Secteur Tertiaire Informatique
Filière « Etude et développement »

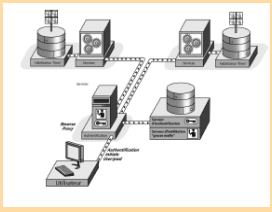
Base de données

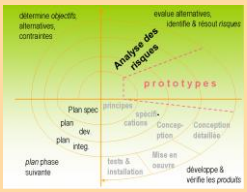
Triggers et procédures stockées

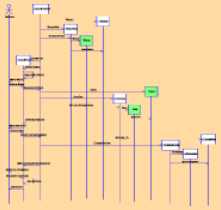
Apprentissage

Mise en pratique

Evaluation







Version	Date	Auteur(s)	Action(s)
1.0	25/03/19	Ludovic Domenici	Création du document

Objectifs

Écrire les règles métiers en langage SQL.

Pré requis

Créer et requêter une base de données.

Outils de développement

MySQL

Méthodologie

Mode d'emploi

Symboles utilisés :

 Renvoie à des supports de cours, des livres ou à la documentation en ligne constructeur.

 Propose des exercices ou des mises en situation pratiques.

 Point important qui mérite d'être souligné !

Ressources

Documentation de référence MySQL :

https://dev.mysql.com/doc/refman/8.0/en/trigger-syntax.html

https://dev.mysql.com/doc/refman/5.7/en/cursors.html

Lectures conseillées

1. INTRODUCTION

Utiliser les fonctions, triggers et procédures stockées MySQL présentent de nombreux avantages, mais aussi quelques difficultés. Nous allons détailler ces différents points, et voir comment contourner les quelques éléments qui pourraient poser problème (versionning, tests unitaires…).

1.1 AVANTAGES A L'UTILISATION DES FONCTIONS, TRIGGERS ET PROCEDURES STOCKEES MYSQL

Si vous êtes dans le cadre d'une application web en mode LAMP, avec du code PHP et une base de données MySQL, deux choix s'offrent à vous au moment d'implémenter vos règles métier. Soit créer des méthodes PHP dans vos classes, ou utiliser des fonctions ou procédures en MySQL. La question du choix à faire peut légitimement se poser, chacune des méthodes ayant ses avantages et inconvénients.

Développer côté base de données présentent plusieurs avantages :

- Si vous êtes dans un environnement mixte (serveurs LAMP et serveurs Apache-Tomcat), vous ne pourrez pas utiliser dans Tomcat par exemple les développements que vous avez fait en PHP. Alors que vous pourrez exécuter et récupérer sans problème le jeu de données que vous renverra votre procédure stockée. Cela vous évitera donc de devoir coder deux fois la même chose, en plus dans deux environnements différents (avec tous les problèmes que cela peut vous apporter).
- Cela vous apporte un plus en terme de sécurité, car vos données ne sont plus modifiées en direct par un code externe. Elles passent par votre couche de développement SQL, où vous pouvez définir vos différents contrôles. Le trigger dans le cadre de la sécurité de vos données est également primordial, car c'est le seul qui vous permet des contrôles qui ne pourront jamais être outrepassés.
- Les règles métiers sont au plus proche de la donnée. Cela évite donc pour vos traitements de traverser à chaque fois toutes les couches (navigateur côté client, serveur web, interpréteur PHP et base de données). Vous êtes déjà au niveau le plus bas.

Vous êtes donc proche de la donnée, vos développements font foi pour garantir l'intégrité de vos données, et ils sont également réutilisables sur différentes applications et sur différents environnements.

1.2 INCONVENIENTS ET SOLUTIONS DE CONTOURNEMENT

Vous venez de voir que les avantages étaient nombreux et non négligeables. En revanche, cela ne va pas sans poser certains soucis. Mais une fois ces problématiques intégrées, il devient assez simple de proposer des solutions pour y faire face.

1.2.1 Versionning du code

C'est un souci souvent cité lors de développement côté MySQL. Le code des fonctions, triggers et procédures stockées MySQL n'est pas versionnable.

Autre problème, lors d'une recherche complète dans l'application avec un IDE comme Eclipse, par exemple dans le cadre d'une refactorisation de code impliquant un changement de nom dans un champ, il ne verra pas le code MySQL.

La solution à ce problème est assez simple : il faut intégrer le code SQL à l'application, et mettre en place un mécanisme pour charger le code dans la base de données.

Voyons cela plus en détail. Si vous avez une fonction, un trigger ou une procédure stockées MySQL à créer, vous allez créer le code dans votre IDE. L'idéal est sans doute de dédier un dossier de votre application (un dossier nommé SQL par exemple), avec des sous-niveaux permettant de les classer par type (fonction, trigger, vue, procédure) et avec une arborescence correspondant à votre application. Les fichiers créés seront d'extension `.sql` (vous bénéficierez en plus de la coloration syntaxique, voire des corrections de votre IDE qui sait en général reconnaître le langage SQL).

Vous procéderez ainsi pour tout votre code SQL, vous ajouterez au fur et à mesure votre code, en ajoutant le `DROP` et le `CREATE`. Il vous faudra définir également un `DELIMITER` particulier (`$$` par exemple, ou tout autre que vous serez certain de ne pas rencontrer dans votre code SQL) que vous ajouterez après l'instruction `DROP` et à la toute fin de votre trigger, fonction ou procédure stockée MySQL.

Comme il s'agit de fichier, vous pourrez tout à fait les gérer dans votre outil de gestion de version du code, par exemple avec Git et SourceTree :

Reste maintenant à charger le code dans la base de données.

Pour cela il vous suffit de mettre en place un script qui récupère vos fichiers `.sql` (par exemple avec un `RecursiveDirectoryIterator`[1]), les agrège dans un fichier temporaire (via un `file_put_contents`[2] par exemple) et joue ensuite ce fichier sur votre base (par l'appel de MySQL en ligne de commande).

Si vous voulez aller un peu plus loin, vous pouvez ajouter des arguments à votre script, pour par exemple ne traiter que les fonctions, ou les procédures, ou les triggers. Ou bien encore pour ne sortir qu'un fichier SQL d'agrégation sans le lancer sur la base (pour contrôle).

Si vous réalisez vos passages en production via un dépôt de version, rien n'empêche également de mettre en place un script de mise en production qui irait mettre à jour la production avec le dépôt de votre projet, et qui jouerait ensuite la mise en place de vos fichiers SQL.

1.2.2 Tests unitaires

Un autre reproche possible au fait d'utiliser des triggers, fonctions et procédures stockées MySQL est l'impossibilité de les intégrer dans des systèmes de tests unitaires automatisés qui peuvent être faits par exemple avec Jenkins dans un processus d'intégration continue.

C'est vrai. En revanche si vous développez une fonction MySQL par exemple, il y a de fortes chances pour qu'elle soit utilisée ensuite dans une méthode Java qui se chargera de l'appeler en lui passant des paramètres.

Hors, cette méthode java peut tout à fait être appelée elle dans un test unitaire.

---

[1] La classe PHP `RecursiveDirectoryIterator` fournit un moyen d'itérer récursivement sur des dossiers d'un système de fichiers. L'équivalent en java n'existe pas. Il faut créer une méthode en utilisant le pacquetage `java.File.*`.

[2] La méthode PHP `file_put_contents` écrit un contenu dans un fichier. En java cette méthode est à créer en utilisant les classes `File` et `FileWriter`, toutes deux de `java.io`.

Prenons l'exemple d'une procédure stockée qui met à jour les tarifs d'un article. Vous l'avez développé en MySQL car elle peut être appelée par différents outils, et vous vouliez être sûr d'avoir un point de passage unique pour la modification de prix. Cette procédure peut être appelé par la méthode `majPrix` de votre classe article java, que vous pourrez tester unitairement.

Enfin, si l'on parle de tests unitaires (hors intégration continue là), il n'y a rien de plus simple que de tester une procédure stockée MySQL. Il suffit de l'appeler en ligne de commande en lui passant les bons paramètres. C'est même encore moins contraignant que de tester une méthode java. Rien n'empêche même de développer un script SQL permettant de tester avec différents paramètres l'appel d'une fonction ou d'une procédure stockée MySQL.

2. LES TRIGGERS

Tout d'abord nous allons commencer par définir ce qu'est un Trigger dans MySQL. Un Trigger est un mécanisme permettant d'indiquer à la base de données qu'elle doit exécuter une action avant ou après avoir reçu un évènement (CRUD sur une des tables). Par exemple, il est possible d'indiquer à MySQL d'exécuter une fonction avant de réaliser un `INSERT` ou un `UPDATE` dans une table.

Il faut aussi savoir qu'un Trigger exécute un traitement pour chaque ligne insérée, supprimée ou modifiée. Ainsi si l'on traite dix lignes à la fois, le traitement sera effectué dix fois. De plus, le traitement ne peut être exécuté sur la même table.

2.1 CREATION D'UN TRIGGER :

Dans sa forme la plus basique un trigger se crée comme suit :

```
CREATE TRIGGER nom_du_trigger
  [Moment du déclenchement] [table cible] [type d'accès à la base]
  BEGIN
    [Instructions]
  END;
```

Par exemple :

```
/* Création de la table */
CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));

/* Création du trigger */
CREATE TRIGGER before_account_insert
  BEFORE INSERT ON account
  FOR EACH ROW SET @sum = @sum + NEW.amount;
```

L'instruction `CREATE TRIGGER` crée un déclencheur nommé `before_account_insert` associé à la table `account`. Il inclut également des clauses qui spécifient le temps d'action du trigger, l'événement déclencheur et l'action à effectuer lorsque le trigger est activé:

L'instruction qui suit `FOR EACH ROW` définit le corps du trigger; c'est-à-dire l'instruction à exécuter chaque fois que le trigger est activé, ce qui se produit une fois pour chaque ligne affectée par l'événement déclencheur. Dans l'exemple, le corps du déclencheur est un simple `SET` qui accumule dans une variable utilisateur les valeurs insérées dans la colonne `amount`. L'instruction fait référence à la colonne en tant que `NEW.amount`, ce qui *signifie la valeur de la colonne amount à insérer dans la nouvelle ligne*.

`NEW.amount` correspond à la nouvelle valeur de `amount`. Par opposition `OLD.amount` correspondrait à la valeur avant sa modification. Donc `:NEW` et `:OLD` représentent dans les triggers `update for each row` la ligne avant et après modification.

Pour utiliser le déclencheur, définissez la variable d'accumulateur (`sum`) sur zéro, exécutez une instruction `INSERT`, puis voyez quelle valeur la variable a par la suite:

```
SET @sum = 0;
INSERT INTO account VALUES(137, 14.98),
  (141, 1937.50),
  (97, -100.00);
```

```
SELECT @sum AS 'Total amount inserted';
+-----------------------+
| Total amount inserted |
+-----------------------+
|               1852.48 |
+-----------------------+
```

En utilisant la construction `BEGIN ... END`, vous pouvez définir un trigger qui exécute plusieurs instructions. Dans le bloc `BEGIN`, vous pouvez également utiliser une autre syntaxe autorisée dans les routines stockées telles que les conditions et les boucles. Cependant, comme pour les routines stockées, si vous utilisez le programme MySQL pour définir un déclencheur qui exécute plusieurs instructions, il est nécessaire de redéfinir le délimiteur d'instruction MySQL afin que vous puissiez utiliser le délimiteur d'instruction "`;`" dans la définition du déclencheur.

```
DELIMITER $$

CREATE TRIGGER trigger_utilisateur
BEFORE INSERT ON utilisateur
FOR EACH ROW
BEGIN
    [Actions à réaliser];
END $$

DELIMITER ;
```

L'exemple suivant illustre ces points. Il définit un déclencheur `UPDATE` qui vérifie la nouvelle valeur à utiliser pour la mise à jour de chaque ligne et modifie la valeur afin qu'elle soit comprise entre 0 et 100. Il doit s'agir d'un déclencheur `BEFORE` car la valeur doit être vérifiée avant d'être utilisée pour la mise à jour :

```
mysql> delimiter //
mysql> CREATE TRIGGER before_account_upd
       BEFORE UPDATE ON account
       FOR EACH ROW
       BEGIN
           IF NEW.amount < 0 THEN
               SET NEW.amount = 0;
           ELSEIF NEW.amount > 100 THEN
               SET NEW.amount = 100;
           END IF;
       END;//
mysql> delimiter ;
```

Nous allons maintenant prendre un exemple concret.
Si vous souhaitez savoir quand de nouveaux utilisateurs ont été insérés dans la table `utilisateur` il vous faudra créer un Trigger qui insérera une date dans une table différente. Ainsi, il faut commencer par créer une seconde table qui servira à lister les dates d'insertion des utilisateurs.

```
CREATE TABLE date_utilisateur(
```

```
  id Int(20) Auto_Increment PRIMARY KEY,
  date_insertion  DATE
)ENGINE = InnoDB DEFAULT CHARSET = utf8;
```

Ensuite, adaptons notre déclencheur :

```
DELIMITER $$

CREATE TRIGGER trigger_utilisateur
  BEFORE INSERT ON utilisateur

  FOR EACH ROW
    BEGIN
      INSERT INTO date_utilisateur VALUES("", NOW());
    END $$

DELIMITER ;
```

Nous avons maintenant un Trigger qui va insérer une ligne dans la table `date_utilisateur` pour chaque ligne insérée dans la table utilisateur.

Voici le résultat sous MySQL :

```
mysql> INSERT INTO utilisateur VALUES("","jb_admin","password","Jérémy","BEDDES");
Query OK, 1 row affected, 1 warning (0.01 sec)

mysql> SELECT * FROM date_utilisateur;
+----+----------------+
| id | date_insertion |
+----+----------------+
|  1 | 2015-04-22     |
+----+----------------+
1 row in set (0.00 sec)
```

Les Triggers sont très intéressants dans le contexte de validation ou de Workflow d'une application.

Cependant il est important de garder à l'esprit que ces Triggers sont consommateurs en ressources : ils capturent des évènements et traitent des actions supplémentaires. Ces différentes étapes ralentissent tout autant le fonctionnement de la base de données. Lorsque vous pouvez vous passer des Triggers, passez-vous en. Une action peut-être aisément déportée sur la partie métier, et non sur la partie base de données dans de nombreux cas.

2.1.1 Syntaxe

- Le nom du trigger est défini après l'instruction `CREATE TRIGGER`. Le nom du trigger doit respecter la convention de dénomination [moment d'activation du trigger]_[nom de la table]_[événement du trigger], par exemple, `before_employees_update`.
- Le moment d'activation du trigger peut être `BEFORE` (avant) ou `AFTER` (après). Vous devez spécifier le moment d'activation lorsque vous définissez un trigger. Vous utilisez le mot clé **`BEFORE`** si vous souhaitez traiter une action avant que la modification ne soit apportée à la table et **`AFTER`** si vous devez la traiter après la modification.

- L'événement déclencheur peut être **INSERT**, **UPDATE** ou **DELETE**. Cet événement provoque l'appel du trigger. Un trigger ne peut être invoqué que par un seul événement. Pour définir un trigger appelé par plusieurs événements, vous devez définir plusieurs triggers, un pour chaque événement.
- Un trigger doit être associé à une table spécifique. Sans la définition de la table, il n'y aurait pas de trigger. Vous devez donc spécifier le nom de la table après le mot clé `ON`.
- Vous placez les instructions SQL entre les blocs `BEGIN` et `END`. C'est ici que vous définissez la logique du trigger.

2.1.2 Le 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 matable

  FOR EACH ROW
    BEGIN
     -- Code à exécuter;
    END;//
```

2.1.3 Voir les triggers

Pour connaître les différents triggers d'une base de données la commande est :

```
Show triggers;
```

2.1.4 Activer / désactiver un trigger

Il peut être utile et même conseillé suivant le contexte de désactiver un Trigger lors d'une opération de maintenance lourde ou d'administration sur une table (PL/SQL, SQL*Loader, mise à jour en masse, etc...).

- Avec **`ALTER TRIGGER`**
- Avec **`ALTER TABLE`**
- Avec **`DROP TRIGGER`**

```
ALTER TRIGGER MonTrigger ENABLE;
```

```
ALTER TRIGGER MonTrigger DISABLE;
```

2.1.4.1 Activer / Désactiver les triggers d'une table

```
ALTER TABLE MaTable ENABLE ALL TRIGGERS;
ALTER TABLE MaTable DISABLE ALL TRIGGERS;
```

2.1.5 Supprimer un trigger

Seul le propriétaire de la table peut supprimer un trigger.

La commande est

```
DROP TRIGGER [nom du trigger]
```

Pour PostgreSQL, il faut rajouter les instructions suivantes :

```
ON [table]
[CASCADE | RESTRICT]
```

2.1.5.1 Paramètres

- **nom** : Le nom du déclencheur à supprimer.
- **table** : Le nom de la table (éventuellement qualifié du nom du schéma) sur laquelle le déclencheur est défini.
- **CASCADE** : Les objets qui dépendent du déclencheur sont automatiquement supprimés.
- **RESTRICT** : Le déclencheur n'est pas supprimé si un objet en dépend. Comportement par défaut.

2.1.5.2 Exemples

Destruction du déclencheur `si_dist_existe` de la table `films` :

```
DROP TRIGGER si_dist_existe ON films;
```

2.2 EXEMPLES

Les exemples qui suivent fonctionnent sur MySQL.

Prenons comme exemple celui des messages et topics de forum : nous voulons qu'à chaque nouveau message, le champ `message_at` du topic soit mis à jour à la date courante.

2.2.1 Création des tables

```
CREATE TABLE topic (
  id_topic INT NOT NULL,
  description VARCHAR(255),
  message_at DATETIME);

CREATE TABLE message (
  id_message INT NOT NULL AUTO_INCREMENT,
  id_topic INT,
  msg TEXT,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP);
```

2.2.2 … et des déclencheurs

Dans le trigger, on accède aux données de l'enregistrement à insérer avec le préfixe `NEW`

```
DELIMITER |
  CREATE TRIGGER after_message_insert
  AFTER INSERT ON message

  FOR EACH ROW
    BEGIN
      INSERT INTO topic VALUES (NEW.id_topic, NEW.msg, NOW());
    END;
|
```

Le `FOR EACH ROW` est obligatoire même si dans notre cas nous ne traitons qu'un seul enregistrement.

Supposons que nous voulions supprimer un message, la date du topic doit être mise à jour avec :

- Soit la date de l'avant-dernier message s'il existe
- Soit `null` s'il n'existe pas

```
DELIMITER |
CREATE TRIGGER delMsg BEFORE DELETE ON message
  FOR EACH ROW
    BEGIN
    DECLARE date_topic DATETIME;
    DECLARE date_av_dernier DATETIME;
    DECLARE id_av_dernier INT DEFAULT 0;

    /* On modifie le topic si seulement le message supprime est
le dernier */
    SELECT message_at INTO date_topic FROM topic WHERE id_topic
= OLD.id_topic;
    IF date_topic <= OLD.created_at THEN
      BEGIN
        /* On recherche l'avant dernier message du même topic*/
        SELECT   created_at,   id_topic   INTO   date_av_dernier,
id_av_dernier
          FROM message
          WHERE  id_message  <>  OLD.id_message  AND  id_topic  =
OLD.id_topic ORDER BY Created_at DESC LIMIT 1;

        /* On met a jour la date dans le topic ou NULL si pas de
message antérieur */
        IF id_av_dernier = 0 THEN
```

```
          UPDATE topic SET message_at = NULL WHERE id_topic =
OLD.id_topic;
        ELSE
          UPDATE topic SET message_at = date_av_dernier WHERE
id_topic = OLD.id_topic;
        END IF;
      END;
    END IF;
  END;
|
```

Autre solution en plaçant le trigger après la suppression.

Dans ce cas le message est déjà supprimé, on recherche donc la date du dernier message s'il existe.

```
DELIMITER |
CREATE TRIGGER delMsg AFTER DELETE ON message
  FOR EACH ROW
    BEGIN
      DECLARE date_topic DATETIME;
      DECLARE date_dernier DATETIME;
      DECLARE id_dernier INT DEFAULT 0;

      /* On modifie le topic si seulement le message supprime
était le dernier */
      SELECT message_at INTO date_topic FROM topic WHERE
id_topic = OLD.id_topic;
      IF date_topic <= OLD.created_at THEN
        BEGIN
          /* On recherche le dernier message existant du même
topic*/
          SELECT created_at, id_topic INTO date_dernier,
id_dernier
            FROM message
            WHERE id_topic = OLD.id_topic ORDER BY Created_at
DESC LIMIT 1;

          /* On met a jour la date dans le topic ou NULL si pas
de message antérieur */
          IF id_dernier = 0 THEN
            UPDATE topic SET message_at = NULL WHERE id_topic =
OLD.id_topic;
          ELSE
            UPDATE topic SET message_at = date_av_dernier WHERE
id_topic = OLD.id_topic;
          END IF;
        END;
      END IF;
    END;
|
```

2.2.3 Utilisation de déclencheurs pour gérer un compteur de messages

On suppose que la table `topic` contient le nombre de messages. Pour cela on rajoute un `champ nb_messages` initialisé à zéro.

```
CREATE TABLE topic (
  id_topic INT NOT NULL AUTO_INCREMENT,
  description VARCHAR(255),
  message_at DATETIME,
  nb_messages INT DEFAULT '0');
```

On crée un trigger sur l'ajout de message et un autre sur la suppression

```
CREATE TRIGGER AFTER_MESSAGE_INSERT AFTER INSERT ON message
FOR EACH ROW
  INSERT INTO topic (nb_messages) VALUES (NEW.nb_messages + 1);
END;
```

On crée un trigger sur la suppression de message

```
CREATE TRIGGER AFTER_MESSAGE_DELETE AFTER DELETE ON message
FOR EACH ROW
  DELETE FROM topic
  WHERE id_topic = OLD.id_topic;
END;
```

2.2.4 Comportement avec les auto-incréments

Supposons que lorsque l'on crée un topic, on crée automatiquement un premier message qui reprend la description du topic.

```
DELIMITER |
CREATE TRIGGER instopic AFTER INSERT ON topic
FOR EACH ROW
  INSERT INTO message (id_topic, msg)
  VALUES (NEW.id_topic, NEW.description);
|
```

2.2.5 Suppression en cascade

Lorsque l'on crée une liaison entre deux tables, on a la possibilité de définir une suppression en cascade.

Cela signifie que si on supprime un topic, tous ses messages sont supprimés dans la foulée (sans avoir besoin de trigger).

Le problème c'est que la suppression dans la table mère désactive le trigger DELETE de la table fille !!!

Il faut choisir l'un ou l'autre.

3. LES PROCEDURES STOCKEES

Une procédure stockée est un petit programme stocké dans la base de données et appelable à partir d'un client comme on peut le faire pour une requête. Une procédure est exécutée par le serveur de base de données.

3.1 MODE SQL

Lorsque l'on crée une procédure, un des premiers soucis à contourner et tout bête : comment puis-je écrire une instruction qui se termine par un point-virgule alors que ma procédure contient un point-virgule à chaque ligne ?

La réponse est : changeons le caractère **délimiteur** par un autre qui ne risque pas d'apparaitre dans nos instructions, prenons par exemple le pipe '**|**'

```
DELIMITER |
```

Créons notre procédure avec la commande `CREATE PROCEDURE name ([param[,param ...]])`

On peut utiliser des paramètres `IN`, `OUT` ou `INOUT` c'est à dire en Entrée, en Sortie et à la fois en Entrée et en Sortie.

Chaque paramètre est défini par son sens, son nom et son type.

Voici un premier exemple d'une procédure qui met à jour les prix en leur appliquant un coefficient :

```
CREATE PROCEDURE maj_prix (IN coef FLOAT)
      BEGIN
          UPDATE T1 SET PRICE = PRICE * coef;
      END
      |
```

Notez le `|` final qui nous fait sortir de l'édition.

Chaque bloc d'instructions doit être encadré par `BEGIN` et `END`, mais peut être ignoré dans le cas d'une seule instruction.

Pour lancer notre procédure, on pense à remettre le délimiteur standard, c'est à dire le point-virgule.

On lance une procédure par une commande `call` en SQL

```
DELIMITER ;
CALL maj_prix(1.05);
```

3.1.1 Paramètre de retour

Pour affecter un résultat SQL à une variable, on utilise le mot clé **`INTO`**

```
DELIMITER |
CREATE PROCEDURE get_count (OUT nb INT)
    BEGIN
        SELECT COUNT(*) INTO nb FROM T1;
    END
    |
```

Pour récupérer le résultat, on doit initialiser la variable qui va recevoir le résultat

```
SET @n = 0;
CALL get_count(@n);
SELECT @n;
```

Pour modifier une procédure, il faut la supprimer avec `DROP` puis la recréer, il n'existe pas de commande `ALTER PROCEDURE`

```
DROP PROCEDURE maproc;
```

Pour afficher le code d'une procédure

```
SHOW CREATE PROCEDURE maproc;
```

Pour voir les procédures existantes

```
SHOW PROCEDURE STATUS LIKE '%%'
```

3.1.2 Déclaration des variables

Les variables sont déclarées par le mot clé `DECLARE` et les types sont les types SQL de MySQL

On peut déclarer plusieurs variables du même type sur la même ligne.

On peut fixer une valeur d'initialisation.

```
DECLARE myvar CHAR(10);
DECLARE i, j INT DEFAULT 0;
```

Commentaires

```
/*
* une ligne
* deuxième ligne
*/
-- une seule ligne (il y a un espace après les 2 tirets)
```

3.1.3 Affectation directe de variables

On utilise le mot clé `SET`

```
DECLARE x INT;
DECLARE nom VARCHAR(50);
SET x = 10;
SET nom = 'toto';
```

3.1.4 Retourner des enregistrements

Si une procédure exécute une requête `SELECT`, les enregistrements résultant sont retournés.

```
DELIMITER |
CREATE PROCEDURE mesenreg()
  SELECT * FROM matable;
  |
```

Cependant il n'est pas possible d'appeler une procédure à partir d'une autre requête.

3.2 Mode Java

L'interface `CallableStatement`, qui étend `PreparedStatement`, permet de faire appel aux procédures stockées et aux fonctions de manière standard pour tous les SGBD. Pour que cet appel soit indépendant du SGBD ciblé, `CallableStatement` utilise la syntaxe d'échappement.

La principale différence avec les `PreparedStatement` se situe au niveau des paramètres. Ceux-ci sont toujours définis par des points d'interrogation, mais en plus des paramètres d'entrée (`IN`), `CallableStatement` peut avoir des paramètres de sortie (`OUT`). Ces paramètres définissent le résultat de la procédure. On peut aussi combiner ces deux types (`INOUT`).

Une instance de `CallableStatement` s'obtient grâce aux méthodes `prepareCall` de `Connection`. Le premier argument de ces méthodes est une chaîne de caractères définissant l'instruction SQL. Cette chaîne de caractères utilise la syntaxe d'échappement et peut avoir deux formes.

Pour les procédures stockées :

```
// [(?, ?, ...)] sont les éventuels arguments de la procédure
String sql = "{call nomDeLaProcedure[(?, ?, ...)]}";

//ces arguments peuvent être de type IN, OUT ou INOUT
CallableStatement statement = connection.prepareCall(sql);
```

Pour les fonctions (procédures stockées renvoyant un résultat) :

```
// Le premier ? est le résultat de la procédure
String sql = "{? = call nomDeLaProcedure[(?, ?, ...)]}";

// [(?, ?, ...)] sont les éventuels arguments de la procédure
CallableStatement statement = connection.prepareCall(sql);
```

Les autres arguments de la méthode `prepareCall` servent à déterminer les types de `ResultSet` obtenus à partir de la procédure.

Par exemple :

```
String sql = "{? = call max(?, ?)}";
CallableStatement statement = connection.prepareCall(sql,
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_READ_ONLY);
```

3.2.1 Passer des paramètres à la procédure

Comme pour les `PreparedSatement`, le passage des paramètres d'entrée des `CallableStatement` se fait grâce à l'ensemble des méthodes `setXXX`. Il est important de connaître les correspondances entre les types SQL et les types java. En plus de l'index, on peut cibler un paramètre grâce à son nom.

3.2.1.1 Exemple simple

```
String url = "jdbc:oracle:thin:@localhost:1521/info";
String driver = "oracle.jdbc.driver.OracleDriver";
String login = "root";
String mdp = "root";
Connection con = null;

try {
  // Chargement du driver
  Class.forName(driver);

  con = DriverManager.getConnection(url, login, mdp);
  con.setAutoCommit(false);

  String sql = "{call maproc(?, ?, ?, ?, ?, ?, ?, ?, ?, ?)}";
  CallableStatement call = con.prepareCall(sql);

  call.setString(1, revnom.getText());
  call.setString(2, revadr.getText());
  call.setString(3, revcp.getText());
  call.setString(4, revville.getText());
  call.setString(5, revtel.getText());
  call.setString(6, revfax.getText());
  call.setString(7, revcontactnom.getText());
  call.setString(8, revcontactprenom.getText());
  call.setString(9, revcontacttel.getText());
  call.setString(10, revcontactfax.getText());

  call.execute();
  call.commit();
}
catch(SQLException e1){
  System.out.println("SQLException : " + e1);
  e1.printStackTrace();
  con.rollback();
}
catch(Exception e2){
  System.out.println("Exception : " + e2);
  e2.printStackTrace();
  con.rollback();
}
finally(){
  con.close();
}
```

3.2.1.2 Exemple complet

Voici un exemple plus fourni, pour une procédure stockée chargée de retrouver les abonnés d'un annuaire ayant un certain nom. Cette procédure renvoie plusieurs `ResultSet`.

Le code Java chargé de la créer et de l'appeler :

```
// CHARGEMENT DU DRIVER
Class.forName("com.mysql.jdbc.Driver");

// CONNECTION AU SGBD
String url = "jdbc:mysql://localhost/test";
String user = "user";
String password = "root";
connection = DriverManager.getConnection(url,user,password);

// CREATION DE LA PROCEDURE
Statement statement = connection.createStatement();

// D'abord on la supprime si elle existe
statement.executeUpdate (
    "DROP PROCEDURE IF EXISTS rechercherNom");

// Ensuite on la créée
statement.executeUpdate(
    "CREATE PROCEDURE rechercherNom(IN leNom VARCHAR(50))\n"
    + "BEGIN\n"
    + " SELECT * FROM Annuaire WHERE nom = leNom;\n"
    + " SELECT COUNT(*) FROM Annuaire WHERE nom = leNom;\n"
    + "END\n");

// APPEL DE LA PROCEDURE
String sql = "{call rechercherNom(?)}";
CallableStatement call = connection.prepareCall(sql);

// Passage de la chaîne "ioio" comme valeur du premier paramètre
call.setString(1, "ioio");
if(call.execute()){
    // Récupération des ResultSet
    ResultSet rs1 = call.getResultSet();
    call.getMoreResults(Statement.KEEP_CURRENT_RESULT);
    ResultSet rs2 = call.getResultSet();

    // Traitement des informations
    while(rs1.next()){
        for(int i = 0;
          i < rs1.getMetaData().getColumnCount();
          i++){
            System.out.print(rs1.getObject(i + 1) + ", ");
        }
        System.out.println("");
    }
    rs2.next();
    StringBuilder sb = new StringBuilder();
    sb.append("Nombre de lignes = ")
      .append(rs2.getObject(1)
    System.out.println(sb.toString());
```

```
    resultat1.close();
    resultat2.close();
}
```

Comme pour les `PreparedStatement`, les paramètres `IN` peuvent être identifiés par leur index dans l'instruction SQL ou par leur nom. Pour cet exemple, les instructions ci-dessous sont équivalentes :

```
call.setString("leNom", "ioio");
call.setObject(1, "ioio");
call.setObject(1, "ioio", Types.VARCHAR);
call.setObject("leNom", "ioio", Types.VARCHAR);
```

3.2.2 Récupérer le résultat d'une procédure stockée

Pour récupérer les résultats (`OUT`) d'un `CallableStatement`, il faut utiliser les méthodes `getXXX` correspondant au type retourné. Les paramètres `OUT` doivent être enregistrés comme tels. Voici un exemple récupérant le nombre d'abonnés d'un annuaire.

```
/* Code SQL */
CREATE PROCEDURE nombreAbonnes(OUT nb INTEGER)
   BEGIN
      SELECT COUNT(*) INTO nb FROM Annuaire;
   END
```

En java, comme pour les paramètres d'entrée, un paramètre peut être ciblé par son nom ou par son index.

```
String sql = "{call nombreAbonnes(?)}";
CallableStatement statement = connection.prepareCall(sql);

// Enregistrement du paramètre de sortie en fonction de son type
// et de son nom
statement.registerOutParameter("nb", java.sql.Types.INTEGER);

// Enregistrement du paramètre de sortie en fonction de son type
// et de son index
// statement.registerOutParameter(1, java.sql.Types.INTEGER);
statement.execute();

// Récupération du résultat en fonction de l'index
int resultat = statement.getInt(1);

//Récupération du résultat en fonction du nom du paramètre

// int resultat = statement.getInt("nb");
StringBuilder sb = new StringBuilder();
sb.append("Nombre d'abonnés = ")
  .append(resultat)
System.out.println(sb.toString());
```

3.2.3 Paramètre d'entrée / sortie (INOUT)

Pour spécifier un paramètre `INOUT`, il suffit de combiner `registerOutParameter` et `setXXX`. Voici un exemple qui montre la marche à suivre.

```
StringBuilder sb = new StringBuilder();
Connection connection = ... ;

// CREATION DE LA PROCEDURE
Statement stmt = connection.createStatement();
stmt.executeUpdate("DROP PROCEDURE IF EXISTS helloProc");

sb.append("CREATE PROCEDURE helloProc ")
  .append("(INOUT param VARCHAR(30))\n")
  .append("BEGIN\n")
  .append(" SELECT CONCAT('Hello', param, 'avec JDBC !!!')")
  .append(" INTO param;\n")
  .append("END\n");
stmt.executeUpdate(sb.toString());

// APPEL DE LA PROCEDURE
String sql = "{call hello(?)}";

CallableStatement call = connection.prepareCall(sql);

// Passage de la valeur du paramètre
call.setString(1, " world ");

// Enregistrement du paramètre en tant que paramètre OUT
call.registerOutParameter(1, Types.VARCHAR);

// Exécution et récupération du résultat
call.execute();
System.out.println(call.getString(1));
```

Et voici à quoi devrait ressembler le résultat :

```
"Hello world avec JDBC !!!"
```

3.2.3.1 Exemple IN, OUT et INOUT

```
DELIMITER $$

DROP PROCEDURE IF EXISTS sp_nested_loop$$
CREATE PROCEDURE sp_nested_loop(IN i INT, IN j INT, OUT x INT,
OUT y INT, INOUT z INT)
BEGIN
    DECLARE a INTEGER DEFAULT 0;
    DECLARE b INTEGER DEFAULT 0;
    DECLARE c INTEGER DEFAULT 0;
    WHILE a < i DO
        WHILE b < j DO
```

```
            SET c = c + 1;
            SET b = b + 1;
        END WHILE;
        SET a = a + 1;
        SET b = 0;
    END WHILE;
    SET x = a, y = c;
    SET z = x + y + z;
END $$
DELIMITER ;
```

Invocation de la procédure :

```
SET @z = 30;
call sp_nested_loop(10, 20, @x, @y, @z);
SELECT @x, @y, @z;
```

Résultats :

```
@x = 10
@y = 200
@z = 240
```

3.2.4 Arguments de la méthode `getMoreResults` de `Statement`

Les arguments de la méthode `getMoreResults` peuvent prendre les valeurs suivantes :

- `Statement.CLOSE_CURRENT_RESULT` : le `ResultSet` courant doit être fermé lors de l'appel à la méthode `getMoreResults` ;
- `Statement.CLOSE_ALL_RESULTS` : tous les `ResultSet` précédemment ouverts doivent être fermés lors de l'appel à la méthode `getMoreResults` ;
- `Statement.KEEP_CURRENT_RESULT` : Le `ResultSet` courant ne doit pas être fermé lors de l'appel à la méthode `getMoreResults`.

L'argument par défaut est `Statement.CLOSE_ALL_RESULTS`.

3.2.5 Appel de fonction

L'appel d'une fonction grâce aux `CallableStatement` utilise la syntaxe suivante :

```
String sql = "{? = call nomDeLaProcedure[(?, ?, ...)]}";
```

Le premier paramètre doit être enregistré comme paramètre de sortie :

```
String sql = "?= call Hello(?)";
CallableStatement statement = connection.prepareCall(sql);
statement.registerOutParameter(1, Types.VARCHAR);
statement.setString(2, "world");
statement.execute();

System.out.println(statement.getString(1));
```

3.2.6 Savoir si un paramètre de retour (OUT) est de type SQL NULL

Cette question a son intérêt, car bien souvent un `CallableStatement` va renvoyer une valeur par défaut s'il rencontre un type `SQL NULL`. Ce sera par exemple la chaine vide pour `VARCHAR` ou `0` pour `INTEGER` ou `NUMERIC`. `CallableStatement` propose donc la méthode `wasNull` qui renvoie un `boolean` indiquant si le dernier paramètre de type `OUT` était `NULL`. Cette méthode doit être utilisée seulement après l'appel de la méthode `get` correspondant au paramètre de sortie.

Par exemple :

```
String sql = "{getUnNombre(?)}";
CallableStatement statement = connection.prepareCall(sql);
statement.registerOutParameter(1, Types.INTEGER);
statement.execute();

int resultat = statement.getInt(1);
if(statement.wasNull()){
   System.out.pritnln("Le résultat est de type SQL NULL");
}
else{
   System.out.println("Le résultat vaut " + resultat);
}
```

3.3 Mode Php

Si le pilote de la base de données le prend en charge, vous pouvez également lier des paramètres aussi bien pour l'entrée que pour la sortie.
Les paramètres de sortie sont utilisés typiquement pour récupérer les valeurs d'une procédure stockée. Les paramètres de sortie sont un peu plus complexes à utiliser que les paramètres d'entrée car vous devez savoir la longueur qu'un paramètre donné pourra atteindre lorsque vous le liez. Si la valeur retournée est plus longue que la taille qui vous aurez suggéré, une erreur sera émise.

3.3.1 Simple exemple

```
<?php
$stmt = $dbh->prepare("CALL maproc(?)");
$stmt->bindParam(1, $return_value, PDO::PARAM_STR, 4000);

// Appel de la procédure stockée
$stmt->execute();

print "La procédure a retourné : $return_value\n";
?>
```

3.3.2 Autre exemple

3.3.2.1 La procédure

```
DROP FUNCTION 'ajoutFormateur'//
CREATE DEFINER = 'root'@'*'
  FUNCTION 'ajoutFormateur'(nom varchar(20),
```

```
                              prenom varchar(20),
                              adresse varchar(100),
                              password char(6))
    RETURNS int(11)
    DETERMINISTIC
BEGIN
  DECLARE v_ret INT;
  SELECT COUNT(*) INTO v_ret
    FROM formateur f
    WHERE f.nom LIKE nom
    AND f.prenom LIKE prenom
    AND f.adresse LIKE adresse;

  IF v_ret = 0 THEN
    INSERT INTO formateur
      VALUES (null, nom, prenom, adresse, MD5(password));
    SET v_ret = last_insert_id();
  ELSE
    SET v_ret = -1;
  END IF;

  RETURN v_ret;
END
```

 Les fonctions déterministes retournent toujours le même résultat quel que soit le moment auquel elles sont appelées avec un ensemble spécifique de valeurs d'entrée et sur la base du même état de la base de données.
Les fonctions non déterministes peuvent retourner différents résultats chaque fois qu'elles sont appelées avec un ensemble spécifique de valeurs d'entrée, même si l'état de la base de données à laquelle elles accèdent demeure inchangé.

Par exemple, la fonction `AVG` retourne toujours le même résultat pour les conditions ci-dessus, mais la fonction `GETDATE`, qui retourne la valeur `datetime` actuelle, retourne toujours un résultat différent.

3.3.2.2 Php

```
<?php
session_start();

try{
  $bdd = new PDO('mysql:host=localhost;dbname=formation',
                 'root',
                 '');
  }
  catch (Exception $e){
    die('Erreur : ' . $e->getMessage());
  }
?>
```

```php
<?php
  $nom =       'nom';
  $prenom =    'prenom';
  $adresse =   'adresse';
  $password =  'mdp';
  $stmt = $dbh->prepare("CALL ajoutFormateur(?)");

  $stmt->bindParam(1, $nom, PDO::PARAM_STR);
  $stmt->bindParam(1, $prenom, PDO::PARAM_STR);
  $stmt->bindParam(1, $adresse, PDO::PARAM_STR);
  $stmt->bindParam(1, $mdp, PDO::PARAM_STR);
  $stmt->execute();

  echo "La procédure a retourné : $return_value";
?>
```

3.3.3 Appel d'une procédure stockée avec un paramètre de sortie

```php
<?php
…
$stmt = $dbh->prepare("CALL sp_returns_string(?)");
$stmt->bindParam(1, $return_value, PDO::PARAM_STR, 4000);

// Appel de la procédure stockée
$stmt->execute();

print "La procédure a retourné : $return_value\n";
?>
```

3.3.4 Autre exemple

3.3.4.1 SQL

```sql
DROP PROCEDURE IF EXISTS 'proc1';
DELIMITER $$
CREATE PROCEDURE proc1( OUT Result BIGINT)
BEGIN
    DECLARE o BIGINT UNSIGNED;
    SET o := 99999999;
    SELECT o INTO Result;
END
$$
```

3.3.4.2 Php

```php
/*Variables pour la connexion à la base de données*/
$name_db       = "nom de ma base de données";
$pw_db         = "mot de passe";
$name_user     = "utilisateur";
$host          = "hôte";

/*Connexion à la base de données*/
```

```
$dbh = new PDO('mysql:host = '.$host.
               ';dbname = '.$name_db,
               $name_user,
               $pw_db);

echo "req1\n";
// Appel de la procédure stockée
$stmt = $dbh->prepare("CALL proc1(@TOTO)");
$stmt->execute();
$return_value = $stmt->fetch(PDO::FETCH_ASSOC) ;
var_dump($return_value);

echo "req2 \n";
// Appel de la procédure stockée
$stmt = $dbh->prepare("SELECT @TOTO");
$stmt->execute();
$return_value = $stmt->fetch(PDO::FETCH_ASSOC) ;
var_dump($return_value);

print "La procédure a retourné : ".$return_value['@TOTO']."\n";
```

3.3.5 Appel d'une procédure stockée avec paramètre d'entrée/sortie

```
<?php
$stmt = $dbh->prepare("CALL sp_takes_string_returns_string(?)");
$value = 'hello';
$stmt->bindParam(1,
                 $value,
                 PDO::PARAM_STR|PDO::PARAM_INPUT_OUTPUT,
                 4000);

// appel de la procédure stockée
$stmt->execute();

print "La procédure a retourné : $value\n";
?>
```

3.3.6 Les `fetch` modes de PDO

PDO propose une quantité assez impressionnante de *modes de récupération* (*fetch mode*) des données. Entendez par là qu'il est possible de personnaliser le comportement des méthodes `fetch` et `fetchAll` pour obtenir les résultats des requêtes SQL sous des formes très diverses.

Si certains modes sont relativement classiques et ont des équivalents avec les précédents drivers MySQL (`mysql_fetch_array`, `mysql_fetch_row`, etc.), d'autres sont totalement inédits… mais très mal documentés ! Et pourtant, certains sont très pratiques… Voici donc une série d'articles présentant tous les modes de `fetch`, leur fonctionnement et surtout leurs performances.

Le mode de récupération peut être définis de plusieurs façons :

- directement en paramètre de la méthode `query()` ;

- avec la méthode `setFetchMode()` appliquée à l'objet de résultats ;
- en paramètre de la méthode `fetchAll()` ;
- en paramètre de la méthode `fetch()` ; dans ce cas le mode s'applique ligne par ligne, et il est ainsi possible de changer de mode pendant la récupération !

Attention, tous les modes ne peuvent pas s'utiliser avec toutes les méthodes (sinon ça serait trop simple) : certains fonctionnement seulement avec `fetchAll`, d'autres avec `fetch`, d'autres encore nécessitent obligatoirement un appel à `setFetchMode` pour passer des paramètres supplémentaires…

À propos de `fetchAll` : il n'est pas recommandé de l'utiliser pour traiter de grosses requêtes, car cela revient à mettre d'abord en mémoire la totalité des résultats pour les traiter ensuite. En utilisant `fetch` dans une boucle `while`, on met en mémoire une ligne et on la traite, puis la suivante réutilise la mémoire occupée, etc.

3.3.6.1 FETCH_BOTH

C'est le mode de récupération par défaut de PDO. Le tableau qui est retourné contient les résultats en double : une fois indexés par le nom des colonnes et une deuxième fois indexés par leur numéro.

Exemple :

```
$results->fetchAll(PDO::FETCH_BOTH);
```

```
Array (
  [0] => Array (
    [id] => 1
    [0] => 1
    [login] => toto
    [1] => toto
  )

  [1] => Array (
    [id] => 2
    [0] => 2
    [login] => titi
    [1] => titi
  )
)
```

Il est possible de modifier le mode de récupération par défaut avec :

```
$dbh->setAttribute(
    PDO::ATTR_DEFAULT_FETCH_MODE,
    PDO::FETCH_<em>FOOBAR</em>);
```

Ce mode sera utilisé lorsqu'aucun autre mode n'est précisé, mais parfois PDO revient malgré tout à `FETCH_BOTH`, alors on ne peut pas dire que ça soit très fiable.

3.3.6.2 FETCH_NUM

Le tableau retourné contient les résultats indexés par le numéro de la colonne.

Exemple :

```
$results->fetchAll(PDO::FETCH_NUM);
```

```
Array (
  [0] => Array (
    [0] => 1
    [1] => toto
  )

  [1] => Array (
    [0] => 2
    [1] => titi
  )
)
```

Par rapport à `FETCH_BOTH` :

- Temps de récupération : -7,1%
- Taille du résultat : -23%

Ce mode est le plus rapide et retourne le tableau le plus petit. Mais il n'est pas toujours pratique de travailler avec le numéro des colonnes. D'autant plus que le moindre changement dans la requête SQL (suppression d'une colonne par exemple) risque de casser tout le reste du script ! En clair, à utiliser dans des cas très spécifiques et quand le nombre de colonnes est réduit pour ne pas risquer d'avoir une application impossible à maintenir.

3.3.6.3 FETCH_ASSOC

Le tableau retourné contient les résultats indexés par le nom de la colonne. Ce mode ne permet pas de récupérer plusieurs colonnes avec le même nom (une seule valeur est retournée, en général celle la plus à droite dans la requête, c'est-à-dire. la dernière).

Exemple :

```
$results->fetchAll(PDO::FETCH_ASSOC);
```

```
Array (
  [0] => Array (
    [id] => 1
    [login] => toto
  )

  [1] => Array (
    [id] => 2
    [login] => titi
  )
)
```

Par rapport à FETCH_BOTH :

- Temps de récupération : -4%
- Taille du résultat : -20%

Les performances sont légèrement moins bonnes qu'avec FETCH_NUM (mais toujours bien meilleures que le mode par défaut).

3.3.6.4 FETCH_NAMED

Ce mode un peu particulier est identique à FETCH_ASSOC, sauf dans le cas où plusieurs colonnes portent le même nom dans la requête SQL : FETCH_ASSOC ne va retourner qu'une seule valeur tandis que FETCH_NAMED va retourner un tableau de valeurs.

Prenons par exemple la requête suivante :

```
SELECT email as id, id, login FROM user LIMIT 0,2;
```

Le résultat avec FETCH_ASSOC est identique à ci-dessus, et la première colonne n'est pas accessible. Avec FETCH_NAMED, on obtient :

```
$results->fetchAll(PDO::FETCH_NAMED);
```

```
Array (
  [0] => Array (
    [id] => Array (
      [0] => 1
      [1] => toto@foo
    )
    [login] => toto
  )

  [1] => Array (
    [id] => Array (
     [0] => 2
     [1] => titi@foo
    )
    [login] => titi
  )
)
```

3.3.6.5 Autres modes de récupération

FETCH_BOUND : retourne TRUE et assigne les valeurs des colonnes du jeu de résultats dans les variables PHP à laquelle elles sont liées avec la méthode PDOStatement::bindColumn().

FETCH_CLASS : retourne une nouvelle instance de la classe demandée, liant les colonnes du jeu de résultats aux noms des propriétés de la classe et en appelant le constructeur par la suite, sauf si PDO::FETCH_PROPS_LATE est également donné.

Si `fetch_style` inclut `PDO::FETCH_CLASS` (c'est-à-dire `PDO::FETCH_CLASS | PDO::FETCH_CLASSTYPE`), alors le nom de la classe est déterminé à partir d'une valeur de la première colonne.

`FETCH_INTO` : met à jour une instance existante de la classe demandée, liant les colonnes du jeu de résultats aux noms des propriétés de la classe.

`FETCH_OBJ` : retourne un objet anonyme avec les noms de propriétés qui correspondent aux noms des colonnes retournés dans le jeu de résultats.

`FETCH_PROPS_LATE` : lorsqu'il est utilisé avec `PDO::FETCH_CLASS`, le constructeur de la classe est appelé avant que les propriétés ne soient assignées à partir des valeurs de colonne respectives.

`FETCH_LAZY` : combine `PDO::FETCH_BOTH` et `PDO::FETCH_OBJ`, créant ainsi les noms des variables de l'objet, comme elles sont accédées.

`FETCH_BOUND` : retourne `true` et assigne les valeurs des colonnes du jeu de résultats dans les variables PHP auxquelles elles sont liées avec la méthode `PDOStatement::bindParam()` ou la méthode `PDOStatement::bindColumn()`.

`FETCH_CLASS | FETCH_CLASSTYPE` :

retourne une nouvelle instance de la classe demandée, liant les colonnes aux propriétés nommées dans la classe.

Nom de la classe = 1ère colonne.

4. LES FONCTIONS

Une fonction comme une procédure s'exécute sur le serveur, mais une fonction retourne un résultat et peut être utilisée directement dans une requête SQL .

4.1 MODE SQL

4.1.1 Création d'une fonction

On utilise la commande **`CREATE FUNCTION name (params) RETURNS returnType`**

```
DELIMITER |
CREATE FUNCTION getlib (param_id INT) RETURNS CHAR(50)
BEGIN
 DECLARE lib CHAR(50);
 SELECT Libelle INTO lib FROM T1 WHERE id = param_id;
 RETURN lib;
END;
 |
```

4.1.2 Erreur 1418

Lorsque vous essayez d'exécuter une fonction stockée, si `binlog_format = STATEMENT` est défini, le mot clé `DETERMINISTIC` doit être spécifié dans la définition de la fonction. Si ce n'est pas le cas, une erreur est générée et la fonction ne s'exécute pas, sauf si `log_bin_trust_function_creators = 1` est spécifié pour ignorer cette vérification. Pour les appels de fonction récursifs, le mot clé `DETERMINISTIC` est requis uniquement pour l'appel le plus à l'extérieur.

Si la journalisation binaire en mode ligne ou mixte est en cours d'utilisation, l'instruction est acceptée et répliquée même si la fonction a été définie sans le mot clé `DETERMINISTIC`.

```
set global log_bin_trust_routine_creators = 1;
```

4.1.3 Utilisation d'une fonction stockée dans une requête SQL

```
SELECT getlib(id) FROM T WHERE ...
```

Pour la création de routines telles que les procédures ou fonctions, il faut posséder le droit `ALTER ROUTINE`.

Exemple de fonction qui arrondit un montant à 50 centimes prés :

```
DELIMITER |
CREATE FUNCTION arrondi50(v DECIMAL(8,2)) RETURNS DECIMAL(8,2)
    RETURN ROUND((v * 2) + 0.49999) / 2;
    |
DELIMITER ;
```

4.1.4 IF … ELSE … THEN

```
IF var = 2 THEN
    ...
```

```
ELSE
    ...
ELSEIF
    ...
END IF;
```

4.1.5 CASE

Suivant la valeur de la variable qui suit `CASE`, le programme va traiter tel ou tel cas

```
CASE var
WHEN 1 THEN ...;
WHEN 2 THEN ...;
ELSE  ...;  // autres cas
END CASE;
```

4.1.6 Boucle LOOP

```
LOOP
    ...
END LOOP
```

Si on souhaite sortir de la boucle on doit rajouter une étiquette. L'appel à `LEAVE` suivi de l'étiquette provoque la sortie de la boucle.

```
LOOP
    ...
    IF myvar = 0 THEN LEAVE unlabel;
END LOOP unlabel;
```

4.1.7 Boucle REPEAT UNTIL

```
REPEAT
    ...
UNTIL var = 5 END REPEAT;
```

Si on veut recommencer l'itération

```
un_label: REPEAT
        ...
    IF i = 3 THEN ITERATE un_label; END IF;
UNTIL i < 100 END REPEAT un_label;
```

4.1.8 Boucle WHILE

```
WHILE i < 100 DO
    ...
END WHILE
```

4.2 Mode Java

L'appel d'une fonction grâce aux `CallableStatement` utilise la syntaxe suivante :

```
String sql = "{? = call nomDeLaProcedure[(?, ?, ...)]}";
```

Le premier paramètre doit être enregistré comme paramètre de sortie :

```
String sql = "?= call Hello(?)";
CallableStatement statement = connection.prepareCall(sql);
statement.registerOutParameter(1, Types.VARCHAR);
statement.setString(2, "world");
statement.execute();

System.out.println(statement.getString(1));
```

4.2.1 Comment savoir si un paramètre de retour est de type sql NULL ?

Cette question a son intérêt, car bien souvent un `CallableStatement` va renvoyer une valeur par défaut s'il rencontre un type SQL `NULL`.
Ce sera par exemple la chaine vide pour `VARCHAR` ou `0` pour `INTEGER` ou `NUMERIC`. `CallableStatement` propose donc la méthode `wasNull` qui renvoie un `boolean` indiquant si le dernier paramètre de type `OUT` était `NULL`. Cette méthode doit être utilisée seulement après l'appel de la méthode `get` correspondant au paramètre de sortie.

Par exemple :

```
String sql = "{getUnNombre(?)}";
CallableStatement statement = connection.prepareCall(sql);
statement.registerOutParameter(1, Types.INTEGER);
statement.execute();

int resultat = statement.getInt(1);
if(statement.wasNull()){
  System.out.pritnln("Le résultat est de type SQL NULL");
}else{
  System.out.println("Le résultat vaut " + resultat);
}
```

5. LES CURSEURS

Un Curseur est un moyen de parcourir les résultats d'une requête `SELECT` et d'en traiter les valeurs une par une quel que soit le nombre de lignes qui ont été récupérées dans la requête `SELECT`.

5.1 MODE SQL

On rencontre les curseurs dans les procédures stockées.

5.1.1 Déclaration

Voici la syntaxe sql de déclaration d'un curseur :

```
DECLARE cursor_utilisateur CURSOR
FOR SELECT * FROM utilisateur;
```

Pour déclarer un curseur, il faut utiliser l'instruction `DECLARE CURSOR` avec le `SELECT` permettant d'avoir le jeu de données dans la zone de déclaration de la procédure. C'est à dire que la déclaration d'un curseur ne peut se faire qu'à l'intérieur d'un bloc `BEGIN` / `END`. Nous allons donc étudier une procédure stockée utilisant un curseur et détailler son fonctionnement.

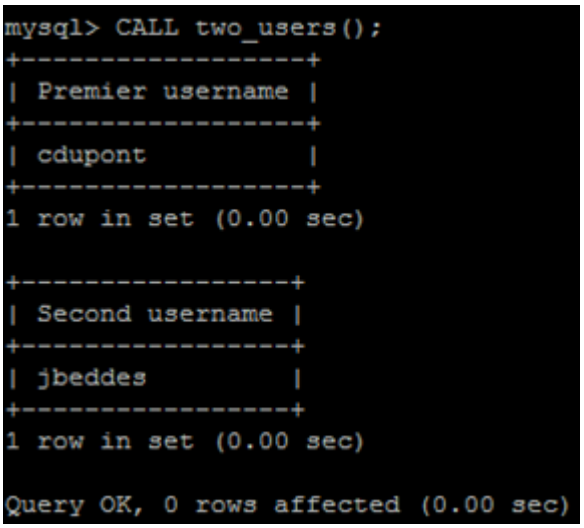
```
DELIMITER $$
CREATE PROCEDURE two_users()
BEGIN

  DECLARE c_username VARCHAR(100);

  DECLARE cursor_users CURSOR
  FOR SELECT username FROM utilisateur

    OPEN cursor_users;

    FETCH cursor_users INTO c_username;
    SELECT c_username AS "Premier username";

    FETCH cursor_users INTO c_username;
    SELECT c_username AS "Second username";

    CLOSE cursor_users;
END $$
DELIMITER ;
```

Tout d'abord, il faut modifier le délimiteur pour que MySQL interprète correctement la suite. Ensuite, dans la procédure, on définit une variable `c_username` qui servira à prendre les valeurs données par le curseur dans la suite de la procédure. Puis il faut créer le curseur avec la syntaxe vue précédemment.

C'est ici que les choses se compliquent. En effet, l'instruction `OPEN cursor_users` permet d'ouvrir le curseur pour l'utiliser. Par la suite, l'instruction `FETCH cursor_users INTO c_username` permet de donner la valeur retournée par le curseur dans la variable `c_username`. Il faut aussi noter que `FETCH` permet d'avancer dans les résultats du curseur.

Ainsi, on ne tombera pas deux fois sur les mêmes données. Afin de terminer correctement la procédure, il ne faut pas oublier de fermer le curseur.

Voici le retour de cette procédure stockée sous MySQL :



5.1.2 Boucler sur les résultats d'un curseur

Pour boucler sur les résultats d'un curseur, il va falloir ouvrir le curseur et la boucle, affecter les champs de la ligne à des variables, faire les traitements que vous souhaitez, et enfin fermer la boucle et le curseur :

```
DECLARE var_prix_ht DECIMAL(18,2);
DECLARE var_taux_tva DECIMAL(18,2);
DECLARE var_prix_ttc DECIMAL(18,2);
...

OPEN monCurseur;
tarif_loop: LOOP
  FETCH monCurseur INTO var_prix_ht, var_taux_tva, var_prix_ttc;
  ...
END LOOP;
CLOSE monCurseur;
```

Une fois vos variables alimentées par le `FETCH`, vous pouvez faire les différents traitements appropriés.

5.1.3 Définir une condition d'arrêt à la boucle

En l'état, avec le code ci-dessus vous aurez une erreur d'exécution lorsqu'il va parcourir le curseur et arriver à la fin du jeu d'enregistrement. Pour éviter cela, et pour avoir également une condition de sortie à la boucle (et éviter une éventuelle boucle infinie), vous allez pouvoir définir un `HANDLER` dédié.

```
DECLARE done INT DEFAULT 0;
...
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
```

```
...
/* dans la boucle */
IF done = 1 THEN
  LEAVE tarif_loop;
END IF;
```

Autre façon de sortir également, vous pouvez mettre à `NULL` vos variables en fin de tour, et tester si une variable qui ne devrait pas être nulle l'est dans la boucle, et quitter (`LEAVE`) à ce moment là :

```
/* dans la boucle */
IF var_prix_ht IS NULL THEN
  LEAVE tarif_loop;
END IF;
...
SET var_prix_ht = NULL;
```

En résumé, le code complet de notre exemple serait le suivant :

```
DECLARE var_prix_ht DECIMAL(18,2);
DECLARE var_taux_tva DECIMAL(18,2);
DECLARE var_prix_ttc DECIMAL(18,2);
DECLARE done INT DEFAULT 0;
DECLARE monCurseur CURSOR FOR
  SELECT prix_ht, taux_tva, prix_ttc FROM article_tarif;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

OPEN monCurseur;
tarif_loop: LOOP
  FETCH monCurseur INTO var_prix_ht, var_taux_tva, var_prix_ttc;

  IF done = 1 THEN
    LEAVE tarif_loop;
  END IF;

/* mettre ici le traitement à effectuer */

END LOOP;
CLOSE monCurseur;
```

5.1.4 Conclusion

Enfin pour terminer cette partie sur les curseurs, il faut savoir qu'ils sont utilisés dans de nombreux langages de programmation de la même façon pour parcourir les résultats d'un `SELECT`.

Quels intérêts aux curseurs ? Ils offrent la possibilité de traiter les enregistrements un par un et non dans leur totalité. Cet avantage a bien évidemment un revers important : les performances sont toutes autant bridées. Chaque enregistrement nécessite son lot de traitement, contrairement à une requête classique où la majorité des traitements n'interviennent qu'une fois à la fin.

Beaucoup de codes traitent les actions après récupération complète, ce qui multiplie les requêtes ou déporte le travail sur les serveurs métiers.

5.2 Mode Java

Contrôle du curseur d'un `ResultSet`

Exemple

```
import java.sql.*;

public class Connect {
  private final String login;
  private final String mdp;
  private final String driver;

  public static void main(String args[]) {
    login = "root";
    mdp = "root";
    driver = "jdbc:mysql://localhost/nosamisleschiens";

    try {
      Class.forName("com.mysql.jdbc.Driver");
    }
    catch(java.lang.ClassNotFoundException e) {
      System.err.print("ClassNotFoundException: ");
      System.err.println(e.getMessage());
    }
    try {
      Connection con = DriverManager.getConnection(
        driver,
        login,
        mdp);
      Statement st = con.createStatement(
        ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);

      // createStatement() par defaut :
      //   ResultSet.TYPE_FORWARD_ONLY
      String query = "SELECT * FROM maTable ";
      ResultSet rs = st.executeQuery(query);
      rs.last();
      int nombreLignes = rs.getRow();

      StringBuilder sb = new StringBuilder();
      sb.append("Ce ResultSet contient ")
        .append(nombreLignes)
        .append(" lignes.");
      System.out.println(sb.toString());

      Do {
        StringBuider sb = new StringBuilder();
```

```
        sb.append("Nom : ")
          .append(rs.getString("nom"))
          .append(" nombre de puces = ")
          .append(rs.getString("Prenom");
        System.out.println(sb.toString());
      }
      while (rs.previous());
      rs.close();
      con.close();
    }
    catch(SQLException ex) {
      StringBuilder sb = new StringBuilder();
      sb.append("SQLException: ")
        .append(ex.getMessage());
      System.err.println(sb.toString());
    }
  }
}
```

La boucle d'affichage part du dernier enregistrement pour aller au premier.

Contrôle du `ResultSet` et en particulier de son curseur :

- Les attributs/comportements du `ResultSet` définis lors de sa création de l'instruction `createStatement()`
  - Déplacement du curseur du `ResultSet` :
    - `ResultSet.TYPE_FORWARD_ONLY`
      Déplacement en avant seulement

      Valeur par défaut

      Quand la dernière ligne atteinte, le `ResultSet` est fermé
    - `ResultSet.TYPE_SCROLL_INSENSITIVE`
      Dans les deux sens, de manière absolue ou relative.

      De plus, le `ResultSet` est insensible aux modifications des valeurs dans la base de données.

      `ResultSet.TYPE_SCROLL_SENSITIVE` : Cette valeur indique que le curseur peut être déplacé dans les deux sens, mais aussi arbitrairement (de manière absolue ou relative). De plus, le `ResultSet` est sensible aux modifications des valeurs dans la base de données.
  - Mise à jour :
    - `ResultSet.CONCUR_READ_ONLY` :

      Valeur par défaut

      Les données contenues dans le `ResultSet` sont seulement lisibles.
    - `ResultSet.CONCUR_UPDATABLE` :

      Il est possible modifier les données de la base via le `ResultSet`.
    - …
  - Méthodes du curseur :

- `getRow()`

Donne la position actuelle du curseur

- `next()`
- `previous()`
- `first()`
- `last()`
- `beforeFirst()`
  Positionne le curseur avant le premier élément

  Position par défaut du curseur à la création du ResulSet
- `afterLast()`
  Positionne le curseur après le dernier élément
- `absolute(`entier_positif`)`
  Positionne le curseur à l'index indiqué
- `relative(`entier `dx)`
  Positionne le curseur à la ligne courante + `dx`

  `dx` étant un entier positif ou négatif

CREDITS
Arnaud Guignant – gafish
Huggy – grafikart
Jérémy Beddes - Supinfo
Université de Picardie Jules verne
Développez.com
Php.net
Alsacreation
ŒUVRE COLLECTIVE DE l'AFPA
Sous le pilotage de la DIIP et du centre d'ingénierie sectoriel Tertiaire-Services

Equipe de conception (IF, formateur, mediatiseur)
Ludovic Domenici - Formateur Développeur logiciel
Ch. Perrachon – Ingénieure de formation

Date de mise à jour : 25/03/19