE KEY UPDATE pour tenir à jour les tables. Cela fonctionne, car nous avons utilisé des clés primaires (uniques) qui conviennent : par exemple, dans « recettes_mois_mat », la clé primaire est (rc_year, rc_month) ; il ne peut y avoir qu'une seule ligne pour un mois d'une année en particulier. C'est ce que nous voulons.

Les *triggers* contiennent les mots-clés NEW et OLD qui font référence à la ligne qui est insérée (NEW), supprimée (OLD) ou mise à jour (OLD = ancienne version, NEW = nouvelle version) dans la table sur laquelle nous avons créé le *trigger*.

Pour le *trigger* sur UPDATE, il serait tout à fait possible de supprimer la moitié des requêtes, en utilisant :

Code : SQL

```
SET rc_montant = rc_montant + (NEW.rc_montant - OLD.rc_montant)
```

à condition cependant de bien vérifier que les colonnes vd_id et rc_date n'ont pas été modifiées.

Initialiser les vues matérialisées

Les *triggers* maintiennent les vues matérialisées à jour, mais si la table « recettes_vendeurs » contient déjà des données, les vues matérialisées ne seront pas initialisées avec les valeurs correctes.

Nous avons deux solutions :

1. vider toutes les tables et remplir « recettes_vendeurs », laissant le trigger d'INSERT faire son travail ;
2. ou bien préremplir les vues matérialisées et n'utiliser les *triggers* que pour les modifications futures.

La première solution est la plus simple, mais la seconde est la plus rapide.

Pour la première solution, nous ferions transiter les données par une table temporaire :

Code : SQL

```
DROP TABLE IF EXISTS recettes_vendeurs_2;
CREATE TABLE recettes_vendeurs_2 AS SELECT * FROM recettes_vendeurs
LIMIT 0;

BEGIN;
INSERT INTO recettes_vendeurs_2 SELECT * FROM recettes_vendeurs FOR
UPDATE;
DELETE FROM recettes_vendeurs;
DELETE FROM recettes_jour_mat;
DELETE FROM recettes_mois_mat;
DELETE FROM recettes_vendeur_mois_mat;
INSERT INTO recettes_vendeurs SELECT * FROM recettes_vendeurs_2;
COMMIT;
DROP TABLE recettes_vendeurs_2;

SELECT * FROM recettes_mois WHERE rc_year=2009;
+---------+----------+------------+
| rc_year | rc_month | rc_montant |
+---------+----------+------------+
|    2009 |        1 |  110529.59 |
|    2009 |        2 |  142694.42 |
|    2009 |        3 |  113394.46 |
|    2009 |        4 |  147740.43 |
|    2009 |        5 |  110104.06 |
```

```
|    2009 |         6 |  113821.84 |
|    2009 |         7 |  146420.22 |
|    2009 |         8 |  112581.44 |
|    2009 |         9 |  106731.73 |
|    2009 |        10 |  192303.25 |
|    2009 |        11 |  114419.72 |
|    2009 |        12 |  107647.62 |
+---------+-----------+------------+
```

La deuxième solution est plus simple, car nous avons déjà créé les vues qui nous intéressent. Il suffit de verrouiller les tables, et de remplir les vues matérialisées avec le contenu des vues précédemment définies. On pourrait aussi ne pas avoir défini de vues du tout et utiliser des requêtes SQL directement.

Code : SQL

```
LOCK TABLES recettes_vendeurs READ, recettes_jour_mat WRITE,
recettes_mois_mat WRITE, recettes_vendeur_mois_mat WRITE;
BEGIN;
DELETE FROM recettes_jour_mat;
DELETE FROM recettes_mois_mat;
DELETE FROM recettes_vendeur_mois_mat;
INSERT INTO recettes_jour_mat SELECT * FROM recettes_jour;
INSERT INTO recettes_mois_mat SELECT * FROM recettes_mois;
INSERT INTO recettes_vendeur_mois_mat SELECT * FROM
recettes_vendeur_mois;
COMMIT;
UNLOCK TABLES;
```

Nos vues matérialisées sont maintenant prêtes.

Les triggers en action

Code : SQL

```
SELECT max(rc_date) FROM recettes_vendeurs;
+--------------+
| max(rc_date) |
+--------------+
| 2010-02-24   |
+--------------+
```

Supposons qu'on se trouve maintenant demain. On ajoute un jour à la date.

Code : SQL

```
SELECT * FROM recettes_vendeurs WHERE rc_date >= '2010-02-25';
--> rien

SELECT * FROM recettes_jour_mat WHERE rc_date >= '2010-02-25';
--> rien

SELECT * FROM recettes_mois_mat WHERE rc_year = 2010 AND rc_month =
2;
+---------+----------+------------+
| rc_year | rc_month | rc_montant |
+---------+----------+------------+
|    2010 |        2 | 5019640.78 |
+---------+----------+------------+
```

Quelques vendeurs insèrent leurs données :

Code : SQL

```
INSERT INTO recettes_vendeurs (vd_id, rc_date, rc_montant) VALUES
(1,'2010-02-25',100), (2,'2010-02-25',1000), (3,'2010-02-25',10),
(4,'2010-02-25',1);
```

Comme par magie, on a maintenant :

Code : SQL

```
SELECT * FROM recettes_jour_mat WHERE rc_date >= '2010-02-25';
+------------+------------+
| rc_date    | rc_montant |
+------------+------------+
| 2010-02-25 |    1111.00 |
+------------+------------+
1 row in set (0,00 sec)

SELECT * FROM recettes_mois_mat WHERE rc_year = 2010 AND rc_month =
2;
+---------+----------+------------+
| rc_year | rc_month | rc_montant |
+---------+----------+------------+
|    2010 |        2 | 5020751.78 |
+---------+----------+------------+
```

Et voilà, le *trigger* a maintenu les informations à jour automatiquement.

Quel est le meilleur vendeur, chaque mois ?

Nous pouvons l'exprimer en SQL comme suit, de deux façons différentes. La seconde donne les ex-æquo aussi.

Code : SQL

```
SELECT
    best.*,
    v.vd_name,
    rm.rc_montant
FROM
    (SELECT
        rm.rc_year,
        rm.rc_month,
        (SELECT vm.vd_id
            FROM recettes_vendeur_mois vm
            WHERE vm.rc_year = rm.rc_year AND vm.rc_month =
rm.rc_month
            ORDER BY rc_montant DESC LIMIT 1
        ) AS vd_id
        FROM recettes_mois rm
    ) AS best
    JOIN vendeurs v ON (v.vd_id = best.vd_id)
    JOIN recettes_vendeur_mois rm
        ON (rm.vd_id = best.vd_id AND rm.rc_year = best.rc_year AND
rm.rc_month = best.rc_month )
ORDER BY rc_year, rc_month;

SELECT
```

```
    best.*,
    rm.vd_id,
    v.vd_name
FROM
    (SELECT rc_year, rc_month, max(rc_montant) AS rc_montant FROM
recettes_vendeur_mois GROUP BY rc_year, rc_month) AS best
    JOIN recettes_vendeur_mois rm
        ON (rm.rc_year = best.rc_year AND rm.rc_month =
best.rc_month AND rm.rc_montant = best.rc_montant )
    JOIN vendeurs v ON (v.vd_id = rm.vd_id)
ORDER BY rc_year, rc_month;
```

Ces requêtes sont un peu compliquées, et aucun index ne peut être utilisé, puisque les conditions sont appliquées au résultat de l'agrégat sum(). Elles sont donc très lentes (2.2 s pour la première, et 1.8 s pour la seconde).

Utilisons nos vues matérialisées. Un de leurs avantages est qu'elles contiennent beaucoup moins de lignes que la table « recettes_vendeurs » :

Code : SQL

```
SELECT count(*) FROM recettes_vendeurs;            500004
SELECT count(*) FROM recettes_vendeur_mois_mat;    16500
SELECT count(*) FROM recettes_mois_mat;            165
SELECT count(*) FROM recettes_jour_mat;            5001
```

Par conséquent, cela risque d'être beaucoup plus rapide. Dans les deux grosses requêtes précédentes, il suffit de remplacer « recettes_vendeur_mois » par « recettes_vendeur_mois_mat » et « recettes_mois » par « recettes_mois_mat ».

Et nous pouvons aussi créer un index sur :

Code : SQL

```
ALTER TABLE recettes_vendeur_mois_mat ADD KEY (rc_year, rc_month,
rc_montant);
```

Résultat : 900 fois plus rapide (en bas à droite). Et pan.

Code : Autre

```
>           VIEWs    mat-VIEWs   mat-VIEWs + index
Requête 1   2.2 s    0.95 s      0.95  s
Requête 2   1.8 s    0.01 s      0.002 s
```

Un trigger pour calculer la valeur par défaut d'une colonne

Supposons que nous ayons une flotte de véhicules à gérer. Les véhicules peuvent être de plusieurs types (tourisme, commercial, etc).

Le système devra gérer les dates de contrôles techniques des véhicules, et calculer automatiquement la date du prochain contrôle, qui dépend du type de véhicule.

Normalement, nous utiliserions une table types_vehicules contenant les différents types de véhicules et leurs intervalles de renouvellement de contrôles techniques. Pour simplifier (et aussi parce que MySQL ne sait pas stocker un intervalle dans une table...), nous allons seulement créer une table pour les véhicules :

Code : SQL

```
CREATE TABLE vehicules(
   vehicule_id          INTEGER PRIMARY KEY AUTO_INCREMENT,
   vehicule_type        VARCHAR(2) NOT NULL, -- 'vt' pour tourisme
et 'vc' pour commercial
   vehicule_date_mes    DATE NOT NULL,  -- date de mise en service
   vehicule_dernier_ct  DATE NULL       -- date du dernier contrôle
technique
) ENGINE=InnoDB;

INSERT INTO vehicules (vehicule_type, vehicule_date_mes,
vehicule_dernier_ct) VALUES
 ('vc', '2010-01-01', NULL),
 ('vt', '2010-01-01', NULL),
 ('vc', '2005-01-01', '2009-01-01'),
 ('vt', '2005-01-01', '2009-01-01');

SELECT * FROM vehicules;
+-------------+--------------+------------------+----------------
----+
| vehicule_id | vehicule_type | vehicule_date_mes |
vehicule_dernier_ct |
+-------------+--------------+------------------+----------------
----+
|           1 | vc            | 2010-01-01       | NULL
|
|           2 | vt            | 2010-01-01       | NULL
|
|           3 | vc            | 2005-01-01       | 2009-01-01
|
|           4 | vt            | 2005-01-01       | 2009-01-01
|
+-------------+--------------+------------------+----------------
----+
```

Pour calculer la date du prochain contrôle technique de chaque véhicule, nous pouvons utiliser une vue :

Code : SQL

```
CREATE VIEW vehicules_ct AS
SELECT *, CASE
     WHEN vehicule_dernier_ct IS NULL THEN
DATE_ADD(vehicule_date_mes, INTERVAL 4 YEAR)
     WHEN vehicule_type = 'vt' THEN DATE_ADD(vehicule_dernier_ct,
INTERVAL 2 YEAR)
     WHEN vehicule_type = 'vc' THEN DATE_ADD(vehicule_dernier_ct,
INTERVAL 1 YEAR)
     ELSE NULL END AS vehicule_prochain_ct
FROM vehicules;

SELECT * FROM vehicules_ct;
+-------------+--------------+------------------+----------------
----+-----------------------+
| vehicule_id | vehicule_type | vehicule_date_mes |
vehicule_dernier_ct | vehicule_prochain_ct |
+-------------+--------------+------------------+----------------
----+-----------------------+
|           1 | vc            | 2010-01-01       | NULL
| 2014-01-01            |
|           2 | vt            | 2010-01-01       | NULL
| 2014-01-01            |
|           3 | vc            | 2005-01-01       | 2009-01-01
| 2010-01-01            |
|           4 | vt            | 2005-01-01       | 2009-01-01
| 2011-01-01            |
+-------------+--------------+------------------+----------------
----+-----------------------+
```

La date du prochain contrôle est recalculée à chaque utilisation de la vue (qui n'est en réalité qu'un SELECT auquel on donne un nom) :

Code : SQL

```
UPDATE vehicules SET vehicule_dernier_ct = CURRENT_DATE WHERE
vehicule_id = 3;
SELECT * FROM vehicules_ct WHERE vehicule_id = 3;
+------------+--------------+------------------+----------------
----+-----------------------+
| vehicule_id | vehicule_type | vehicule_date_mes |
vehicule_dernier_ct | vehicule_prochain_ct |
+------------+--------------+------------------+----------------
----+-----------------------+
|           3 | vc            | 2005-01-01        | 2011-06-20
| 2012-06-20            |
+------------+--------------+------------------+----------------
----+-----------------------+
```

La date du prochain contrôle est donc mise à jour en temps réel. Pour connaître les véhicules dont il faut prévoir le contrôle prochainement, un SELECT suffit :

Code : SQL

```
SELECT *, vehicule_prochain_ct < CURRENT_DATE AS en_retard
FROM vehicules_ct WHERE vehicule_prochain_ct < CURRENT_DATE -
INTERVAL 1 MONTH;
+------------+--------------+------------------+----------------
----+-----------------------+-----------+
| vehicule_id | vehicule_type | vehicule_date_mes |
vehicule_dernier_ct | vehicule_prochain_ct | en_retard |
+------------+--------------+------------------+----------------
----+-----------------------+-----------+
|           4 | vt            | 2005-01-01        | 2009-01-01
| 2011-01-01            |         1 |
+------------+--------------+------------------+----------------
----+-----------------------+-----------+
```

Un avantage appréciable de la vue est que l'algorithme de calcul de la date du prochain contrôle est placé dans la vue, au lieu d'être copié-collé dans l'application en plusieurs endroits, ce qui rend le système beaucoup plus facile à modifier si la législation vient à changer.

Nous aurions aussi pu placer ce calcul dans une fonction, par exemple.

Ici, la condition du WHERE est appliquée sur une colonne qui est calculée par la vue. MySQL ne supportant pas l'indexation des expressions ou des vues, cette condition n'est pas indexable, donc toutes les lignes devront être examinées, ce qui peut prendre un certain temps sur une grosse table.

Nous allons matérialiser la colonne vehicule_prochain_ct dans la table vehicules pour pouvoir l'indexer, et la tenir à jour avec des triggers.

Il s'agit d'une dénormalisation puisque cette colonne ne contient aucune information nouvelle (elle est calculée à partir de colonnes existantes). Le choix de la matérialiser dans la table est donc un compromis : on dénormalise pour gagner en performance.

Code : SQL

```
DROP TABLE vehicules;
```

```
CREATE TABLE vehicules(
   vehicule_id          INTEGER PRIMARY KEY AUTO_INCREMENT,
   vehicule_type        VARCHAR(2) NOT NULL, -- 'vt' pour tourisme
et 'vc' pour commercial
   vehicule_date_mes    DATE NOT NULL,   -- date de mise en service
   vehicule_dernier_ct  DATE NULL,       -- date du dernier contrôle
technique
   vehicule_prochain_ct DATE NULL        -- date du prochain
contrôle technique
) ENGINE=InnoDB;
```

La table est la même que la précédente, plus la colonne vehicule_prochain_ct.

Un trigger BEFORE INSERT (ou BEFORE UPDATE) peut manipuler les données **avant** leur insertion dans la table, ce qui permet, entre autres, de :

- Mettre dans une colonne une valeur par défaut dépendant des autres colonnes ou d'un calcul
- Vérifier la validité des informations et renvoyer une erreur le cas échéant
- Entraîner une action si la valeur d'une colonne change dans un UPDATE (NEW.colonne != OLD.colonne)
- etc.

Un trigger AFTER INSERT (ou AFTER UPDATE) ne peut pas manipuler ces données, car la table a déjà été mise à jour lors de l'appel du trigger.

Pour modifier la ligne avant son insertion, on n'utilise pas la commande UPDATE (qui modifie le contenu d'une table). Il faut modifier le contenu de la variable NEW. MySQL utilise une syntaxe particulière (SET), à retenir.

Dans un trigger BEFORE INSERT, nous avons accès à la variable NEW uniquement (contenu de la ligne à insérer, provenant des VALUES() de la commande INSERT, plus les DEFAULT)
Dans un trigger BEFORE UPDATE, nous avons accès aux variables NEW (nouveau contenu de la ligne, modifié par l'UPDATE) et OLD (ancien contenu de la ligne).

Ce trigger sera donc très simple : il suffit de changer la valeur de NEW.vehicule_prochain_ct en fonction des autres colonnes.

Code : SQL

```
DELIMITER //
CREATE TRIGGER vehicules_bi BEFORE INSERT
ON vehicules
FOR EACH ROW
BEGIN
   SET NEW.vehicule_prochain_ct = CASE
     WHEN NEW.vehicule_dernier_ct IS NULL THEN
DATE_ADD(NEW.vehicule_date_mes, INTERVAL 4 YEAR)
     WHEN NEW.vehicule_type = 'vt' THEN
DATE_ADD(NEW.vehicule_dernier_ct, INTERVAL 2 YEAR)
     WHEN NEW.vehicule_type = 'vc' THEN
DATE_ADD(NEW.vehicule_dernier_ct, INTERVAL 1 YEAR)
     ELSE NULL END;
END;
//
CREATE TRIGGER vehicules_bu BEFORE UPDATE
ON vehicules
FOR EACH ROW
BEGIN
   SET NEW.vehicule_prochain_ct = CASE
     WHEN NEW.vehicule_dernier_ct IS NULL THEN
DATE_ADD(NEW.vehicule_date_mes, INTERVAL 4 YEAR)
     WHEN NEW.vehicule_type = 'vt' THEN
DATE_ADD(NEW.vehicule_dernier_ct, INTERVAL 2 YEAR)
     WHEN NEW.vehicule_type = 'vc' THEN
DATE_ADD(NEW.vehicule_dernier_ct, INTERVAL 1 YEAR)
```

```
        ELSE NULL END;
    END;
    //
    DELIMITER ;
```

Dans le trigger BEFORE UPDATE, NEW.vehicule_prochain_ct est toujours modifié : tout UPDATE modifiant cette colonne sera donc écrasé avec la valeur calculée.

Il serait possible de modifier ce comportement : soit permettre la modification (pourquoi faire ?), soit provoquer une erreur si une modification est tentée (avec un IF NEW.vehicule_prochain_ct != OLD.vehicule_prochain_ct placé avant le SET).

Code : SQL

```
INSERT INTO vehicules (vehicule_type, vehicule_date_mes,
vehicule_dernier_ct) VALUES
 ('vc', '2010-01-01', NULL),
 ('vt', '2010-01-01', NULL),
 ('vc', '2005-01-01', '2009-01-01'),
 ('vt', '2005-01-01', '2009-01-01');

SELECT * FROM vehicules;
+------------+--------------+------------------+----------------
----+----------------------+
| vehicule_id | vehicule_type | vehicule_date_mes |
vehicule_dernier_ct | vehicule_prochain_ct |
+------------+--------------+------------------+----------------
----+----------------------+
|           1 | vc           | 2010-01-01       | NULL
| 2014-01-01           |
|           2 | vt           | 2010-01-01       | NULL
| 2014-01-01           |
|           3 | vc           | 2005-01-01       | 2009-01-01
| 2010-01-01           |
|           4 | vt           | 2005-01-01       | 2009-01-01
| 2011-01-01           |
+------------+--------------+------------------+----------------
----+----------------------+
```

Les modifications sont gérées par le trigger BEFORE UPDATE :

Code : SQL

```
UPDATE vehicules SET vehicule_dernier_ct = CURRENT_DATE WHERE
vehicule_id = 3;
SELECT * FROM vehicules_ct WHERE vehicule_id = 3;
+------------+--------------+------------------+----------------
----+----------------------+
| vehicule_id | vehicule_type | vehicule_date_mes |
vehicule_dernier_ct | vehicule_prochain_ct |
+------------+--------------+------------------+----------------
----+----------------------+
|           3 | vc           | 2005-01-01       | 2011-06-20
| 2012-06-20           |
+------------+--------------+------------------+----------------
----+----------------------+
```

Le comportement est identique à celui de la vue : la colonne vehicule_prochain_ct est tenue à jour.

Les modifications (INSERT, UPDATE) sont légèrement ralenties puisqu'il faut exécuter le trigger. Par contre, la colonne vehicule_prochain_ct existe réellement, elle peut donc être indexée, ce qui peut être utile.

Un trigger pour historiser les modifications d'une table

Qui a modifié cette maudite fiche client ? C'est encore la faute du stagiaire !...

Certaines bases de données proposent des fonctions PITR (Point in time recovery) pour retrouver l'état de la base de donnée à un instant précis, ou bien des systèmes de versions qui permettent de tracer l'historique d'une ligne dans une table.

Ici, nous verrons un exemple simple d'utilisation de triggers pour enregistrer une historique des modifications d'une table.

Soit une table d'utilisateurs :

Code : SQL

```
CREATE TABLE users (
   user_id           INTEGER PRIMARY KEY AUTO_INCREMENT,
   user_create_dt    DATETIME NOT NULL,          -- date de création

   -- colonnes historisées :
   user_name         VARCHAR( 255 ) NOT NULL,
   user_level        INTEGER NOT NULL DEFAULT 0, -- droits de
l'utilisateur
   user_comment      TEXT NULL,                  -- commentaires de
l'admin
   user_lastmod_dt   DATETIME NOT NULL,          -- date de dernière
modification
   user_lastmod_db_user VARCHAR( 255 ) NOT NULL, -- qui a effectué
la dernière modification (login mysql)
   user_lastmod_user_id INTEGER NULL             -- qui a effectué
la dernière modification (id utilisateur)
                        REFERENCES users( user_id ) ON DELETE SET
NULL
) ENGINE=InnoDB;
```

Mettons que user_level vaut 0 pour un utilisateur standard, 1 pour un modérateur, et 2 pour un admin par exemple.

Une table pour l'historique :

Code : SQL

```
CREATE TABLE users_history (
   user_id           INTEGER NOT NULL REFERENCES users( user_id ) ON
DELETE CASCADE,
   user_lastmod_dt   DATETIME NOT NULL,
   PRIMARY KEY (user_id,user_lastmod_dt),

   -- colonnes historisées :
   user_name         VARCHAR( 255 ) NOT NULL,
   user_level        INTEGER NOT NULL DEFAULT 0, -- droits de
l'utilisateur
   user_comment      TEXT NULL,                  -- commentaires de
l'admin
   user_lastmod_db_user VARCHAR( 255 ) NOT NULL, -- qui a effectué
la dernière modification (login mysql)
   user_lastmod_user_id INTEGER NULL             -- qui a effectué
la dernière modification (id utilisateur)
                        REFERENCES users( user_id ) ON DELETE SET
NULL
) ENGINE=InnoDB;
```

Nous allons donc stocker les informations **courantes** dans la table users, et les informations **passées** (l'historique) dans users_history.

Il y a d'autres façons de faire : par exemple nous aurions pu définir que les valeurs courantes sont données par l'enregistrement le plus récent dans users_history, et supprimer les colonnes concernées de la table users, mais cela générerait des jointures complexes pour récupérer les dernières informations, ce qui peut être gênant si on les utilise souvent.

Ici, les suppressions ne seront pas gérées : la suppression d'un utilisateur entraînera la suppression de son historique. Nous pourrions aussi créer des tables d'archivage, et au moyen d'un trigger ON DELETE, déplacer la ligne de la table users, et les lignes de la table users_history qui en dépendent, vers ces tables d'archivage. Pour des utilisateurs (cas d'un forum par exemple) le choix le plus courant est de ne pas effacer les utilisateurs mais plutôt de les désactiver, car cela entraînerait des problèmes avec les références depuis les tables de posts (affichage de pseudos inconnus, etc).

Définissions les triggers : quand une modification est effectuée sur la table des utilisateurs, les valeurs précédentes doivent être déplacées vers la table d'historique. Puisque les valeurs courantes sont stockées dans la table users, l'insertion d'une ligne ne créera pas d'historique, mais il nous faut tout de même un trigger BEFORE INSERT pour gérer les valeurs par défaut des colonnes.

Pour enregistrer l'utilisateur qui a effectué les modifications, il y a plusieurs options :

- Utiliser l'utilisateur SQL courant : c'est le plus robuste puisque MySQL gère les permissions et les comptes, mais cela oblige à créer un compte MySQL par utilisateur.
- Utiliser une variable de session qui contient l'id de l'utilisateur qui fait la modification (ici, un utilisateur de site web, stocké dans la table users) ce qui permet d'éviter de créer un compte MySQL par utilisateur, mais oblige à initialiser une variable de session à la connection.

Les deux façons de faire sont montrées ici.

Code : SQL

```
DELIMITER //
CREATE TRIGGER users_bi BEFORE INSERT
ON users
FOR EACH ROW
BEGIN
   -- MySQL ne sait pas faire "DEFAULT now()"...
   SET NEW.user_create_dt  = NOW();

   -- Valeurs par défaut des autres colonnes :
   SET NEW.user_lastmod_dt      = NEW.user_create_dt;
   SET NEW.user_lastmod_db_user = CURRENT_USER;
   SET NEW.user_lastmod_user_id = @current_user_id;
END;
//
CREATE TRIGGER users_bu BEFORE UPDATE
ON users
FOR EACH ROW
BEGIN
   -- mise à jour des informations courantes dans la table users
   -- par modification de la ligne en cours (variable NEW)
   SET NEW.user_lastmod_dt      = NOW();
   SET NEW.user_lastmod_db_user = CURRENT_USER;
   SET NEW.user_lastmod_user_id = @current_user_id;

   -- enregistrement dans la table d'historique des valeurs
précédentes (variable OLD)
   INSERT INTO users_history (
      user_id,
      user_lastmod_dt,
      user_name,
      user_level,
      user_comment,
```

```
            user_lastmod_db_user,
            user_lastmod_user_id)
        VALUES (
            OLD.user_id,
            OLD.user_lastmod_dt,
            OLD.user_name,
            OLD.user_level,
            OLD.user_comment,
            OLD.user_lastmod_db_user,
            OLD.user_lastmod_user_id
        );
    END;
    //
    DELIMITER ;
```

En avant pour le test : il faut d'abord créer un utilisateur "racine" qui devra créer les autres.

Code : SQL

```
mysql> INSERT INTO users (user_name, user_level) VALUES ('Admin',2);
Query OK, 1 row affected, 3 warnings (0.02 sec)

mysql> SHOW WARNINGS;
+---------+------+-----------------------------------------------------
---------+
| Level   | Code | Message
|
+---------+------+-----------------------------------------------------
---------+
| Warning | 1364 | Field 'user_create_dt' doesn't have a default
value |
| Warning | 1364 | Field 'user_lastmod_dt' doesn't have a default
value      |
| Warning | 1364 | Field 'user_lastmod_db_user' doesn't have a
default value |
+---------+------+-----------------------------------------------------
---------+

mysql> select * from users;
+---------+---------------------+----------+------------+----------
----+---------------------+---------------------+------------------
----+
| user_id | user_create_dt      | user_name | user_level |
user_comment | user_lastmod_dt     | user_lastmod_db_user |
user_lastmod_user_id |
+---------+---------------------+----------+------------+----------
----+---------------------+---------------------+------------------
----+
|       1 | 2011-06-20 16:58:21 | Admin     |          2 | NULL
| 2011-06-20 16:58:21 | lord@localhost       |                  NULL
|
+---------+---------------------+----------+------------+----------
----+---------------------+---------------------+------------------
----+
```

Il y a des warnings... en effet les 3 colonnes qui sont initialisées par le trigger valent, au départ, NULL. Et elles sont spécifiées comme NOT NULL. Pour éviter les warnings, il faudrait se répéter et mettre des valeurs par défaut dans l'INSERT, ce qui n'est pas vraiment utile. Le trigger a en tout cas fonctionné : nous ne voyons pas "ERROR 1048 (23000): Column 'machin' cannot be null".

Supposons que l'utilisateur "admin" se log sur le site et créé deux utilisateurs :

Code : SQL

```
SET @current_user_id = 1; -- effectué au login sur le site

INSERT INTO users (user_name, user_level) VALUES ('Modérateur',1);
INSERT INTO users (user_name, user_level) VALUES ('Machin',0);

+---------+---------------------+------------+------------+--------
------+--------------------+---------------------+---------------
------+
| user_id | user_create_dt      | user_name   | user_level |
user_comment | user_lastmod_dt     | user_lastmod_db_user |
user_lastmod_user_id |
+---------+---------------------+------------+------------+--------
------+--------------------+---------------------+---------------
------+
|       1 | 2011-06-20 16:58:21 | Admin       |          2 | NULL
| 2011-06-20 16:58:21 | lord@localhost       |                  NULL
|
|       2 | 2011-06-20 16:58:43 | Modérateur  |          1 | NULL
| 2011-06-20 16:58:43 | lord@localhost       |                     1
|
|       3 | 2011-06-20 16:58:43 | Machin      |          0 | NULL
| 2011-06-20 16:58:43 | lord@localhost       |                     1
|
+---------+---------------------+------------+------------+--------
------+--------------------+---------------------+---------------
------+
```

Les colonnes de DATETIME ainsi que user_lastmod_db_user et user_lastmod_user_id ont bien été mises à jour.

Supposons que l'utilisateur "Modérateur" se log sur le site et modifie l'utilisateur "Machin" pour lui donner des droits :

Code : SQL

```
SET @current_user_id = 2; -- effectué au login sur le site

UPDATE users SET user_level = 1, user_comment = 'Pistonné' WHERE
user_id = 3;

mysql> SELECT * FROM users;
+---------+---------------------+------------+------------+--------
------+--------------------+---------------------+---------------
------+
| user_id | user_create_dt      | user_name   | user_level |
user_comment | user_lastmod_dt     | user_lastmod_db_user |
user_lastmod_user_id |
+---------+---------------------+------------+------------+--------
------+--------------------+---------------------+---------------
------+
|       1 | 2011-06-20 16:58:21 | Admin       |          2 | NULL
| 2011-06-20 16:58:21 | lord@localhost       |                  NULL
|
|       2 | 2011-06-20 16:58:43 | Modérateur  |          1 | NULL
| 2011-06-20 16:58:43 | lord@localhost       |                     1
|
|       3 | 2011-06-20 16:58:43 | Machin      |          1 |
Pistonné     | 2011-06-20 16:59:11 | lord@localhost       |
2 |
+---------+---------------------+------------+------------+--------
------+--------------------+---------------------+---------------
------+

mysql> SELECT * FROM users_history;
+---------+---------------------+-----------+------------+---------
----+--------------------+
| user id | user lastmod dt     | user name | user level |
```

```
user_comment | user_lastmod_db_user | user_lastmod_user_id |
+---------+---------------------+----------+-----------+----------
----+---------------------+---------------------+
|       3 | 2011-06-20 16:58:43 | Machin    |          0 | NULL
| lord@localhost       |                   1 |
+---------+---------------------+----------+-----------+----------
----+---------------------+---------------------+
```

Le trigger a modifié les colonnes user_lastmod_dt et user_lastmod_user_id dans la table users, et inséré une ligne dans users_history avec les anciennes informations.

Historisation : variante

Nous pouvons aussi stocker les informations courantes dans la table d'historique. Pour les retrouver facilement, nous utiliserons la date de dernière modification de la table principale.

Code : SQL

```
CREATE TABLE users (
   user_id           INTEGER PRIMARY KEY AUTO_INCREMENT,
   user_create_dt    DATETIME NOT NULL,          -- date de création
   user_lastmod_dt   DATETIME NOT NULL           -- date de dernière
modification
) ENGINE=InnoDB;

CREATE TABLE users_history (
   user_id           INTEGER NOT NULL REFERENCES users( user_id ) ON
DELETE CASCADE,
   user_lastmod_dt   DATETIME NOT NULL,
   PRIMARY KEY (user_id,user_lastmod_dt),

   -- colonnes historisées :
   user_name         VARCHAR( 255 ) NOT NULL,
   user_level        INTEGER NOT NULL DEFAULT 0, -- droits de
l'utilisateur
   user_comment      TEXT NULL,                  -- commentaires de
l'admin
   user_lastmod_db_user VARCHAR( 255 ) NOT NULL, -- qui a effectué
la dernière modification (login mysql)
   user_lastmod_user_id INTEGER NULL             -- qui a effectué
la dernière modification (id utilisateur)
                        REFERENCES users( user_id ) ON DELETE SET
NULL
) ENGINE=InnoDB;

DELIMITER //
CREATE TRIGGER users_history_ai BEFORE INSERT
ON users_history
FOR EACH ROW
BEGIN
   SET NEW.user_lastmod_dt      = NOW();
   SET NEW.user_lastmod_db_user = CURRENT_USER;
   SET NEW.user_lastmod_user_id = @current_user_id;

   -- reporte le timestamp de la dernière version de l'historique
dans la table users
   UPDATE users SET user_lastmod_dt = NEW.user_lastmod_dt WHERE
user_id=NEW.user_id;
END;
//
DELIMITER ;

-- une vue pour récupérer l'état courant
```

```
CREATE VIEW users_current AS
SELECT * FROM users LEFT JOIN users_history USING (user_id,
user_lastmod_dt);
```

L'insertion d'un utilisateur se fait en deux parties puisqu'il faut entrer une ligne dans chaque table. Les valeurs de datetime à insérer dans users seront modifiées par l'insertion dans users_history, mais comme les colonnes sont NOT NULL, il faut mettre une valeur bidon, par exemple now().

Code : SQL

```
SET @current_user_id = NULL;

BEGIN;
INSERT INTO users (user_create_dt,user_lastmod_dt) VALUES
(NOW(),NOW());
INSERT INTO users_history (user_id, user_name, user_level) VALUES
(LAST_INSERT_ID(), 'Admin', 2);
COMMIT;

-- Admin se log
SET @current_user_id = 1; -- effectué au login sur le site

-- Admin créé 2 utilisateurs
BEGIN;
INSERT INTO users (user_create_dt,user_lastmod_dt) VALUES
(NOW(),NOW());
INSERT INTO users_history (user_id, user_name, user_level) VALUES
(LAST_INSERT_ID(), 'Modérateur', 1);

INSERT INTO users (user_create_dt,user_lastmod_dt) VALUES
(NOW(),NOW());
INSERT INTO users_history (user_id, user_name, user_level) VALUES
(LAST_INSERT_ID(), 'Machin', 0);
COMMIT;


mysql> SELECT * FROM users_current;
+---------+---------------------+---------------------+------------
+------------+-------------+---------------------+---------------
------+
| user_id | user_lastmod_dt     | user_create_dt      | user_name
| user_level | user_comment | user_lastmod_db_user |
user_lastmod_user_id |
+---------+---------------------+---------------------+------------
+------------+-------------+---------------------+---------------
------+
|       1 | 2011-06-20 17:28:48 | 2011-06-20 17:28:48 | Admin
|          2 | NULL         | lord@localhost       |
NULL |
|       2 | 2011-06-20 17:34:51 | 2011-06-20 17:34:51 | Modérateur
|          1 | NULL         | lord@localhost       |
NULL |
|       3 | 2011-06-20 17:34:51 | 2011-06-20 17:34:51 | Machin
|          0 | NULL         | lord@localhost       |
NULL |
+---------+---------------------+---------------------+------------
+------------+-------------+---------------------+---------------
------+
```

Avec la VIEW, le résultat est identique à celui de la méthode précédente. On ne sait pas d'où viennent les données.

On pourrait éventuellement créer une RULE sur la VIEW, pour qu'un UPDATE sur cette VIEW soit transformé directement en INSERT dans la table d'historique. Mais MySQL ne supporte pas les RULEs... ni les triggers sur les views d'ailleurs.

Un UPDATE sur la table users doit donc être remplacé par un INSERT dans la table users_history, avec les nouvelles valeurs à utiliser.

Comme plus haut, supposons que l'utilisateur "Modérateur" se log sur le site et modifie l'utilisateur "Machin" pour lui donner des droits :

Code : SQL

```
SET @current_user_id = 2; -- effectué au login sur le site

INSERT INTO users_history (user_id, user_name, user_level,
user_comment) VALUES (3, 'Machin', 1, 'Pistonné');

mysql> SELECT * FROM users_current;
+---------+---------------------+---------------------+-------------
+------------+-------------+---------------------+---------------
------+
| user_id | user_lastmod_dt     | user_create_dt      | user_name
| user_level | user_comment | user_lastmod_db_user |
user_lastmod_user_id |
+---------+---------------------+---------------------+-------------
+------------+-------------+---------------------+---------------
------+
|       1 | 2011-06-20 17:28:48 | 2011-06-20 17:28:48 | Admin
|          2 | NULL         | lord@localhost       |
NULL |
|       2 | 2011-06-20 17:34:51 | 2011-06-20 17:34:51 | Modérateur
|          1 | NULL         | lord@localhost       |
NULL |
|       3 | 2011-06-20 17:36:55 | 2011-06-20 17:34:51 | Machin
|          1 | Pistonné     | lord@localhost       |
2 |
+---------+---------------------+---------------------+-------------
+------------+-------------+---------------------+---------------
------+
3 rows in set (0.00 sec)

mysql> SELECT * FROM users_history;
+---------+---------------------+------------+------------+--------
------+---------------------+---------------------+
| user_id | user_lastmod_dt     | user_name  | user_level |
user_comment | user_lastmod_db_user | user_lastmod_user_id |
+---------+---------------------+------------+------------+--------
------+---------------------+---------------------+
|       1 | 2011-06-20 17:28:48 | Admin      |          2 | NULL
| lord@localhost       |                NULL |
|       2 | 2011-06-20 17:34:51 | Modérateur |          1 | NULL
| lord@localhost       |                NULL |
|       3 | 2011-06-20 17:34:51 | Machin     |          0 | NULL
| lord@localhost       |                NULL |
|       3 | 2011-06-20 17:36:55 | Machin     |          1 |
Pistonné     | lord@localhost       |                   2 |
+---------+---------------------+------------+------------+--------
------+---------------------+---------------------+
```

La table users_history contient alors au minimum une ligne par utilisateur (contrairement à l'exemple précédent où elle pouvait n'en contenir aucune) et ici, une ligne plus ancienne pour l'utilisateur 3.

Quelle solution est la meilleure ? Ça dépend...

Si on souhaitait ajouter d'autres informations à l'historique, qui n'ont pas leur place dans la table principale, la seconde solution est la plus pertinente : par exemple, pour gérer des véhicules, la table d'historique pourrait contenir des informations sur les interventions effectuées, réparations, devis, factures, et commentaires. La vue qui donne l'état courant du véhicule ne pourrait montrer que certaines colonnes de cette table d'historique, par exemple l'état courant. Mais on s'éloigne d'une historisation

pure...

Autres exemples de triggers

Voici quelques exemples (non vérifiés) de ce que l'on peut faire avec des *triggers*.

Mettre à jour les colonnes d'un *topic* à chaque fois qu'un *post* est inséré dedans :

Code : SQL

```
CREATE TRIGGER posts_insert_t
AFTER INSERT ON posts
FOR EACH ROW
BEGIN
    UPDATE topics SET
        last_post_id = NEW.post_id,
        last_post_time = NEW.post_time,
        last_post_user_id = NEW.user_id,
        post_count = post_count + 1
        WHERE topic_id = NEW.topic_id;
END;
```

Un *trigger* du même style mettra à jour la table « forums » en fonction des *topics*, et il faudra gérer la suppression des topics et des posts pour décrémenter le nombre de topics/posts.

On peut aussi, lors de l'insertion ou de la modification d'une ligne, utiliser un *trigger* pour ajouter une tâche à faire plus tard dans une table de *jobs*. Un programme en tâche de fond examine régulièrement cette table, exécute les *jobs* et supprime les lignes. Cela permet de répondre rapidement à l'utilisateur, sans devoir attendre la fin du traitement. Par exemple :

- réindexer un texte dans Xapian ;
- redimensionner une image ;
- réencoder un fichier vidéo ;
- faire valider un profil par un humain ;
- envoyer des *newsletters* ;
- mettre à jour un cache ;
- etc.

Les *triggers* peuvent aussi être utilisés pour vérifier des contraintes : comme MySQL ne supporte pas de contrainte CHECK, on peut utiliser un *trigger* qui vérifie les données ; et si elles sont erronées, il provoque une erreur.

Méfions-nous des locks

Il faut aussi se méfier des *locks* : dans l'exemple cité, les INSERT sur la table « recettes_vendeurs » déclenchent une UPDATE sur la table « recettes_mois ». Par conséquent, on ne peut exécuter simultanément deux INSERT INTO recettes_vendeurs, si les deux dates insérées sont dans le même mois : les INSERT seront sérialisées.

À l'extrême, si l'on utilisait des *triggers* pour mettre à jour un compteur global par exemple, tous les INSERTS seraient sérialisées. Il faut tenter d'éviter cette situation, et utiliser des transactions courtes (par exemple, l'*output buffering* de PHP doit être activé), car les *locks* subsistent jusqu'au COMMIT.

Méfions-nous de MySQL

MySQL ne déclenche pas les triggers pour toute opération qui résulte d'une contrainte de clé étrangère.

Autrement dit, si nous faisons un DELETE sur la table topics pour supprimer un topic, les posts de ce topic seront supprimés (par ON DELETE CASCADE sur la clé étrangère topic_id), mais un éventuel trigger ON DELETE sur la table posts ne sera pas appelé.

Par conséquent, si on utilise les triggers pour tenir des comptes à jour, il faut bien garder cela à l'esprit. Par exemple, un trigger ON DELETE sur la table des topics devra décrémenter le nombre de topics dans le forum parent, mais aussi le nombre de posts, et ne pas déléguer cette opération aux triggers de la table posts.

Toutes les autres BDD (y compris sqlite) permettent aux triggers de fonctionner normalement suite aux opérations de cascade de clés étrangères, donc cet avertissement ne s'applique qu'à MySQL.
Nous avons vu sur un petit exemple comment utiliser des *triggers* pour maintenir à jour des tables qui contiennent une représentation d'informations déjà présentes dans la base de données, mais sous une autre forme, en l'occurrence des agrégats et des sommes par mois et par vendeur.

Bien sûr, maintenir ces informations à jour a un coût : il faut exécuter les requêtes présentes dans le *trigger*. Toute opération d'écriture (hors SELECT, donc) sur une table dotée de *triggers* sera plus lente que sans *triggers*.

Toutefois, comme on l'a vu, l'utilisation de *materialized views* permet d'accélérer certaines requêtes d'une façon stupéfiante, car :

- les tables obtenues sont de vraies tables, les données sont immédiatement accessibles, et non le résultat d'un calcul potentiellement lent ;
- on peut créer des index dessus ;
- elles peuvent être beaucoup plus petites que les tables d'origine, si la vue en question est une agrégation (dernier élément, somme, etc.).

Ce type de technique est donc adapté dans une situation où l'on écrit peu et où l'on lit beaucoup.

Ce tutoriel a été corrigé par les zCorrecteurs.