

Administrez vos bases de données avec MySQL

40 heures  Moyenne

Mis à jour le 03/09/2019



Procédures stockées

 [Connectez-vous](#) ou [inscrivez-vous](#) gratuitement pour bénéficier de toutes les fonctionnalités de ce cours !

Les procédures stockées sont disponibles depuis la version 5 de MySQL. Elles permettent d'automatiser des actions qui peuvent être très complexes.

Une procédure stockée est en fait une **série d'instructions SQL** désignée par un **nom**. Lorsque l'on crée une procédure stockée, on l'enregistre **dans la base** au même titre qu'une table, par exemple. Une fois la procédure créée, il est possible d'**appeler** celle-ci par son nom. Les instructions de la procédure sont

Contrairement aux requêtes préparées, qui ne sont gardées en mémoire que pour la session courante, les procédures stockées sont, comme leur nom l'indique, **durables**, et font bien **partie intégrante de la base de données** dans laquelle elles sont enregistrées.

Création et utilisation d'une procédure

Voyons tout de suite la syntaxe à utiliser pour créer une procédure :

```
1 CREATE PROCEDURE nom_procedure ([parametre1 [, parametre2, ...]])
2 corps de la procédure;
```

Décodons tout ceci.

- CREATE PROCEDURE** : sans surprise, il s'agit de la commande à exécuter pour créer une procédure. On fait suivre cette commande du nom que l'on veut donner à la procédure.
- ([parametre1 [, parametre2, ...]])** : après le nom de la procédure viennent des parenthèses. **Celles-ci sont obligatoires !** À l'intérieur de ces parenthèses, on peut mettre des paramètres de la procédure. Ces paramètres sont des variables qui pourront être utilisées par la procédure.
- Corps de la procédure : c'est là que l'on met le **contenu** de la procédure, ce qui va être exécuté lorsqu'on lance la procédure. Cela peut être soit **une série d'instructions**.

Les noms des procédures stockées ne sont pas sensibles à la casse.

Procédure avec une seule requête

Voici une procédure toute simple, sans paramètres, qui va juste afficher toutes les races d'animaux.

```
1 CREATE PROCEDURE afficher_races_requete()
2 -- pas de paramètres dans les parenthèses
3 SELECT id, nom, espece_id, prix FROM Race;
```

Procédure avec un bloc d'instructions

Pour délimiter un bloc d'instructions (qui peut donc contenir plus d'une instruction), on utilise les mots **BEGIN** et **END**.

```
1 BEGIN
2 -- Série d'instructions
3 END;
```

Exemple : reprenons la procédure précédente, mais en utilisant un bloc d'instructions.

```
1 CREATE PROCEDURE afficher_races_bloc()
2 -- pas de paramètres dans les parenthèses
3 BEGIN
```

```
4 SELECT id, nom, espece_id, prix FROM Race;
5 END;
```

Malheureusement...

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syn

Que s'est-il passé ? La syntaxe semble correcte...

Les mots-clés sont bons, il n'y a pas de paramètres, mais on a bien mis les parenthèses, `BEGIN` et `END` sont tous les deux présents. Tout cela est correct visiblement omis un détail.

Peut-être aurez-vous compris que le problème se situe au niveau du caractère `;` : en effet, un `;` termine une instruction SQL. Or, on a mis un `;` à `SELECT * FROM Race;`. Cela semble logique, mais pose problème puisque c'est le premier `;` rencontré par l'instruction `CREATE PROCEDURE`, qui nature Ceci déclenche une erreur puisqu'en réalité, l'instruction `CREATE PROCEDURE` n'est pas terminée : le bloc d'instructions n'est pas complet !

Comment faire pour écrire des instructions à l'intérieur d'une instruction alors ?

Il suffit de changer le délimiteur !

Délimiteur

Ce que l'on appelle délimiteur, c'est tout simplement (par défaut), le caractère `;`. C'est-à-dire le caractère qui permet de **délimiter les instructions**. Or, il est tout à fait possible de définir le délimiteur manuellement, de manière à ce que `;` ne signifie plus qu'une instruction se termine. Auquel cas le à l'intérieur d'une instruction, et donc dans le corps d'une procédure stockée.

Pour changer le délimiteur, il suffit d'utiliser cette commande :

```
1 DELIMITER |
```

À partir de maintenant, vous devrez utiliser le caractère `|` pour signaler la fin d'une instruction. `;` ne sera plus compris comme tel par votre session

```
1 SELECT 'test' |
```

test

test

`DELIMITER` n'agit que pour la **session courante**.

Vous pouvez utiliser le ou les caractères de votre choix comme délimiteur. Bien entendu, il vaut mieux choisir quelque chose qui ne risque pas d'être utilisé. Bannissez donc les lettres, les chiffres, le `@` (qui sert pour les variables utilisateurs) et le `\` (qui sert à échapper les caractères spéciaux).

Les deux délimiteurs suivants sont les plus couramment utilisés :

```
1 DELIMITER //
2 DELIMITER |
```

Bien ! Ceci étant réglé, reprenons !

Création d'une procédure stockée

```
1 DELIMITER | -- On change le délimiteur
2 CREATE PROCEDURE afficher_races()
3     -- toujours pas de paramètres, toujours des parenthèses
4 BEGIN
5     SELECT id, nom, espece_id, prix
6     FROM Race; -- Cette fois, le ; ne nous embêtera pas
7 END | -- Et on termine bien sûr la commande CREATE PROCEDURE par notre nouveau délimiteur
```

Cette fois-ci, tout se passe bien. La procédure a été créée.

Lorsque l'on utilisera la procédure, quel que soit le délimiteur défini par `DELIMITER`, les instructions à l'intérieur du corps de la procédure seront bien lors de la création d'une procédure, celle-ci est interprétée – on dit aussi "parsée" – par le serveur MySQL et le parseur des procédures stockées inter

délimiteur. Il n'est pas influencé par la commande `DELIMITER`.

Les procédures stockées n'étant que très rarement composées d'une seule instruction, on utilise presque toujours un bloc d'instructions pour le corps de

Utilisation d'une procédure stockée

Pour appeler une procédure stockée, c'est-à-dire déclencher l'exécution du bloc d'instructions constituant le corps de la procédure, il faut utiliser le mot-clé `CALL` suivi de la procédure appelée, puis de parenthèses (avec éventuellement des paramètres).

```
1 CALL afficher_races() | -- le délimiteur est toujours | !!!
```

id	nom	espece_id	prix
1	Berger allemand	1	485.00
2	Berger blanc suisse	1	935.00
3	Singapura	2	985.00
4	Bleu russe	2	835.00
5	Maine coon	2	735.00
7	Sphynx	2	1235.00
8	Nebelung	2	985.00
9	Rottweiler	1	600.00

Le bloc d'instructions a bien été exécuté (un simple `SELECT` dans ce cas).

Les paramètres d'une procédure stockée

Maintenant que l'on sait créer une procédure et l'appeler, intéressons-nous aux paramètres.

Sens des paramètres

Un paramètre peut être de trois sens différents : entrant (`IN`), sortant (`OUT`), ou les deux (`INOUT`).

- `IN` : c'est un paramètre "entrant". C'est-à-dire qu'il s'agit d'un paramètre dont la valeur est fournie à la procédure stockée. Cette valeur sera utilisée pour un calcul ou une sélection, par exemple).
- `OUT` : il s'agit d'un paramètre "sortant", dont la valeur sera établie au cours de la procédure et qui pourra ensuite être utilisé en dehors de cette procédure.
- `INOUT` : un tel paramètre sera utilisé pendant la procédure, verra éventuellement sa valeur modifiée par celle-ci, et sera ensuite utilisable en dehors de la procédure.

Syntaxe

Lorsque l'on crée une procédure avec un ou plusieurs paramètres, chaque paramètre est défini par trois éléments.

- Son sens : entrant, sortant, ou les deux. Si aucun sens n'est donné, il s'agira d'un paramètre `IN` par défaut.
- Son nom : indispensable pour le désigner à l'intérieur de la procédure.
- Son type : `INT`, `VARCHAR(10)` ...

Exemples

Procédure avec un seul paramètre entrant

Voici une procédure qui, selon l'*id* de l'espèce qu'on lui passe en paramètre, affiche les différentes races existant pour cette espèce.

```
1 DELIMITER | -- Facultatif si votre délimiteur est toujours |
2 CREATE PROCEDURE afficher_race_selon_espece (IN p_espece_id INT)
3 -- Définition du paramètre p_espece_id
4 BEGIN
5     SELECT id, nom, espece_id, prix
6     FROM Race
7     WHERE espece_id = p_espece_id; -- Utilisation du paramètre
8 END |
9 DELIMITER ; -- On remet le délimiteur par défaut
```

Notez que, à la suite de la création de la procédure, j'ai remis le délimiteur par défaut `;`. Ce n'est absolument pas obligatoire, vous pouvez continuer à utiliser le délimiteur `|` si vous le souhaitez.

Pour l'utiliser, il faut donc passer une valeur en paramètre de la procédure, soit directement, soit par l'intermédiaire d'une variable utilisateur.

```
1 CALL afficher_race_selon_espece(1);
2 SET @espece_id := 2;
3 CALL afficher_race_selon_espece(@espece_id);
```

id	nom	espece_id	prix
1	Berger allemand	1	485.00
2	Berger blanc suisse	1	935.00
9	Rottweiler	1	600.00

id	nom	espece_id	prix
3	Singapura	2	985.00
4	Bleu russe	2	835.00
5	Maine coon	2	735.00
7	Sphynx	2	1235.00
8	Nebelung	2	985.00

Le premier appel à la procédure affiche bien toutes les races de chiens, et le second, toutes les races de chats.

J'ai fait commencer le nom du paramètre par "p_". Ce n'est pas obligatoire, mais je vous conseille de le faire systématiquement pour vos paramètres facilement. Si vous ne le faites pas, soyez extrêmement prudent avec les noms que vous leur donnez. Par exemple, dans cette procédure, si l'on avait *espece_id*, cela aurait posé problème, puisque *espece_id* est aussi le nom d'une colonne dans la table *Race*. Qui plus est, c'est le nom de la colonne condition `WHERE`. En cas d'ambiguïté, MySQL interprète l'élément comme étant le paramètre, et non la colonne. On aurait donc eu `WHERE 1 = 1`, qui est vrai.

Procédure avec deux paramètres, un entrant et un sortant

Voici une procédure assez similaire à la précédente, si ce n'est qu'elle n'affiche pas les races existant pour une espèce, mais compte combien il y en a, puis un paramètre sortant.

```
1 DELIMITER |
2 CREATE PROCEDURE compter_races_selon_espece (p_espece_id INT, OUT p_nb_races INT)
3 BEGIN
4     SELECT COUNT(*) INTO p_nb_races
5     FROM Race
6     WHERE espece_id = p_espece_id;
7 END |
8 DELIMITER ;
```

Aucun sens n'a été précisé pour *p_espece_id*, il est donc considéré comme un paramètre entrant.

`SELECT COUNT(*) INTO p_nb_races`, voilà qui est nouveau ! Comme vous l'avez sans doute deviné, le mot-clé `INTO` placé après la clause `SELECT` permet de sélectionner par ce `SELECT` à des variables, au lieu de simplement afficher les valeurs sélectionnées.

Dans le cas présent, la valeur du `COUNT(*)` est assignée à *p_nb_races*.

Pour pouvoir l'utiliser, il est nécessaire que le `SELECT` ne renvoie qu'une seule ligne, et il faut que le nombre de valeurs sélectionnées et le nombre de variables soient égaux.

Exemple 1 : `SELECT ... INTO` correct avec deux valeurs

```
1 SELECT id, nom INTO @var1, @var2
2 FROM Animal
3 WHERE id = 7;
4 SELECT @var1, @var2;
```

@var1	@var2
7	Caroline

Le `SELECT ... INTO` n'a rien affiché, mais a assigné la valeur 7 à *@var1*, et la valeur 'Caroline' à *@var2*. Nous les avons ensuite affichées avec un autre `SELECT`.

Exemple 2 : `SELECT ... INTO` incorrect, car le nombre de valeurs sélectionnées (deux) n'est pas le même que le nombre de variables à assigner (une).

```
1 SELECT id, nom INTO @var1
2 FROM Animal
3 WHERE id = 7;
```

ERROR 1222 (21000): The used SELECT statements have a different number of columns

Exemple 3 : `SELECT ... INTO` incorrect, car il y a plusieurs lignes de résultats.

```
1 SELECT id, nom INTO @var1, @var2
2 FROM Animal
3 WHERE espece_id = 5;
```

ERROR 1172 (42000): Result consisted of more than one row

Revenons maintenant à notre nouvelle procédure `compter_races_selon_espece()` et exécutons-la. Pour cela, il va falloir lui passer deux paramètres : `p_esc`. Le premier ne pose pas de problème, il faut simplement donner un nombre, soit directement, soit par l'intermédiaire d'une variable, comme pour la proc `afficher_race_selon_espece()`.

Par contre, pour le second, il s'agit d'un paramètre sortant. Il ne faut donc pas donner une valeur, mais quelque chose dont la valeur sera déterminée par `SELECT ... INTO`) et que l'on pourra utiliser ensuite : **une variable utilisateur !**

```
1 CALL compter_races_selon_espece (2, @nb_races_chats);
```

Et voilà ! La variable `@nb_races_chats` contient maintenant le nombre de races de chats. Il suffit de l'afficher pour vérifier.

```
1 SELECT @nb_races_chats;
```

@nb_races_chats

5

Procédure avec deux paramètres, un entrant et un entrant-sortant

Nous allons créer une procédure qui va servir à calculer le prix que doit payer un client. Pour cela, deux paramètres sont nécessaires : l'animal acheté (par paramètre `INOUT`).

La raison pour laquelle le prix est un paramètre à la fois entrant et sortant est que l'on veut pouvoir, avec cette procédure, calculer simplement un prix tot achèterait plusieurs animaux.

Le principe est simple : si le client n'a encore acheté aucun animal, le prix est de 0. Pour chaque animal acheté, on appelle la procédure, qui ajoute au prix question.

Une fois n'est pas coutume, commençons par voir les requêtes qui nous serviront à tester la procédure. Cela devrait clarifier le principe. Je vous propose c même la procédure correspondante avant de regarder à quoi elle ressemble.

```
1 SET @prix = 0; -- On initialise @prix à 0
2
3 CALL calculer_prix (13, @prix); -- Achat de Rouquine
4 SELECT @prix AS prix_intermediaire;
5
6 CALL calculer_prix (24, @prix); -- Achat de Cartouche
7 SELECT @prix AS prix_intermediaire;
8
9 CALL calculer_prix (42, @prix); -- Achat de Bilba
10 SELECT @prix AS prix_intermediaire;
11
12 CALL calculer_prix (75, @prix); -- Achat de Mimi
13 SELECT @prix AS total;
```

On passe donc chaque animal acheté tour à tour à la procédure, qui modifie le prix en conséquence. Voici quelques indices et rappels qui devraient vous procédure.

- Le prix n'est pas un nombre entier.
- Il est possible de faire des additions directement dans un `SELECT`.
- Pour déterminer le prix, il faut utiliser la fonction `COALESCE()`.

Réponse :

```
1 DELIMITER |
2
3 CREATE PROCEDURE calculer_prix (IN p_animal_id INT, INOUT p_prix DECIMAL(7,2))
4 BEGIN
5     SELECT p_prix + COALESCE(Race.prix, Espece.prix) INTO p_prix
6     FROM Animal
7     INNER JOIN Espece ON Espece.id = Animal.espece_id
8     LEFT JOIN Race ON Race.id = Animal.race_id
9     WHERE Animal.id = p_animal_id;
10 END |
11
12 DELIMITER ;
```

Et voici ce qu'affichera le code de test :

prix_intermediaire

485.00

prix_intermediaire

685.00

prix_intermediaire

1420.00

total

1430.00

Voilà qui devrait nous simplifier la vie. Et nous n'en sommes qu'au début des possibilités des procédures stockées !

Suppression d'une procédure

Vous commencez à connaître cette commande : pour supprimer une procédure, on utilise `DROP` (en précisant qu'il s'agit d'une procédure).

Exemple :

```
1 DROP PROCEDURE afficher_races;
```

Pour rappel, les procédures stockées ne sont pas détruites à la fermeture de la session, mais bien enregistrées comme un élément de la base de données exemple.

Notons encore qu'il n'est pas possible de modifier une procédure directement. La seule façon de modifier une procédure existante est de la supprimer puis de la recréer avec les modifications.

Il existe bien une commande `ALTER PROCEDURE`, mais elle ne permet de changer ni les paramètres ni le corps de la procédure. Elle permet uniquement de modifier certaines caractéristiques de la procédure, et ne sera pas couverte dans ce cours.

Avantages, inconvénients et usage des procédures stockées

Avantages

Les procédures stockées permettent de **réduire les allers-retours entre le client et le serveur MySQL**. En effet, si l'on englobe en une seule procédure l'exécution de plusieurs requêtes, le client ne communique qu'une seule fois avec le serveur (pour demander l'exécution de la procédure) pour exécuter la procédure. Cela permet donc un certain **gain en performance**.

Elles permettent également de **sécuriser** une base de données. Par exemple, il est possible de **restreindre les droits des utilisateurs** de façon à ce qu'ils ne puissent pas exécuter de requêtes dangereuses. On peut leur donner le droit d'exécuter des procédures, mais pas de modifier les données. Les commandes `DELETE` et `UPDATE` sont donc considérées comme dangereuses. Chaque requête exécutée par les utilisateurs est créée et contrôlée par l'administrateur par l'intermédiaire des procédures stockées.

Cela permet ensuite de **s'assurer qu'un traitement est toujours exécuté de la même manière**, quelle que soit l'application/le client qui le lance. Il arrive que des données soient exploitées par plusieurs applications, lesquelles peuvent être écrites avec différents langages. Si on laisse chaque application avoir son propre traitement, il est possible que des différences apparaissent (distraction, mauvaise communication, erreur ou autre). Par contre, si chaque application appelle une procédure, on s'assure que le traitement est toujours le même.

Inconvénients

Les procédures stockées **ajoutent évidemment à la charge du serveur de données**. Plus on implémente de logique de traitement directement dans la serveur est disponible pour son but premier : le stockage de données.

Par ailleurs, certains traitements seront toujours plus simples et plus courts à écrire (et donc à maintenir) s'ils sont développés dans un langage informatique s'agit de traitements complexes. **La logique qu'il est possible d'implémenter avec MySQL permet de nombreuses choses, mais reste assez basique.**

Enfin, **la syntaxe des procédures stockées diffère beaucoup d'un SGBD à un autre**. Par conséquent, si l'on désire en changer, il faudra procéder à un grand nombre d'ajustements.

Conclusion et usage

Comme souvent, tout est question d'**équilibre**. Il faut savoir utiliser des procédures quand c'est utile, quand on a une bonne raison de le faire. Il ne sert à rien d'en abuser. Pour une base contenant des données ultrasensibles, une bonne gestion des droits des utilisateurs couplée à l'usage de procédures stockées peut se révéler utile. Pour une base de données destinée à être utilisée par plusieurs applications différentes, on choisira de créer des procédures pour les traitements généraux. Une erreur peut poser de gros problèmes.

Pour un traitement long, impliquant de nombreuses requêtes et une logique simple, on peut sérieusement gagner en performance en le faisant dans une procédure (ce traitement est souvent lancé).

À vous de voir quelles procédures sont utiles pour **votre application et vos besoins**.

En résumé

- Une procédure stockée est un **ensemble d'instructions** que l'on peut exécuter sur commande.
- Une procédure stockée est un objet de la base de données **stocké de manière durable**, au même titre qu'une table. Elle n'est pas supprimée à la fin de la requête préparée.
- On peut passer des **paramètres** à une procédure stockée, qui peuvent avoir trois sens : **IN** (entrant), **OUT** (sortant) ou **INOUT** (les deux).
- `SELECT ... INTO` permet d'assigner des données sélectionnées à des variables ou des paramètres, à condition que le `SELECT` ne renvoie qu'une seule valeur sélectionnée que de variables à assigner.
- Les procédures stockées peuvent permettre de **gagner en performance** en diminuant les allers-retours entre le client et le serveur. Elles peuvent être utilisées sur une **base de données** et à s'assurer que les traitements sensibles sont toujours exécutés de la même manière.
- Par contre, elle **ajoute à la charge du serveur** et sa syntaxe n'est **pas toujours portable** d'un SGBD à un autre.

Que pensez-vous de ce cours ?

[REQUÊTES PRÉPARÉES](#)[STRUCTUREZ VOS INSTRUCTIONS](#)

Le professeur

Chantal Gribaumont

Senior Developer Diplômée de l'Université libre de Bruxelles en biologie

Découvrez aussi ce cours en...



Livre



PDF

OpenClassrooms

L'entreprise

Alternance

Forum

Blog

Nous rejoindre

Entreprises

Business

En plus


Devenez mentor

Aide et FAQ

Conditions Générales d'Utilisation

Politique de Protection des Données Personnelles

Nous contacter

 Français ▼

