Administrez vos bases de données avec MySQL

40 heures Moyenne

Mis à jour le 03/09/2019

Structurez vos 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 :** elles vont permettre de **stocker et de modifier des valeurs** pendant le déroulement d'une procédure ;
- les **conditions :** elles vont permettre d'exécuter certaines instructions seulement **si une certaine condition est remplie ;**
- les **boucles :** elles 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 que l'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.

sql
```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

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

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

Cette instruction doit se trouver tout au 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 :

```sql
BEGIN
    -- Déclarations (de variables locales par exemple)

    -- Instructions (dont éventuels blocs d'instructions imbriqués)
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.

```sql
DELIMITER |
CREATE PROCEDURE aujourdhui_demain ()
BEGIN
    DECLARE v_date DATE DEFAULT CURRENT_DATE();
    -- On déclare une variable locale et on lui met une valeur par défaut

    SELECT DATE_FORMAT(v_date, '%W %e %M %Y') AS Aujourdhui;

    SET v_date = v_date + INTERVAL 1 DAY;
    -- On change la valeur de la variable locale
    SELECT DATE_FORMAT(v_date, '%W %e %M %Y') AS Demain;
END|
DELIMITER ;
```

Testons-la :

```sql
SET lc_time_names = 'fr_FR';
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_".

Portée des variables locales dans un bloc d'instructions

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 :

sql

```sql
DELIMITER |
CREATE PROCEDURE test_portee1()
BEGIN
    DECLARE v_test1 INT DEFAULT 1;

    BEGIN
        DECLARE v_test2 INT DEFAULT 2;

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

END|
DELIMITER ;

CALL test_portee1();
```

Bloc
Imbriqué

v_test1	v_test2
1	2

Bloc
Principal

```
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 :

sql

```sql
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 :

sql

```sql
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 lorsque l'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**.

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`.

La structure `IF`

Voici la syntaxe de la structure `IF` :

sql

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

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

Voici la structure minimale d'un `IF` :

sql

```sql
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.

sql

```sql
DELIMITER |
CREATE PROCEDURE est_adopte(IN p_animal_id INT)
BEGIN
    DECLARE v_nb INT DEFAULT 0;
    -- On crée une variable locale

    SELECT COUNT(*) INTO v_nb
    FROM Adoption
    WHERE animal_id = p_animal_id;
    -- On met le nombre de lignes correspondant à l'animal dans Adoption dans notre variable locale

    IF v_nb > 0 THEN
    -- On teste si v_nb est supérieur à 0 (donc si l'animal a été adopté)
        SELECT 'J''ai déjà été adopté !';
    END IF;
    -- Et on n'oublie surtout pas le END IF et le ; final
END |
DELIMITER ;
```

```
19
20 CALL est_adopte(3);
21 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.

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.

sql

```sql
DELIMITER |
CREATE PROCEDURE avant_apres_2010(IN p_animal_id INT)
BEGIN
    DECLARE v_annee INT;

    SELECT YEAR(date_naissance) INTO v_annee
    FROM Animal
    WHERE id = p_animal_id;

    IF v_annee < 2010 THEN
        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
```

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.

sql

```
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
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`).

La structure `CASE`

Deux syntaxes sont possibles pour utiliser `CASE`.

Première syntaxe : conditions d'égalité

sql

```
CASE valeur_a_comparer
    WHEN possibilite1 THEN instructions
    [WHEN possibilite2 THEN instructions] ...
    [ELSE instructions]
END CASE;
```

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

sql

```
DELIMITER |
CREATE PROCEDURE message_sexe2(IN p_animal_id INT)
BEGIN
    DECLARE v_sexe VARCHAR(10);

    SELECT sexe INTO v_sexe
    FROM Animal
    WHERE id = p_animal_id;

    CASE v_sexe
        WHEN 'F' THEN   -- Première possibilité
            SELECT 'Je suis une femelle !' AS sexe;
        WHEN 'M' THEN   -- Deuxième possibilité
            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` .

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).

sql

```
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()*, que l'on réécrit avec `CASE` , et en donnant une possibilité en plus. De plus, on passe le message en paramètre `OUT` pour changer un peu.

sql

```
1  DELIMITER |
2  CREATE PROCEDURE avant_apres_2010_case (IN p_animal_id INT, OUT p_message VARCHAR(100))
3  BEGIN
```

```
    DECLARE v_annee INT;

    SELECT YEAR(date_naissance) INTO v_annee
    FROM Animal
    WHERE id = p_animal_id;

    CASE
        WHEN v_annee < 2010 THEN
            SET p_message = 'Je suis né avant 2010.';
        WHEN v_annee = 2010 THEN
            SET p_message = 'Je suis né en 2010.';
        ELSE
            SET p_message = 'Je suis né après 2010.';
    END CASE;
END |
DELIMITER ;

CALL avant_apres_2010_case(59, @message);
SELECT @message;
CALL avant_apres_2010_case(62, @message);
SELECT @message;
CALL avant_apres_2010_case(69, @message);
SELECT @message;
```

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 :

sql

```
DELIMITER |
CREATE PROCEDURE salut_nom(IN p_animal_id INT)
BEGIN
    DECLARE v_terminaison CHAR(1);

    SELECT SUBSTRING(nom, -1, 1) INTO v_terminaison
    -- Une position négative signifie qu'on recule au lieu d'avancer, -1 est donc la dernière lettre du nom..
    FROM Animal
    WHERE id = p_animal_id;

    CASE v_terminaison
        WHEN 'a' THEN
            SELECT 'Bonjour !' AS Salutations;
        WHEN 'o' THEN
            SELECT 'Salut !' AS Salutations;
        WHEN 'i' THEN
            SELECT 'Coucou !' AS Salutations;
    END CASE;

END|
DELIMITER ;

```

```
CALL salut_nom(69);  -- Baba
CALL salut_nom(5);   -- Choupi
CALL salut_nom(29);  -- Fiero
CALL salut_nom(54);  -- Bubulle
```

Salutations
Bonjour !

Salutations
Coucou !

Salutations
Salut !

```
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 de mettre un bloc d'instructions vide (`BEGIN END;`).

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

sql

```sql
DROP PROCEDURE salut_nom;
DELIMITER |
CREATE PROCEDURE salut_nom(IN p_animal_id INT)
BEGIN
    DECLARE v_terminaison CHAR(1);

    SELECT SUBSTRING(nom, -1, 1) INTO v_terminaison
    FROM Animal
    WHERE id = p_animal_id;

    CASE v_terminaison
        WHEN 'a' THEN
            SELECT 'Bonjour !' AS Salutations;
        WHEN 'o' THEN
            SELECT 'Salut !' AS Salutations;
        WHEN 'i' THEN
```

```
            SELECT 'Coucou !' AS Salutations;
        ELSE
            BEGIN   -- Bloc d'instructions vide
            END;
    END CASE;

END|
DELIMITER ;

CALL salut_nom(69);  -- Baba
CALL salut_nom(5);   -- Choupi
CALL salut_nom(29);  -- Fiero
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.

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() :*

sql

```
SELECT id, nom, CASE
          WHEN sexe = 'M' THEN 'Je suis un mâle !'
          WHEN sexe = 'F' THEN 'Je suis une femelle !'
          ELSE 'Je suis en plein questionnement existentiel...'
       END AS message
FROM Animal
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 :

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

Exemple :

sql
```sql
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

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`.

La boucle `WHILE`

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

sql
```sql
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).

sql

```sql
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).

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`.

sql

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

    REPEAT
        SELECT v_i AS nombre;

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

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

```sql
-- Condition fausse dès le départ, on ne rentre pas dans la boucle
CALL compter_jusque_while(0);

-- Condition fausse dès le départ, on rentre quand même une fois dans la boucle
CALL compter_jusque_repeat(0);
```

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

```sql
-- Boucle WHILE
-- ------------
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`.

Les instructions `LEAVE` et `ITERATE`

`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**.

sql

```sql
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.

sql

```sql
DELIMITER |
CREATE PROCEDURE test_leave1(IN p_nombre INT)
BEGIN
    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 while1;   -- On quitte la boucle WHILE.
        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_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

sql

```
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`.

sql

```
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.

sql

```
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;
            END if_impair;
        END IF;

        SET v_i = v_i - 1;
    END WHILE;
```

```
24 END|
25 DELIMITER ;
26
27 CALL test_leave3();
```

message
4 est pair

message
3 est impair

message
1 est impair

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

`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`.

sql

```sql
DELIMITER |
CREATE PROCEDURE test_iterate()
BEGIN
    DECLARE v_i INT DEFAULT 0;

    boucle_while: WHILE v_i < 3 DO
        SET v_i = v_i + 1;
        SELECT v_i, 'Avant IF' AS message;

        IF v_i = 2 THEN
            ITERATE boucle_while;
        END IF;

        SELECT v_i, 'Après IF' AS message;
        -- Ne sera pas exécuté pour v_i = 2
    END WHILE;
END |
DELIMITER ;

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`.

La boucle `LOOP`

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

sql

```sql
[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*.

sql

```
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);
```

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 boucles 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.

PROCÉDURES STOCKÉES

GESTIONNAIRES D'ERREURS, CURSEURS ET UTILISATION AVANCÉE

Le professeur

Chantal Gribaumont

Senior Developper 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

Télécharger dans l'App Store