V.4. 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.

V.4.1. 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 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**.

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

V.4.1.1. 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() -- pas de paramètres dans
      les parenthèses
2 SELECT id, nom, espece_id, prix FROM Race;
```

V.4.1.2. 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() -- pas de paramètres dans
      les parenthèses
2 BEGIN
3     SELECT id, nom, espece_id, prix FROM Race;
4 END;
```

Malheureusement…

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

? 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 !

V.4.1.3. 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 :

```
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

i `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 :

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

Bien ! Ceci étant réglé, reprenons !

V.4.1.4. Création d'une procédure stockée

```
1  DELIMITER |                                  -- On change le délimiteur
2  CREATE PROCEDURE afficher_races()      -- toujours pas de
       paramètres, toujours des parenthèses
3  BEGIN
4      SELECT id, nom, espece_id, prix
5      FROM Race;                               -- Cette fois, le ; ne nous
           embêtera pas
6  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.

> *i* 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.

V.4.1.5. 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).

```
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	Rottweiller	1	600.00

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

V.4.2. Les paramètres d'une procédure stockée

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

V.4.2.1. 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.

V.4.2.2. 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)`,…

V.4.2.3. Exemples

V.4.2.3.1. 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.

```
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.

```
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.

V.4.2.3.2. 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.

```
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`. 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

```
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*, 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).

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

```
1 ERROR 1222 (21000): The used SELECT statements have a different number of colu
```

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;
```

```
1 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** !

```
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.

```sql
SELECT @nb_races_chats;
```

@nb_races_chats
5

V.4.2.3.3. 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.

```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 :

Contenu masqué n°49

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 !

V.4.3. 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, 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.

V.4.4. Avantages, inconvénients et usage des procédures stockées

V.4.4.1. 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.

V.4.4.2. 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.

V.4.4.3. 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**.

V.4.4.4. 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.

Contenu masqué

Contenu masqué n°49

```
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 ;
```

Retourner au texte.

V.5. 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.

V.5.1. Blocs d'instructions et variables locales

V.5.1.1. 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.

```
1 BEGIN
2     SELECT 'Bloc d''instructions principal';
3
4     BEGIN
5         SELECT
              'Bloc d''instructions 2, imbriqué dans le bloc principal';
```

```
6
7           BEGIN
8               SELECT
                    'Bloc d''instructions 3, imbriqué dans le bloc d''instructions
9           END;
10      END;
11
12      BEGIN
13          SELECT
                'Bloc d''instructions 4, imbriqué dans le bloc principal';
14      END;
15
16  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.

V.5.1.2. 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.

V.5.1.2.1. Déclaration d'une variable locale

La déclaration d'une variable locale se fait avec l'instruction `DECLARE` :

```
1  DECLARE nom_variable type_variable [DEFAULT valeur_defaut];
```

Cette instruction doit se trouver au tout début du bloc d'instructions dans lequel la variable locale sera utilisée (donc directement après le `BEGIN`).

On a donc une structure générale des blocs d'instructions qui se dégage :

```
1  BEGIN
2      -- Déclarations (de variables locales par exemple)
3
4      -- Instructions (dont éventuels blocs d'instructions imbriqués)
```

```
5 END;
```

Tout comme pour les variables utilisateur, le nom des variables locales n'est **pas** sensible à la casse.

Si aucune valeur par défaut n'est précisée, la variable vaudra `NULL` tant que sa valeur n'est pas changée. Pour changer la valeur d'une variable locale, on peut utiliser `SET` ou `SELECT ... INTO`.

Exemple : voici une procédure stockée qui donne la date d'aujourd'hui et de demain :

```
1  DELIMITER |
2  CREATE PROCEDURE aujourdhui_demain ()
3  BEGIN
4      DECLARE v_date DATE DEFAULT CURRENT_DATE();               -- On
           déclare une variable locale et on lui met une valeur par
           défaut
5
6      SELECT DATE_FORMAT(v_date, '%W %e %M %Y') AS Aujourdhui;
7
8      SET v_date = v_date + INTERVAL 1 DAY;                     -- On
           change la valeur de la variable locale
9      SELECT DATE_FORMAT(v_date, '%W %e %M %Y') AS Demain;
10 END|
11 DELIMITER ;
```

Testons-la :

```
1 SET lc_time_names = 'fr_FR';
2 CALL aujourdhui_demain();
```

Aujourdhui
mardi 1 mai 2012

Demain
mercredi 2 mai 2012

Tout comme pour les paramètres, les variables locales peuvent poser problème si l'on ne fait pas attention au nom qu'on leur donne. En cas de conflit (avec un nom de colonne par

exemple), comme pour les paramètres, le nom sera interprété comme désignant la variable locale en priorité. Par conséquent, toutes mes variables locales seront préfixées par "v_".

V.5.1.2.2. Portée des variables locales dans un bloc d'instruction

Les variables locales n'existent que dans le bloc d'instructions dans lequel elles ont été déclarées. Dès que le mot-clé `END` est atteint, toutes les variables locales du bloc sont détruites.

Exemple 1 :

```
1  DELIMITER |
2  CREATE PROCEDURE test_portee1()
3  BEGIN
4      DECLARE v_test1 INT DEFAULT 1;
5
6      BEGIN
7          DECLARE v_test2 INT DEFAULT 2;
8
9          SELECT 'Imbriqué' AS Bloc;
10         SELECT v_test1, v_test2;
11     END;
12     SELECT 'Principal' AS Bloc;
13     SELECT v_test1, v_test2;
14
15 END|
16 DELIMITER ;
17
18 CALL test_portee1();
```

Bloc
Imbriqué

v_test1	v_test2
1	2

Bloc
Principal

```
1  ERROR 1054 (42S22): Unknown column 'v_test2' in 'field list'
```

La variable locale *v_test2* existe bien dans le bloc imbriqué, puisque c'est là qu'elle est définie, mais pas dans le bloc principal. *v_test1* par contre existe dans le bloc principal (où elle est définie), mais aussi dans le bloc imbriqué.

Exemple 2 :

```
DELIMITER |
CREATE PROCEDURE test_portee2()
BEGIN
    DECLARE v_test1 INT DEFAULT 1;

    BEGIN
        DECLARE v_test2 INT DEFAULT 2;

        SELECT 'Imbriqué 1' AS Bloc;
        SELECT v_test1, v_test2;
    END;

    BEGIN
        SELECT 'imbriqué 2' AS Bloc;
        SELECT v_test1, v_test2;
    END;


END|
DELIMITER ;

CALL test_portee2();
```

Bloc
Imbriqué 1

v_test1	v_test2
1	2

Bloc
imbriqué 2

```
ERROR 1054 (42S22): Unknown column 'v_test2' in 'field list'
```

À nouveau, *v_test1*, déclarée dans le bloc principal, existe dans les deux blocs imbriqués. Par

contre, *v_test2* n'existe que dans le bloc imbriqué dans lequel elle est déclarée.

Attention cependant à la subtilité suivante : si un bloc imbriqué déclare une variable locale ayant le même nom qu'une variable locale déclarée dans un bloc d'un niveau supérieur, il s'agira toujours de deux variables locales différentes, et seule la variable locale déclarée dans le bloc imbriqué sera visible dans ce même bloc.

Exemple 3 :

```
DELIMITER |
CREATE PROCEDURE test_portee3()
BEGIN
    DECLARE v_test INT DEFAULT 1;

    SELECT v_test AS 'Bloc principal';

    BEGIN
        DECLARE v_test INT DEFAULT 0;

        SELECT v_test AS 'Bloc imbriqué';
        SET v_test = 2;
        SELECT v_test AS 'Bloc imbriqué après modification';
    END;

    SELECT v_test AS 'Bloc principal';
END |
DELIMITER ;

CALL test_portee3();
```

Bloc principal
1

Bloc imbriqué
0

Bloc imbriqué après modification
2

Bloc principal

```
1
```

La variable locale *v_test* est déclarée dans le bloc principal et dans le bloc imbriqué, avec deux valeurs différentes. Mais lorsqu'on revient dans le bloc principal après exécution du bloc d'instructions imbriqué, *v_test* a toujours la valeur qu'elle avait avant l'exécution de ce bloc et sa deuxième déclaration. Il s'agit donc bien de **deux variables locales distinctes**.

V.5.2. Structures conditionnelles

Les structures conditionnelles permettent de déclencher une action ou une série d'instructions lorsqu'une condition préalable est remplie.

MySQL propose deux structures conditionnelles : `IF` et `CASE`.

V.5.2.1. La structure IF

Voici la syntaxe de la structure `IF` :

```
IF condition THEN instructions
[ELSEIF autre_condition THEN instructions
[ELSEIF ...]]
[ELSE instructions]
END IF;
```

V.5.2.1.1. Le cas le plus simple : si la condition est vraie, alors on exécute ces instructions

Voici la structure minimale d'un `IF` :

```
IF condition THEN
    instructions
END IF;
```

Soit on exécute les instructions (si la condition est vraie), soit on ne les exécute pas.

Exemple : la procédure suivante affiche `'J''ai déjà été adopté !'`, si c'est le cas, à partir de l'*id* d'un animal :

```
DELIMITER |
CREATE PROCEDURE est_adopte(IN p_animal_id INT)
```

```
3  BEGIN
4      DECLARE v_nb INT DEFAULT 0;           -- On crée une variable
           locale
5
6      SELECT COUNT(*) INTO v_nb             -- On met le nombre de
           lignes correspondant à l'animal
7      FROM Adoption                         --  dans Adoption dans
           notre variable locale
8      WHERE animal_id = p_animal_id;
9
10     IF v_nb > 0 THEN                      -- On teste si v_nb est
           supérieur à 0 (donc si l'animal a été adopté)
11         SELECT 'J''ai déjà été adopté !';
12     END IF;                               -- Et on n'oublie surtout
           pas le END IF et le ; final
13 END |
14 DELIMITER ;
15
16 CALL est_adopte(3);
17 CALL est_adopte(28);
```

Seul le premier appel à la procédure va afficher `'J''ai déjà été adopté !'`, puisque l'animal 3 est présent dans la table *Adoption*, contrairement à l'animal 28.

V.5.2.1.2. Deuxième cas : si ... alors, sinon ...

Grâce au mot-clé `ELSE`, on peut définir une série d'instructions à exécuter si la condition est fausse.

> `ELSE` ne doit **pas** être suivi de `THEN`.

Exemple : la procédure suivante affiche `'Je suis né avant 2010'` ou `'Je suis né après 2010'`, selon la date de naissance de l'animal transmis en paramètre.

```
1  DELIMITER |
2  CREATE PROCEDURE avant_apres_2010(IN p_animal_id INT)
3  BEGIN
4      DECLARE v_annee INT;
5
6      SELECT YEAR(date_naissance) INTO v_annee
7      FROM Animal
8      WHERE id = p_animal_id;
9
10     IF v_annee < 2010 THEN
11         SELECT 'Je suis né avant 2010' AS naissance;
```

```
    ELSE                                                    -- Pas de
        THEN
        SELECT 'Je suis né après 2010' AS naissance;
    END IF;                                                 -- Toujours
        obligatoire

END |
DELIMITER ;

CALL avant_apres_2010(34);   -- Né le 20/04/2008
CALL avant_apres_2010(69);   -- Né le 13/02/2012
```

V.5.2.1.3. Troisième et dernier cas : plusieurs conditions alternatives

Enfin, le mot-clé `ELSEIF... THEN` permet de vérifier d'autres conditions (en dehors de la condition du `IF`), chacune ayant une série d'instructions définies à exécuter en cas de véracité. Si plusieurs conditions sont vraies en même temps, seule la première rencontrée verra ses instructions exécutées. On peut bien sûr toujours (mais ce n'est pas obligatoire) ajouter un `ELSE` pour le cas où aucune condition ne serait vérifiée.

Exemple : cette procédure affiche un message différent selon le sexe de l'animal passé en paramètre.

```
DELIMITER |
CREATE PROCEDURE message_sexe(IN p_animal_id INT)
BEGIN
    DECLARE v_sexe VARCHAR(10);

    SELECT sexe INTO v_sexe
    FROM Animal
    WHERE id = p_animal_id;

    IF (v_sexe = 'F') THEN                    -- Première
        possibilité
        SELECT 'Je suis une femelle !' AS sexe;
    ELSEIF (v_sexe = 'M') THEN                  -- Deuxième
        possibilité
        SELECT 'Je suis un mâle !' AS sexe;
    ELSE                                          -- Défaut
        SELECT 'Je suis en plein questionnement existentiel...' AS
            sexe;
    END IF;
END|
DELIMITER ;

CALL message_sexe(8);   -- Mâle
CALL message_sexe(6);   -- Femelle
```

```
22 CALL message_sexe(9);   -- Ni l'un ni l'autre
```

Il peut bien sûr y avoir autant de `ELSEIF... THEN` que l'on veut (mais un seul `ELSE`).

V.5.2.2. La structure CASE

Deux syntaxes sont possibles pour utiliser `CASE`.

V.5.2.2.1. Première syntaxe : conditions d'égalité

```
1 CASE valeur_a_comparer
2     WHEN possibilite1 THEN instructions
3     [WHEN possibilite2 THEN instructions] ...
4     [ELSE instructions]
5 END CASE;
```

Exemple : on reprend la procédure *message_sexe()*, et on l'adapte pour utiliser `CASE`.

```
1  DELIMITER |
2  CREATE PROCEDURE message_sexe2(IN p_animal_id INT)
3  BEGIN
4      DECLARE v_sexe VARCHAR(10);
5
6      SELECT sexe INTO v_sexe
7      FROM Animal
8      WHERE id = p_animal_id;
9
10     CASE v_sexe
11         WHEN 'F' THEN                          -- Première
               possibilité
12             SELECT 'Je suis une femelle !' AS sexe;
13         WHEN 'M' THEN                              -- Deuxième
               possibilité
14             SELECT 'Je suis un mâle !' AS sexe;
15         ELSE                                       -- Défaut
16             SELECT 'Je suis en plein questionnement existentiel...'
                   AS sexe;
17     END CASE;
18 END|
19 DELIMITER ;
20
21 CALL message_sexe2(8);   -- Mâle
22 CALL message_sexe2(6);   -- Femelle
```

```
23 CALL message_sexe2(9);   -- Ni l'un ni l'autre
```

On définit donc *v_sexe* comme point de comparaison. Chaque `WHEN` donne alors un élément auquel *v_sexe* doit être comparé. Les instructions exécutées seront celles du `WHEN` dont l'élément est égal à *v_sexe*. Le `ELSE` sera exécuté si aucun `WHEN` ne correspond.

Ici, on compare une variable locale (*v_sexe*) à des chaînes de caractères (`'F'` et `'M'`), mais on peut utiliser différents types d'éléments. Voici les principaux :

— des variables locales ;
— des variables utilisateur ;
— des valeurs constantes de tous types (`0`, `'chaîne'`, `5.67`, `'2012-03-23'`,…) ;
— des expressions (`2 + 4`, `NOW()`, `CONCAT(nom, ' ', prenom)`,…) ;
— …

Cette syntaxe ne permet pas de faire des comparaisons avec `NULL`, puisqu'elle utilise une comparaison de type `valeur1 = valeur2`. Or cette comparaison est inutilisable dans le cas de `NULL`. Il faudra donc utiliser la seconde syntaxe, avec le test `IS NULL`.

V.5.2.2.2. Seconde syntaxe : toutes conditions

Cette seconde syntaxe ne compare pas un élément à différentes valeurs, mais utilise simplement des conditions classiques et permet donc de faire des comparaisons de type "plus grand que", "différent de", etc. (bien entendu, elle peut également être utilisée pour des égalités).

```
1 CASE
2     WHEN condition THEN instructions
3     [WHEN condition THEN instructions] ...
4     [ELSE instructions]
5 END CASE
```

Exemple : on reprend la procédure *avant_apres_2010()*, qu'on réécrit avec `CASE`, et en donnant une possibilité en plus. De plus, on passe le message en paramètre `OUT` pour changer un peu.

```
1 DELIMITER |
2 CREATE PROCEDURE avant_apres_2010_case (IN p_animal_id INT, OUT
      p_message VARCHAR(100))
3 BEGIN
4     DECLARE v_annee INT;
5
6     SELECT YEAR(date_naissance) INTO v_annee
7     FROM Animal
8     WHERE id = p_animal_id;
```

```
9
10      CASE
11          WHEN v_annee < 2010 THEN
12              SET p_message = 'Je suis né avant 2010.';
13          WHEN v_annee = 2010 THEN
14              SET p_message = 'Je suis né en 2010.';
15          ELSE
16              SET p_message = 'Je suis né après 2010.';
17      END CASE;
18  END |
19  DELIMITER ;
20
21  CALL avant_apres_2010_case(59, @message);
22  SELECT @message;
23  CALL avant_apres_2010_case(62, @message);
24  SELECT @message;
25  CALL avant_apres_2010_case(69, @message);
26  SELECT @message;
```

V.5.2.2.3. Comportement particulier : aucune correspondance trouvée

En l'absence de clause `ELSE`, si aucune des conditions posées par les différentes clauses `WHEN` n'est remplie (quelle que soit la syntaxe utilisée), une erreur est déclenchée.

Par exemple, cette procédure affiche une salutation différente selon la terminaison du nom de l'animal passé en paramètre :

```
1   DELIMITER |
2   CREATE PROCEDURE salut_nom(IN p_animal_id INT)
3   BEGIN
4       DECLARE v_terminaison CHAR(1);
5
6       SELECT SUBSTRING(nom, -1, 1) INTO v_terminaison  -- Une
            position négative signifie qu'on recule au lieu d'avancer.
7       FROM Animal                                      --  -1 est
            donc la dernière lettre du nom.
8       WHERE id = p_animal_id;
9
10      CASE v_terminaison
11          WHEN 'a' THEN
12              SELECT 'Bonjour !' AS Salutations;
13          WHEN 'o' THEN
14              SELECT 'Salut !' AS Salutations;
15          WHEN 'i' THEN
16              SELECT 'Coucou !' AS Salutations;
17      END CASE;
18
```

```
19 END|
20 DELIMITER ;
21
22 CALL salut_nom(69);  -- Baba
23 CALL salut_nom(5);   -- Choupi
24 CALL salut_nom(29);  -- Fiero
25 CALL salut_nom(54);  -- Bubulle
```

Salutations
Bonjour !

Salutations
Coucou !

Salutations
Salut !

```
1 ERROR 1339 (20000): Case not found for CASE statement
```

L'appel de la procédure avec Bubulle présente un cas qui n'est pas couvert par les trois `WHEN`. Une erreur est donc déclenchée

Donc, si l'on n'est pas sûr d'avoir couvert tous les cas possibles, il faut toujours ajouter une clause `ELSE` pour éviter les erreurs. Si l'on veut qu'aucune instruction ne soit exécutée par le `ELSE`, il suffit simplement de mettre un bloc d'instructions vide (`BEGIN END;`).

Exemple : reprenons la procédure *salut_nom()*, et ajoutons-lui une clause `ELSE` vide :

```
1  DROP PROCEDURE salut_nom;
2  DELIMITER |
3  CREATE PROCEDURE salut_nom(IN p_animal_id INT)
4  BEGIN
5      DECLARE v_terminaison CHAR(1);
6
7      SELECT SUBSTRING(nom, -1, 1) INTO v_terminaison
8      FROM Animal
9      WHERE id = p_animal_id;
10
11     CASE v_terminaison
12         WHEN 'a' THEN
```

```
13              SELECT 'Bonjour !' AS Salutations;
14          WHEN 'o' THEN
15              SELECT 'Salut !' AS Salutations;
16          WHEN 'i' THEN
17              SELECT 'Coucou !' AS Salutations;
18          ELSE
19              BEGIN                                   -- Bloc
                    d'instructions vide
20              END;
21      END CASE;
22
23  END|
24  DELIMITER ;
25
26  CALL salut_nom(69);   -- Baba
27  CALL salut_nom(5);    -- Choupi
28  CALL salut_nom(29);   -- Fiero
29  CALL salut_nom(54);   -- Bubulle
```

Cette fois, pas d'erreur. Le dernier appel (avec Bubulle) n'affiche simplement rien.

Il faut au minimum une instruction ou un bloc d'instructions par clause `WHEN` et par clause `ELSE`. Un bloc vide `BEGIN END;` est donc nécessaire si l'on ne veut rien exécuter.

V.5.2.3. Utiliser une structure conditionnelle directement dans une requête

Jusqu'ici, on a vu l'usage des structures conditionnelles dans des procédures stockées. Il est cependant possible d'utiliser une structure `CASE` dans une simple requête.

Par exemple, écrivons une requête `SELECT` suivant le même principe que la procédure *message_sexe()* :

```
1  SELECT id, nom, CASE
2            WHEN sexe = 'M' THEN 'Je suis un mâle !'
3            WHEN sexe = 'F' THEN 'Je suis une femelle !'
4            ELSE 'Je suis en plein questionnement existentiel...'
5         END AS message
6  FROM Animal
7  WHERE id IN (9, 8, 6);
```

id	nom	message
6	Bobosse	Je suis une femelle !
8	Bagherra	Je suis un mâle !

9	NULL	Je suis en plein questionnement existentiel...

Quelques remarques :

— On peut utiliser les deux syntaxes de `CASE`.
— Il faut clôturer le `CASE` par `END`, et non par `END CASE` (et bien sûr ne pas mettre de `;` si la requête n'est pas finie).
— Ce n'est pas limité aux clauses `SELECT`, on peut tout à fait utiliser un `CASE` dans une clause `WHERE` par exemple.
— Ce n'est par conséquent pas non plus limité aux requêtes `SELECT`, on peut l'utiliser dans n'importe quelle requête.

Il n'est par contre pas possible d'utiliser une structure `IF` dans une requête. Cependant, il existe une **fonction** `IF()`, beaucoup plus limitée, dont la syntaxe est la suivante :

```
IF(condition, valeur_si_vrai, valeur_si_faux)
```

Exemple :

```
SELECT nom, IF(sexe = 'M', 'Je suis un mâle',
    'Je ne suis pas un mâle') AS sexe
FROM Animal
WHERE espece_id = 5;
```

nom	sexe
Baba	Je ne suis pas un mâle
Bibo	Je suis un mâle
Momy	Je ne suis pas un mâle
Popi	Je suis un mâle
Mimi	Je ne suis pas un mâle

V.5.3. Boucles

Une boucle est une structure qui permet de répéter plusieurs fois une série d'instructions. Il existe trois types de boucles en MySQL : `WHILE`, `LOOP` et `REPEAT`.

V.5.3.1. La boucle WHILE

La boucle `WHILE` permet de répéter une série d'instructions **tant que la condition donnée reste vraie**.

```
WHILE condition DO    -- Attention de ne pas oublier le DO, erreur classique
    instructions
END WHILE;
```

Exemple : la procédure suivante affiche les nombres entiers de 1 à *p_nombre* (passé en paramètre).

```
DELIMITER |
CREATE PROCEDURE compter_jusque_while(IN p_nombre INT)
BEGIN
    DECLARE v_i INT DEFAULT 1;

    WHILE v_i <= p_nombre DO
        SELECT v_i AS nombre;

        SET v_i = v_i + 1;    -- À ne surtout pas oublier, sinon la condition restera vraie
    END WHILE;
END |
DELIMITER ;

CALL compter_jusque_while(3);
```

Vérifiez que votre condition devient bien fausse après un certain nombre d'itérations de la boucle. Sinon, vous vous retrouvez avec une boucle infinie (qui ne s'arrête jamais).

V.5.3.2. La boucle REPEAT

La boucle `REPEAT` travaille en quelque sorte de manière opposée à `WHILE`, puisqu'elle exécute des instructions de la boucle **jusqu'à ce que la condition donnée devienne vraie**.

Exemple : voici la même procédure écrite avec une boucle `REPEAT`.

```
DELIMITER |
CREATE PROCEDURE compter_jusque_repeat(IN p_nombre INT)
BEGIN
```

```
4       DECLARE v_i INT DEFAULT 1;
5
6       REPEAT
7           SELECT v_i AS nombre;
8
9           SET v_i = v_i + 1;    -- À ne surtout pas oublier, sinon la
                condition restera vraie
10      UNTIL v_i > p_nombre END REPEAT;
11  END |
12  DELIMITER ;
13
14  CALL compter_jusque_repeat(3);
```

Attention, comme la condition d'une boucle `REPEAT` est vérifiée après le bloc d'instructions de la boucle, **on passe au moins une fois dans la boucle**, même si la condition est tout de suite fausse !

Test

```
1  -- Condition fausse dès le départ, on ne rentre pas dans la boucle
2  CALL compter_jusque_while(0);
3
4  -- Condition fausse dès le départ, on rentre quand même une fois
       dans la boucle
5  CALL compter_jusque_repeat(0);
```

V.5.3.3. Donner un label à une boucle

Il est possible de donner un label (un nom) à une boucle, ou à un bloc d'instructions défini par `BEGIN... END`. Il suffit pour cela de faire précéder l'ouverture de la boucle/du bloc par ce label, suivi de `:`.

La fermeture de la boucle/du bloc peut alors faire référence à ce label (mais ce n'est pas obligatoire).

Un label ne peut pas dépasser 16 caractères.

Exemples

```
1  -- Boucle WHILE
2  -- ------------
```

```
super_while: WHILE condition DO    -- La boucle a pour label "super_while"
    instructions
END WHILE super_while;             -- On ferme en donnant le label de la boucle (facultatif)

-- Boucle REPEAT
-- -------------
repeat_genial: REPEAT              -- La boucle s'appelle "repeat_genial"
    instructions
UNTIL condition END REPEAT;        -- Cette fois, on choisit de ne pas faire référence au label lors de la fermeture

-- Bloc d'instructions
-- -------------------
bloc_extra: BEGIN                  -- Le bloc a pour label "bloc_extra"
    instructions
END bloc_extra;
```

? Mais en quoi cela peut-il être utile ?

D'une part, cela peut permettre de **clarifier le code** lorsqu'il y a beaucoup de boucles et de blocs d'instructions imbriqués. D'autre part, il est nécessaire de donner un label aux boucles et aux blocs d'instructions pour lesquels on veut pouvoir **utiliser les instructions `ITERATE` et `LEAVE`**.

V.5.3.4. Les instructions LEAVE et ITERATE

V.5.3.4.1. LEAVE : quitter la boucle ou le bloc d'instructions

L'instruction `LEAVE` peut s'utiliser **dans une boucle ou un bloc d'instructions** et **déclenche la sortie immédiate de la structure dont le label est donné**.

```
LEAVE label_structure;
```

Exemple : cette procédure incrémente de 1, et affiche, un nombre entier passé en paramètre. Et cela, 4 fois maximum. Mais si l'on trouve un multiple de 10, la boucle s'arrête.

```
DELIMITER |
CREATE PROCEDURE test_leave1(IN p_nombre INT)
BEGIN
```

```
4       DECLARE v_i INT DEFAULT 4;
5
6       SELECT 'Avant la boucle WHILE';
7
8       while1: WHILE v_i > 0 DO
9
10          SET p_nombre = p_nombre + 1;        -- On incrémente le
                nombre de 1
11
12          IF p_nombre%10 = 0 THEN              -- Si p_nombre est
                divisible par 10,
13              SELECT 'Stop !' AS 'Multiple de 10';
14              LEAVE while1;                   -- On quitte la boucle
                    WHILE.
15          END IF;
16
17          SELECT p_nombre;                    -- On affiche p_nombre
18          SET v_i = v_i - 1;                  -- Attention de ne pas
                l'oublier
19
20      END WHILE while1;
21
22      SELECT 'Après la boucle WHILE';
23  END|
24  DELIMITER ;
25
26  CALL test_leave1(3); -- La boucle s'exécutera 4 fois
```

Avant la boucle WHILE
Avant la boucle WHILE

p_nombre
4

p_nombre
5

p_nombre
6

p_nombre

7

Après la boucle WHILE
Après la boucle WHILE

```
CALL test_leave1(8); -- La boucle s'arrêtera dès qu'on atteint 10
```

Avant la boucle WHILE
Avant la boucle WHILE

p_nombre
9

Multiple de 10
Stop !

Après la boucle WHILE
Après la boucle WHILE

Il est par conséquent possible d'utiliser `LEAVE` pour provoquer la fin de la procédure stockée.

Exemple : voici la même procédure. Cette fois-ci un multiple de 10 provoque l'arrêt de toute la procédure, pas seulement de la boucle `WHILE`.

```
DELIMITER |
CREATE PROCEDURE test_leave2(IN p_nombre INT)
corps_procedure: BEGIN                          -- On donne un
    label au bloc d'instructions principal
    DECLARE v_i INT DEFAULT 4;

    SELECT 'Avant la boucle WHILE';
    while1: WHILE v_i > 0 DO
        SET p_nombre = p_nombre + 1;            -- On incrémente
            le nombre de 1
```

```
        IF p_nombre%10 = 0 THEN                    -- Si p_nombre est
            divisible par 10,
            SELECT 'Stop !' AS 'Multiple de 10';
            LEAVE corps_procedure;                 -- je quitte la
                procédure.
        END IF;

        SELECT p_nombre;                           -- On affiche
            p_nombre
        SET v_i = v_i - 1;                         -- Attention de ne
            pas l'oublier
    END WHILE while1;

    SELECT 'Après la boucle WHILE';
END|
DELIMITER ;

CALL test_leave2(8);
```

`'Après la boucle WHILE'` ne s'affiche plus lorsque l'instruction `LEAVE` est déclenchée, puisque l'on quitte la procédure stockée avant d'arriver à l'instruction `SELECT` qui suit la boucle `WHILE`.

En revanche, `LEAVE` ne permet pas de quitter directement une structure conditionnelle (`IF` ou `CASE`). Il n'est d'ailleurs pas non plus possible de donner un label à ces structures. Cette restriction est cependant aisément contournable en utilisant les blocs d'instructions.

Exemple : la procédure suivante affiche les nombres de 4 à 1, en précisant s'ils sont pairs. Sauf pour le nombre 2, pour lequel une instruction `LEAVE` empêche l'affichage habituel.

```
DELIMITER |
CREATE PROCEDURE test_leave3()
BEGIN
    DECLARE v_i INT DEFAULT 4;

    WHILE v_i > 0 DO

        IF v_i%2 = 0 THEN
            if_pair: BEGIN
                IF v_i = 2 THEN                    -- Si v_i vaut 2
                    LEAVE if_pair;                 -- On quitte le
                        bloc "if_pair", ce qui revient à quitter la
                        structure IF v_i%2 = 0
                END IF;
                SELECT CONCAT(v_i, ' est pair') AS message;
            END if_pair;
        ELSE
            if_impair: BEGIN
                SELECT CONCAT(v_i, ' est impair') AS message;
```

```
18              END if_impair;
19          END IF;
20
21          SET v_i = v_i - 1;
22      END WHILE;
23  END|
24  DELIMITER ;
25
26  CALL test_leave3();
```

message
4 est pair

message
3 est impair

message
1 est impair

`'2 est pair'` n'est pas affiché, puisqu'on a quitté le `IF` avant cet affichage.

V.5.3.4.2. ITERATE : déclencher une nouvelle itération de la boucle

Cette instruction ne peut être utilisée que dans une boucle. Lorsqu'elle est exécutée, une **nouvelle itération de la boucle commence**. Toutes les instructions suivant `ITERATE` dans la boucle sont ignorées.

Exemple : la procédure suivante affiche les nombres de 1 à 3, avec un message avant le `IF` et après le `IF`. Sauf pour le nombre 2, qui relance une itération de la boucle dans le `IF`.

```
1   DELIMITER |
2   CREATE PROCEDURE test_iterate()
3   BEGIN
4       DECLARE v_i INT DEFAULT 0;
5
6       boucle_while: WHILE v_i < 3 DO
7           SET v_i = v_i + 1;
8           SELECT v_i, 'Avant IF' AS message;
9
10          IF v_i = 2 THEN
11              ITERATE boucle_while;
```

```
12          END IF;
13
14          SELECT v_i, 'Après IF' AS message;  -- Ne sera pas exécuté
                pour v_i = 2
15      END WHILE;
16  END |
17  DELIMITER ;
18
19  CALL test_iterate();
```

v_i	message
1	Avant IF

v_i	message
1	Après IF

v_i	message
2	Avant IF

v_i	message
3	Avant IF

v_i	message
3	Après IF

Attention à ne pas faire de boucle infinie avec `ITERATE`, on oublie facilement que cette instruction empêche l'exécution de toutes les instructions qui la suivent dans la boucle. Si j'avais mis par exemple `SET v_i = v_i + 1;` après `ITERATE` et non avant, la boucle serait restée coincée à `v_i = 2`.

V.5.3.5. La boucle LOOP

On a gardé la boucle `LOOP` pour la fin, parce qu'elle est un peu particulière. En effet, voici sa syntaxe :

```
[label:] LOOP
    instructions
END LOOP [label]
```

Vous voyez bien : il n'est question de condition nulle part. En fait, une boucle `LOOP` doit intégrer dans ses instructions un élément qui va la faire s'arrêter : typiquement une instruction `LEAVE`. Sinon, c'est une boucle infinie.

Exemple : à nouveau une procédure qui affiche les nombres entiers de 1 à *p_nombre*.

```
DELIMITER |
CREATE PROCEDURE compter_jusque_loop(IN p_nombre INT)
BEGIN
    DECLARE v_i INT DEFAULT 1;

    boucle_loop: LOOP
        SELECT v_i AS nombre;

        SET v_i = v_i + 1;

        IF v_i > p_nombre THEN
            LEAVE boucle_loop;
        END IF;
    END LOOP;
END |
DELIMITER ;

CALL compter_jusque_loop(3);
```

V.5.3.6. En résumé

- Un bloc d'instructions est délimité par `BEGIN` et `END`. Il est possible d'imbriquer plusieurs blocs d'instructions.
- Une variable locale est définie dans un bloc d'instructions grâce à la commande `DECLARE`. Une fois la fin du bloc d'instructions atteinte, toutes les variables locales qui y ont été déclarées sont supprimées.
- Une structure conditionnelle permet d'exécuter une série d'instructions si une condition est respectée. Les deux structures conditionnelles de MySQL sont `IF` et `CASE`.
- Une boucle est une structure qui permet de répéter une série d'instructions un certain nombre de fois. Il existe trois types de boucle pour MySQL : `WHILE`, `REPEAT` et `LOOP`.
- L'instruction `LEAVE` permet de quitter un bloc d'instructions ou une boucle.
- L'instruction `ITERATE` permet de relancer une itération d'une boucle.