La gestion des erreurs en SQL procédural avec MySQL

Par Alain Defrance

Date de publication : 17 septembre 2008

Depuis la version 5, MySQL supporte la programmation intégrée (ou SQL procédural), qui permet de le rendre très autonome. Nous verrons dans cet article quelles sont les possibilités qui s'offrent à nous pour reporter la programmation sur le SGBD. Je ne prétends pas aborder toutes les possibilités du SQL procédural, mais seulement proposer une des très nombreuses utilisations que l'on peut en faire. Ce tutoriel peut constituer une introduction au SQL procédural pour les débutants, et une solution efficace de gestion d'erreurs pour les plus confirmés.

I - Petit rappel du rôle d'un SGBDR

Le SGBD (ou Système de Gestion de Base de Données) est un programme permettant de stocker les informations. Dans le passé, sans SGBD, nous étions obligés de faire nous-mêmes de la lecture / écriture de fichiers et gérer l'organisation physiques des données.
C'était une tâche très lourde, car avec des données nombreuses, à l'organisation complexe, la création et l'optimisation des algorithmes d'accès aux données étaient difficiles à réaliser. Avec le temps, des programmes ont proposé divers moyens pour optimiser ces tâches, et surtout pour éviter de réinventer la roue à chaque fois.
Le but d'un SGBD est bien entendu de proposer une interface entre les données physiques et une application quelconque. Mais le rôle des SGBDR (Système de Gestion de Base de Données Relationnelles) est bien plus ambitieux : **le contrôle de l'intégrité**. Le contrôle de l'intégrité est une tâche permettant d'assurer la **cohérence des données** au travers de contraintes.
Il nous suffit alors de préciser des règles (les contraintes) qui définissent les valeurs possibles que peuvent prendre telle ou telle colonne. Ces contraintes permettent d'alléger énormément les contrôles à faire du côté applicatif. Il est conseillé d'avoir lu "**Limiter la complexité du code applicatif grâce au SGBD**" qui parle de ce type de pratique.
Même si les contraintes permettent une grande liberté d'action, elles sont parfois insuffisantes pour gérer parfaitement l'intégrité de notre domaine de gestion. En effet, l'aspect métier possède très souvent une sémantique propre qui dépasse largement la simple gestion de l'intégrité référentielle. Puisque le rôle du SGBDR est de gérer parfaitement la cohérence des données, il est alors nécessaire de pouvoir y implémenter son propre code et ses propres fonctions afin de contrôler l'information en fonction de notre domaine de gestion.
Ainsi, le SQL intégré (ou SQL procédural) permet de programmer le SGBD pour réaliser telle ou telle opération, qu'elle permette de gérer l'intégrité ou simplement de faciliter l'exploitation des données.
Le PL/SQL (Oracle), ou le T-SQL (SQL-Server) sont du SQL procédural. Ici, nous allons utiliser le langage de MySQL (qui ne porte pour le moment aucun nom).
Le but est de rendre la base de données la plus autonome possible, et qu'il ne dépende d'aucune application pour garantir une intégrité parfaite.

II - Un exemple qui pose problème

Sans plus attendre, nous allons voir un cas simple de problème qui peut se poser.
Nous souhaitons gérer un système de participation à des courses.

Création de la structure des données

```
CREATE TABLE `Sportif`
(
 `numSportif` INTEGER AUTO_INCREMENT,
 `nomSportif` VARCHAR(20) NOT NULL,
 `prenomSportif` VARCHAR(20) NOT NULL,
 CONSTRAINT `PK_SPORTIF` PRIMARY KEY (`numSportif`)
) ENGINE = `innoDB`;

CREATE TABLE `Course`
(
 `numCourse` INTEGER AUTO_INCREMENT,
 `libelleCourse` varchar(50) NOT NULL,
 `dateCourse` DATETIME NOT NULL,
 `nbMaxParticipant` INTEGER NOT NULL,
 CONSTRAINT `PK_SPORTIF` PRIMARY KEY (`numCourse`)
) ENGINE = `innoDB`;

CREATE TABLE `Participer`
(
 `numSportif` INTEGER NOT NULL,
 `numCourse` INTEGER NOT NULL,
 `tempsParticipation` FLOAT NULL,
 CONSTRAINT `PK_PARTICIPER` PRIMARY KEY(`numSportif`, `numCourse`)
) ENGINE = `innoDB`;

ALTER TABLE `Participer`
ADD CONSTRAINT `FK_PARTICIPER_SPORTIF` FOREIGN KEY (`numSportif`) REFERENCES `Sportif`(`numSportif`),
ADD CONSTRAINT `FK_PARTICIPER_COURSE` FOREIGN KEY (`numCourse`) REFERENCES `Course`(`numCourse`);
```

Création de la structure des données

Insertion d'un jeux d'essai

```
INSERT INTO `Sportif` (`numSportif`, `nomSportif`, `prenomSportif`)
VALUES
 (1, 'MARTIN', 'Pierre'),
 (2, 'BERNARD', 'Jean'),
 (3, 'THOMAS', 'Jacques'),
 (4, 'PETIT', 'François'),
 (5, 'DURAND', 'Charles'),
 (6, 'RICHARD', 'Louis'),
 (7, 'DUBOIS', 'Jean-Baptiste'),
 (8, 'ROBERT', 'Joseph'),
 (9, 'LAURENT', 'Nicolas'),
 (10, 'SIMON', 'Antoine'),
 (11, 'MICHEL', 'Marie'),
 (12, 'LEROY', 'Marguerite');

INSERT INTO `Course` (`numCourse`, `libelleCourse`, `dateCourse`, `nbMaxParticipants`)
VALUES
 (1, 'Pique du Geek', '2009-03-14 08:00:00', 5),
 (2, 'Course de noel', '2009-12-25 14:30:00', 30),
 (3, 'je veux pas courir !', '2009-06-29 12:00:00', 11);

INSERT INTO `Participer` (`numSportif`, `numCourse`)
VALUES
 (1, 1),
 (4, 1),
 (5, 1),
 (8, 1),
 (9, 1);
```

La course Pique du Geek est complète (5/5), alors que se passe-t-il si nous réalisons une insertion supplémentaire ? Et bien rien, puisque le SGBD est incapable de comprendre le sens de la colonne "nbMaxParticipants". La solution la plus évidente est d'effectuer un contrôle au niveau applicatif, c'est-à-dire après avoir fait un SELECT, contrôler ceci puis effectuer ou non l'insertion.
Nous pourrions effectuer le contrôle côté applicatif; en quoi pourquoi est-ce une mauvaise pratique ?
Le rôle du SGBD étant en grande partie de contrôler l'intégrité, il n'assume pas ici entièrement son rôle. Il est gênant de ne pas pouvoir faire confiance à son SGBD en contrôlant nous-mêmes les valeurs qu'on lui envoie ; c'est en partie son rôle de traiter la validité de ces informations. Nous allons donc faire en sorte d'automatiser ce contrôle côté SGBD.

III - La solution : le trigger

III-A - Qu'est-ce qu'un trigger ?

Un trigger est un déclencheur. Autrement dit, c'est la possibilité d'associer un certain code à un évènement.
Cela fait beaucoup penser à la programmation évènementielle qui associe elle aussi du code à un évènement (onclick, onfocus...). C'est, en quelque sorte, la même chose, mis à part que les évènements ne sont pas directement liés aux actions de l'utilisateur, mais aux requêtes effectuées auprès du SGBD.

Liste des évènements possibles :

- INSERT
- UPDATE
- DELETE

Puisque ces évènements provoquent une modification dans le contenu de la base de données, il est possible d'agir avant (BEFORE) ou après (AFTER) ces modifications. Nous agissons souvent après pour ajouter des tuples qui

seront liés aux nouvelles données, alors que nous utiliserons généralement BEFORE dans un souci de contrôle de l'intégrité (annuler une "mauvaise donnée").

III-B - La syntaxe d'un trigger

Prenons l'exemple d'un trigger qui se déclenchera avant un insert sur la table *Participer* :

Syntaxe d'un trigger

```
CREATE TRIGGER `nomTrigger`
BEFORE INSERT
ON `Participer`
FOR EACH ROW
BEGIN
 -- Code à éxecuter;
END;
```

La clause FOR EACH ROW permet de préciser le type de trigger. En effet il existe plusieurs triggers, les triggers sur table et les triggers sur tuple. FOR EACH ROW précise qu'il s'agit d'un trigger sur tuple, c'est-à-dire que le code sera exécuté pour chaque tuple concerné par l'action.
Cela sous-entend que les préfixes OLD et NEW que nous utiliserons plus tard deviennent utilisables.
Pour le moment MySQL gère uniquement les triggers sur tuple.

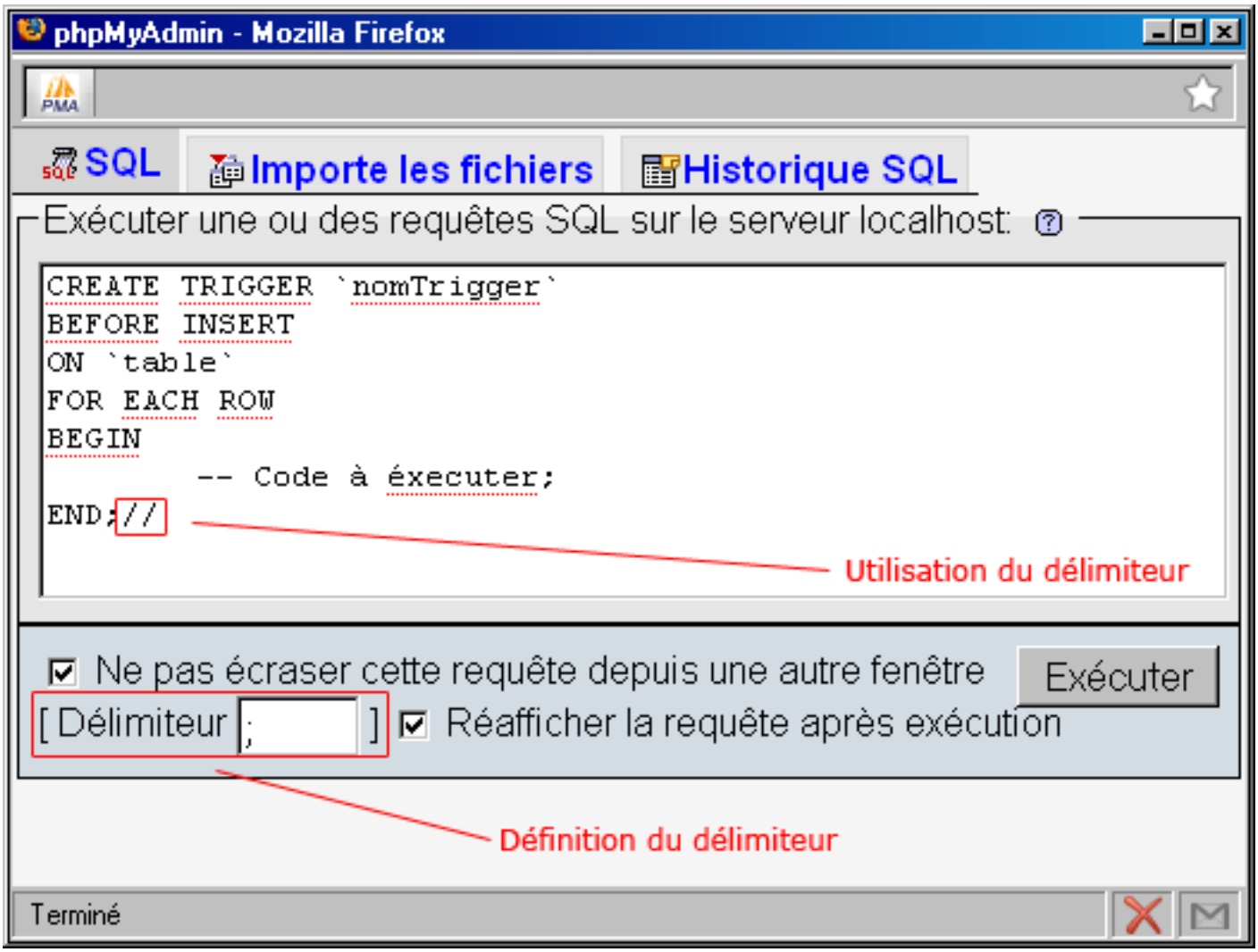
A propos du délimiteur
Le délimiteur est source de beaucoup d'erreurs. Il dépend du client, certains en ont besoin, d'autres pas. De plus ceux qui en ont besoin n'utilisent pas nécessairement la même syntaxe. Si la plupart des clients graphiques (Toad For MySQL, MySQL Query Browser) gèrent eux-mêmes le délimiteur, il n'en est pas de même pour PhpMyAdmin, et le client texte mysql (client en ligne de commande). Le client texte MySQL accepte la commande DELIMITER qui permet de modifier le délimiteur signalant la fin de la procédure, fonction, trigger, ou simple requête (par défaut, le délimiteur est ;).
Par exemple voici un exemple de trigger avec l'utilisation d'un délimiteur // :

Utilisation d'un délimiteur avec le client texte MySQL

```
DELIMITER //

CREATE TRIGGER `nomTrigger`
BEFORE INSERT
ON `table`
FOR EACH ROW
BEGIN
 -- Code à éxecuter;
END;//
```

PhpMyAdmin, quant à lui, offre un paramètre à remplir en bas de la fenêtre d'exécution de SQL :



Configuration du délimiteur sous phpMyAdmin

III-C - Une solution au problème

Nous allons maintenant créer un trigger qui permettra de contrôler l'insertion. En reprenant le problème précédemment posé, il nous serait utile de n'effectuer les insertions dans Participer que s'il reste de la place dans la course concernée.

Contrôle d'insertion de participation

```
CREATE TRIGGER `TR_INSERT_PARTICIPANT`
BEFORE INSERT
ON `Participer`
FOR EACH ROW
BEGIN
 DECLARE CURRENT_NB INTEGER;
 DECLARE MAX_NB INTEGER;

 SELECT COUNT(*) INTO CURRENT_NB FROM `Participer` WHERE numCourse = NEW.numCourse;
 SELECT `nbMaxParticipants` INTO MAX_NB FROM `Course` WHERE numCourse = NEW.numCourse;

 IF (CURRENT_NB >= MAX_NB) THEN
  SET NEW.numCourse = NULL;
  SET NEW.numSportif = NULL;
 END IF;
END;
//
```

Contrôle d'insertion de participation

Le fonctionnement de ce trigger est simple : avant chaque insertion, on vérifie s'il reste de la place et, si ce n'est pas le cas, on modifiera les informations saisies, provoquant ainsi un viol des contraintes d'intégrité, et un refus de l'insertion de la part du SGBD. Cela n'est pas la manière la plus élégante de faire, puisqu'il est impossible de savoir si l'échec d'insertion provient d'une mauvaise saisie, ou bien d'un manque de place. Nous allons peu à peu faire évoluer le système pour arriver à quelque chose de correct.

La gestion de ce type d'erreur spécifique au domaine de gestion est normalement assurée par l'utilisation d'exceptions personnalisées. Les exceptions sont disponibles sur MySQL, mais seul le système peut les propager, c'est-à-dire que le développeur ne peut déclencher de lui-même une exception.
Il n'est donc pas possible d'aborder le problème comme dans d'autres SGBD plus aboutis comme Oracle ou SQL-Server. Nous verrons plus tard ce que MySQL prévoit de mettre en oeuvre dans ses prochaines versions concernant cette gestion d'exceptions personnalisées.

III-D - Traiter tous les cas de figure

Si le SGBD est très performant dans sa gestion de l'intégrité, le développeur est soumis au risque de commettre des erreurs et d'oublier des cas possibles (c'est d'ailleurs ce que nous cherchons à éviter en reportant la gestion des erreurs sur le SGBD). Ici, nous avons prévu que l'on puisse ajouter des participations, mais rien n'empêche de modifier le nombre maximum de participations. Il faut donc prévoir et traiter ce cas.
Nous allons empêcher une diminution trop importante du nombre maximal de participations à une course.

Contrôle de modification des courses

```
CREATE TRIGGER `TR_UPDATE_COURSE`
BEFORE UPDATE
ON `Course`
FOR EACH ROW
BEGIN
 DECLARE CURRENT_NB INTEGER;
 DECLARE MAX_NB INTEGER;

 IF (NEW.nbMaxParticipants < OLD.nbMaxParticipants) THEN
  SELECT COUNT(*) INTO CURRENT_NB FROM `Participer` WHERE numCourse = OLD.numCourse;
  SET MAX_NB = NEW.nbMaxParticipants;

  IF (CURRENT_NB > MAX_NB) THEN
   SET NEW.nbMaxParticipants = NULL;
  END IF;
 END IF;
END;
//
```

Nous sommes maintenant certains que le nombre de participations réelles dans Participer sera en accord avec le nombre maximal de participations prévu par course.

IV - Comment générer un code erreur pour le récupérer côté applicatif par la suite ?

La solution précédemment décrite est incorrecte dans le sens où elle provoque volontairement une erreur qui n'a aucun rapport avec sa cause fonctionnelle. Nous allons voir comment faire pour utiliser la programmation stockée et les mêmes contrôles, mais de manière plus élégante.

IV-A - Utilisation de fonctions utilisateurs

Une fonction utilisateur, comme dans la plupart des langages procéduraux, est un processus recevant des paramètres et renvoyant une valeur. MySQL permet aussi les procédures stockées (qui sont comme leur nom l'indique, de simples procédures) mais elles ne nous seront pas utiles dans ce tutoriel. Le but est de créer et utiliser une interface entre le code applicatif et les données. Des fonctions feront l'intermédiaire et filtreront ce qui doit effectivement être contrôlé, et renverront au besoin un code d'erreur.
Reprenons le premier problème, l'ajout de participation :

Fonction d'ajout de participation

```
CREATE FUNCTION `NEW_PARTICIPATION` (P_numSportif INT, P_numCourse INT) RETURNS INT
BEGIN
 DECLARE CURRENT_NB INTEGER;
 DECLARE MAX_NB INTEGER;
 DECLARE C_ERROR INTEGER;

 SET C_ERROR = 0;
 SELECT COUNT(*) INTO CURRENT_NB FROM `Participer` WHERE numCourse = P_numCourse;
 SELECT `nbMaxParticipants` INTO MAX_NB FROM `Course` WHERE numCourse = P_numCourse;

 IF (CURRENT_NB < MAX_NB) THEN
  INSERT INTO `Participer` (`numSportif`, `numCourse`) VALUES (P_numSportif, P_numCourse);
 ELSE
  SET C_ERROR = 1001;
 END IF;

 RETURN C_ERROR;
END;
//
```

Il suffit d'appeler cette fonction pour toutes les insertions dans la table *Participer*, et de se servir de son code de retour pour évaluer le succès de l'insertion :

Utilisation de la fonction

```
SELECT NEW_PARTICIPATION(<numSportif>, <numCourse>) AS ERROR;
```

Si ERROR vaut 0, alors l'insertion s'est déroulée sans problème ; si ERROR vaut 1000, c'est qu'il y a déjà trop de participants.
Il faut traiter la mise à jour du maximum de participants de la même manière :

Fonction de modifications de participants maximum

```
CREATE FUNCTION `MAJ_MAX_PARTICIPATION` (P_numCourse INT, P_newMax INT) RETURNS INT
BEGIN
 DECLARE CURRENT_NB INTEGER;
 DECLARE MAX_NB INTEGER;
 DECLARE C_ERROR INTEGER;

 SET C_ERROR = 0;
 SELECT COUNT(*) INTO CURRENT_NB FROM `Participer` WHERE numCourse = P_numCourse;
 SET MAX_NB = P_newMax;

 IF (CURRENT_NB <= MAX_NB) THEN
  UPDATE `Course` SET nbMaxParticipants = MAX_NB WHERE numCourse = P_numCourse;
 ELSE
  SET C_ERROR = 1002;
 END IF;

 RETURN C_ERROR;
END;
//
```

Fonction de modifications de participants maximum

On note une différence au niveau du code : ici, on ne teste pas si l'ancienne valeur est supérieure à la nouvelle, autrement dit : on effectue le contrôle même s'il y a eu une augmentation du nombre maximal.
Nous avions antérieurement réalisé cette évaluation car les triggers le permettent rapidement (au travers des préfix OLD et NEW). Ici, obtenir l'ancienne valeur implique une sélection, un test qui devient trop lourd par rapport au gain qu'il apporte quelques rares fois.

Tout comme la fonction d'insertion de Participer, MAJ_MAX_PARTICIPATIONS revoie un code d'erreur.
Il suffit donc de passer par cette fonction pour mettre à jour le nombre maximum de participants :

Utilisation de la fonction

```
SELECT MAJ_MAX_PARTICIPATIONS(<numCourse>, <nbParticipationsMaximum>) AS ERROR;
```

Ici ERROR vaudra 1002 si le nouveau nombre maximum de participations est inférieur aux participations déjà en base de données.

IV-B - Exemple d'utilisation au niveau applicatif avec PHP

Nous allons donner un exemple d'utilisation de nos procédures au sein d'une application.

Utilisation des procédure coté applicatif

```
<?php
 function add_and_updateMax($idSportif, $idCourse, $newMax)
 {
  $errors = array();
  $connexion = mysql_connect(HOST, LOGIN, PASSWD);
  mysql_select_db(DB_NAME, $connexion);

  $rs = mysql_fetch_array(mysql_query("SELECT NEW_PARTICIPATION($idSportif, $idCourse) AS ERROR"));
  $rs2 = mysql_fetch_array(mysql_query("SELECT MAJ_MAX_PARTICIPATIONS($idCourse, $newMax) AS ERROR"));

  $errors[$rs[0]['ERROR']] = true; // Il y a eu une erreur à la première requête.
  $errors[$rs2[0]['ERROR']] = true; // Il y a eu une erreur à la seconde requête.

  foreach($errors as $currentKey => $currentValue)
  {
   switch($currentKey)
   {
    case 1001:
     $toReturn .= "Trop de participants sont déja présents dans cette course.<br/>";
     break;
    case 1002:

  $toReturn .= "La nouveau maximum de participant est inférieur au nombre de participations déja présentes.<br/>"
     break;
   }
  }
  return $toReturn;
 }
?>
```

Nous voyons ici que rajouter des codes erreurs peut se faire rapidement. Il suffit de générer ce code côté SGBD et de l'interpréter côté applicatif.

Il ne reste plus qu'à appeler cette fonction en PHP pour effectuer les modifications sur la base de données, sans même se préoccuper des erreurs :

Utilisation des procédure coté applicatif

```
<?php
 echo add_and_updateMax(10, 1, 2);
 // On ajoute une participation du sportif 10 à la course 1.
 // Puis en définit le maximum de participants de la course 1 à 2.
 // Les erreurs seront directement affichées.
?>
```

V - Les évolutions de la gestion d'erreurs

Nous sommes arrivés à construire un système assez fiable, qui évite certaines erreurs, et assez pratique à utiliser. Cependant, cette solution reste discutable, et peut être améliorée.

V-A - Limites de la démarche adoptée

Nous avons trouvé deux solutions à notre problème, la première avec des triggers. Son défaut est que l'on ne peut pas générer un code erreur spécifique en fonction de notre contexte métier.
La deuxième solution nous permet de connaître précisément d'où vient l'erreur au travers de codes erreurs spécifiques, mais rien n'oblige l'utilisateur à se servir de nos fonctions. Si on néglige ces fonctions, la base de données peut se trouver dans un état incohérent, ce qui doit être absolument évité. La solution n'est donc pas évidente, car même si l'on combine les deux, outre le fait que le coût en traitement s'avère très lourd pour la moindre insertion, nous ne garantissons pas de cibler l'erreur à tous les coups (dans le cas d'un insert sans NEW_PARTICIPATION par exemple).
Malheureusement, nous ne pouvons actuellement pas faire mieux avec MySQL, mais il existe dans le monde du SQL procédural des techniques qui permettent de gérer bien plus efficacement les erreurs. Nous allons les présenter, et comment MySQL prévoit de les implémenter.

V-B - Système d'exceptions

La meilleure solution à notre problème, qui, pour peu qu'elle soit implémentée dans MySQL nous aurait évité tous ces développements, est la gestion des exceptions personnalisées.
Les exceptions sont assimilées à des variables qui sont propagées au travers de notre application. Elles conduiront à un arrêt de l'exécution du programme, et un retour de code d'erreur spécialisé. Il suffit de déclarer une exception, puis de la réveiller quand nous en avons besoin. MySQL n'implémente pas encore ce système, mais de la **documentation** est disponible et nous allons exposer les possibilités qui devraient êtres offertes par des versions ultérieurs de MySQL. Nous nous inspirerons de cette documentation afin d'expliquer comment marcherons ces exceptions.

V-C - Le système d'exceptions de MySQL

Ici sera présentée la même gestion que nous avons simulée, mais avec les futures fonctionnalités de MySQL : l'utilisation de l'instruction **SIGNAL** suivant la norme **SQL:2003**. Nous déclarons une "condition" et un "handler" permettant d'associer un évènement à déclencher lors de la propagation de l'erreur.

Contrôle d'insertion de participation

```
CREATE TRIGGER `TR_INSERT_PARTICIPANT`
BEFORE INSERT
ON `Participer`
FOR EACH ROW
BEGIN
 DECLARE CURRENT_NB INTEGER;
 DECLARE MAX_NB INTEGER;
```

Contrôle d'insertion de participation

```
  DECLARE TROP_DE_PARTICIPANT CONDITION FOR -1001;
  DECLARE EXIT HANDLER FOR TROP_DE_PARTICIPANT SET @error = 'Trop de participants';

  SELECT COUNT(*) INTO CURRENT_NB FROM `Participer` WHERE numCourse = NEW.numCourse;
  SELECT `nbMaxParticipants` INTO MAX_NB FROM `Course` WHERE numCourse = NEW.numCourse;

  IF (CURRENT_NB >= MAX_NB) THEN
   SIGNAL TROP_DE_PARTICIPANT;
  END IF;
 END;
 //
```

Ici, la clause SIGNAL réveille la condition TROP_DE_PARTICIPANT associée à un HANDLER qui provoque l'action EXIT, c'est-à-dire l'arrêt de l'exécution.
Il sera possible de récupérer ce code d'erreur simplement avec mysql_errno(), à partir d'une application PHP par exemple.

Contrôle d'insertion de participation

```
<?php
// ...
mysql_query("INSERT INTO `Participer` (`numSportif`, `numCourse`) VALUES (<numSportif>, <numCourse>)");
echo mysql_errno(); // Retourne -1001 si il y a trop de participants.
// ...
?>
```

Attention, avant de choisir un code d'erreur, il faut s'assurer qu'il n'est pas déjà pris par le SGBD.
MySQL utilise les codes allant de 1000 à 2999 (tous ne sont pas utilisés mais il faut s'attendre à ce qu'ils puissent être pris dans les versions ultérieurs). Nous ne savons pas précisément comment le SGBD va évoluer et si les codes que nous allons choisir seront utilisés par la suite ou resterons libres, il est alors préférable de choisir un code négatif qui ne sera pas utilisé par MySQL.

VI - Conclusion

S'il est facile d'utiliser un SGBDR comme simple moyen de stocker de l'information, il est bien plus délicat d'en faire une utilisation robuste et intelligente. La programmation stockée est certainement l'un des meilleurs moyens de traiter les erreurs rapidement pour les confier au SGBD. Le code applicatif ne sert plus qu'à requêter le server et afficher les erreurs.
Même si MySQL est très jeune et gagnerait beaucoup à développer cette branche, il permet déjà de gérer correctement une base de données, avec un contrôle de l'intégrité satisfaisant. Nous avons donc tout intérêt à confier la gestion des règles d'intégrité au SGBD, nous diminuons ainsi les erreurs possibles, et gagnons beaucoup en rapidité et fiabilité de développement.

VII - Remerciements

Merci à **Antoun** et **ced** pour les relectures et les multiples suggestions.