Procédures stockées

Les procédures stockées sont disponibles depuis la version 5 de MySQL, et 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 de données** que l'on utilise, 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 alors exécutées.

Contrairement aux requêtes préparées, qui ne sont gardées en mémoire que pour la session courante, les procédures stockées sont, comme leur nom l'indique, **stockées de manière durable**, 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 :

Code : SQL

```
CREATE PROCEDURE nom_procedure ([parametre1 [, parametre2, ...]])
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 nouvelle procédure.
- `([parametre1 [, parametre2, ...]])` : après le nom de la procédure viennent des parenthèses. **Celles-ci sont obligatoires** ! À l'intérieur de ces parenthèses, on définit les éventuels 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 seule requête**, soit **un bloc 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.

Code : SQL

```
CREATE PROCEDURE afficher_races_requete() -- pas de paramètres dans
les parenthèses
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`.

Code : SQL

```
BEGIN
    -- Série d'instructions
END;
```

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

Code : SQL

```
CREATE PROCEDURE afficher_races_bloc() -- pas de paramètres dans les
parenthèses
BEGIN
    SELECT id, nom, espece_id, prix FROM Race;
END;
```

Malheureusement…

Code : Console

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that cor
```

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, et pourtant, nous avons 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 la suite de `SELECT * FROM Race;`. Cela semble logique, mais pose problème puisque c'est le premier ; rencontré par l'instruction **CREATE PROCEDURE**, qui naturellement pense devoir s'arrêter là. 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 qu'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 caractère ; pourra être utilisé à l'intérieur d'une instruction, et donc pourra être utilisé dans le corps d'une procédure stockée.

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

Code : SQL

```
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.

Code : SQL

```
SELECT 'test'|
```

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

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

Les deux délimiteurs suivants sont les plus couramment utilisés :

Code : SQL

```
DELIMITER //
DELIMITER |
```

Bien ! Ceci étant réglé, reprenons !

Création d'une procédure stockée

Code : SQL

```
DELIMITER |                              -- On change le délimiteur
CREATE PROCEDURE afficher_races()        -- toujours pas de
paramètres, toujours des parenthèses
BEGIN
    SELECT id, nom, espece_id, prix
    FROM Race;                           -- Cette fois, le ; ne nous
embêtera pas
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.

Lorsqu'on utilisera la procédure, quel que soit le délimiteur défini par **DELIMITER**, les instructions à l'intérieur du corps de la procédure seront bien délimitées par ;. En effet, 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 interprétera toujours ; comme 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 la procédure.

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 du nom de la procédure appelée, puis de parenthèses (avec éventuellement des paramètres).

Code : SQL

```
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	Rottweiller	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 pendant la procédure (pour un calcul ou une sélection par exemple).
- **OUT** : il s'agit d'un paramètre "sortant", dont la valeur va être 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.

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.

Code : SQL

```
DELIMITER |                                                    --
Facultatif si votre délimiteur est toujours |
CREATE PROCEDURE afficher_race_selon_espece (IN p_espece_id INT)   --
Définition du paramètre p_espece_id
BEGIN
    SELECT id, nom, espece_id, prix
    FROM Race
    WHERE espece_id = p_espece_id;                             --
Utilisation du paramètre
END |
```

```
DELIMITER ;                                                    --
On remet le délimiteur par défaut
```

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

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

Code : SQL

```
CALL afficher_race_selon_espece(1);
SET @espece_id := 2;
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	Rottweiller	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 afin de les distinguer facilement. Si vous ne le faites pas, soyez extrêmement prudents avec les noms que vous leur donnez. Par exemple, dans cette procédure, si on avait nommé le paramètre *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 dont on se sert dans la 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 par exemple, ce qui est toujours 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 stocke cette valeur dans un paramètre sortant.

Code : SQL

```
DELIMITER |
CREATE PROCEDURE compter_races_selon_espece (p_espece_id INT, OUT
p_nb_races INT)
BEGIN
    SELECT COUNT(*) INTO p_nb_races
```

```
        FROM Race
        WHERE espece_id = p_espece_id;
    END |
    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`. Voila qui est nouveau ! Comme vous l'avez sans doute deviné, le mot-clé `INTO` placé après la clause `SELECT` permet d'**assigner les valeurs sélectionnées** 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 à assigner **soient égaux** :

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

Code : SQL

```
SELECT id, nom INTO @var1, @var2
FROM Animal
WHERE id = 7;
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*, que nous 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).

Code : SQL

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

Code : Console

```
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.

Code : SQL

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

Code : Console

```
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_espece_id* et *p_nb_races*.
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 procédure *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 la procédure (grâce au **SELECT ... INTO**), et qu'on pourra utiliser ensuite : **une variable utilisateur** !

Code : SQL

```
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.

Code : SQL

```
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 (paramètre **IN**), et le prix à payer (paramètre **INOUT**).
La raison pour laquelle le prix est un paramètre à la fois entrant et sortant est qu'on veut pouvoir, avec cette procédure, calculer simplement un prix total dans le cas où un client 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 total le prix de l'animal en 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 d'essayer ensuite d'écrire vous-mêmes la procédure correspondante avant de regarder à quoi elle ressemble.

Code : SQL

```
SET @prix = 0;                    -- On initialise @prix à 0

CALL calculer_prix (13, @prix);   -- Achat de Rouquine
SELECT @prix AS prix_intermediaire;

CALL calculer_prix (24, @prix);   -- Achat de Cartouche
SELECT @prix AS prix_intermediaire;

CALL calculer_prix (42, @prix);   -- Achat de Bilba
SELECT @prix AS prix_intermediaire;

CALL calculer_prix (75, @prix);   -- Achat de Mimi
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 aider à écrire vous-mêmes la 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 :

Secret (cliquez pour afficher)

Code : SQL

```
DELIMITER |

CREATE PROCEDURE calculer_prix (IN p_animal_id INT, INOUT p_prix
DECIMAL(7,2))
BEGIN
    SELECT p_prix + COALESCE(Race.prix, Espece.prix) INTO p_prix
    FROM Animal
    INNER JOIN Espece ON Espece.id = Animal.espece_id
    LEFT JOIN Race ON Race.id = Animal.race_id
    WHERE Animal.id = p_animal_id;
END |

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 :

Code : SQL

```
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, au même titre qu'une table par 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 changer 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 un processus demandant 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 totalité du traitement. 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 puissent **uniquement exécuter des procédures**. Finis les `DELETE` dangereux ou les `UPDATE` inconsidérés. Chaque requête exécutée par les utilisateurs est créée et contrôlée par l'administrateur de la base de données 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 par exemple qu'une même base de données soit exploitée par plusieurs applications, lesquelles peuvent être écrites avec différents langages. Si on laisse chaque application avoir son propre code pour un même traitement, il est possible que des différences apparaissent (distraction, mauvaise communication, erreur ou autre). Par contre, si chaque application appelle la même procédure stockée, ce risque disparaît.

Inconvénients

Les procédures stockées **ajoutent évidemment à la charge sur le serveur de données**. Plus on implémente de logique de traitement directement dans la base de données, moins le 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 adapté. A fortiori lorsqu'il 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 de corrections et 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 salutaire.
Pour une base de données destinée à être utilisée par plusieurs applications différentes, on choisira de créer des procédures pour les traitements généraux et/ou pour lesquels la moindre 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 stockée (a fortiori si 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 session comme l'est une 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 ligne, et qu'il y ait autant de valeurs sélectionnées 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 également aider à **sécuriser une base de données** et à s'assurer que les traitements sensibles soient

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.

Structurer ses instructions

Lorsque l'on écrit une série d'instructions, par exemple dans le corps d'une procédure stockée, il est nécessaire d'être capable de structurer ses instructions. Cela va permettre d'instiller de la **logique dans le traitement** : exécuter telles ou telles instructions en fonction des données que l'on possède, répéter une instruction un certain nombre de fois, etc.

Voici quelques outils indispensables à la structuration des instructions :

- les **variables locales** : qui vont permettre de **stocker et modifier des valeurs** pendant le déroulement d'une procédure ;
- les **conditions** : qui vont permettre d'exécuter certaines instructions seulement **si une certaine condition est remplie** ;
- les **boucles** : qui vont permettre de **répéter une instruction** plusieurs fois.

Ces structures sont bien sûr utilisables dans les procédures stockées, que nous avons vues au chapitre précédent, mais pas uniquement. Elles sont **utilisables dans tout objet définissant une série d'instructions à exécuter**. C'est le cas des **fonctions stockées** (non couvertes par ce cours et qui forment avec les procédures stockées ce qu'on appelle les "routines"), des événements (non couverts), et également des **triggers**, auxquels un chapitre est consacré à la fin de cette partie.

Blocs d'instructions et variables locales

Blocs d'instructions

Nous avons vu qu'un bloc d'instructions était défini par les mots-clés `BEGIN` et `END`, entre lesquels on met les instructions qui composent le bloc (de zéro à autant d'instructions que l'on veut, séparées bien sûr d'un ;).

Il est possible d'imbriquer plusieurs blocs d'instructions. De même, à l'intérieur d'un bloc d'instructions, plusieurs blocs d'instructions peuvent se suivre. Ceux-ci permettent donc de **structurer les instructions** en plusieurs parties distinctes et sur plusieurs niveaux d'imbrication différents.

Code : SQL

```
BEGIN
    SELECT 'Bloc d''instructions principal';

    BEGIN
        SELECT 'Bloc d''instructions 2, imbriqué dans le bloc
principal';

        BEGIN
            SELECT 'Bloc d''instructions 3, imbriqué dans le bloc
d''instructions 2';
        END;
    END;

    BEGIN
        SELECT 'Bloc d''instructions 4, imbriqué dans le bloc
principal';
    END;

END;
```

Cet exemple montre également l'importance de l'**indentation** pour avoir un code lisible. Ici, toutes les instructions d'un bloc sont au même niveau et décalées vers la droite par rapport à la déclaration du bloc. Cela permet de voir en un coup d'œil où commence et où se termine chaque bloc d'instructions.

Variables locales

Nous connaissons déjà les variables utilisateur, qui sont des variables désignées par @. J'ai également mentionné l'existence des variables système, qui sont des variables prédéfinies par MySQL.
Voyons maintenant les **variables locales**, qui peuvent être définies dans un bloc d'instructions.

Déclaration d'une variable locale