Transactions – Programmation

Dans cette partie, nous allons aborder la programmation dans le SGBDR.

Le langage de programmation utilisé suit le standard SQL/PSM (pas compatible à 100%) qui est également adopté par PostgreSQL (PL/pgSQL), Oracle et IBM DB2 (PL/SQL), Microsoft et Sybase (Transact-SQL).

Au sein du SGBDR, nous utiliserons des procédures stockées (fonctions) et des déclencheurs (triggers, événements).

Programmation – Variables

- Avant d'aborder les instructions de programmation, nous devons apprendre à créer et utiliser des variables
- Une variable comporte un nom et une valeur
- Les identificateurs débutent toujours par le caractère @ (pour indiquer qu'il s'agit d'une variable utilisateur et non système) et peuvent comporter les caractères suivants :
  - Lettres ([a-z][A-Z] hors caractères spéciaux)
  - Chiffres
  - _ (underscore)
  - $
  - . (point)

Programmation – Variables

- Les identificateurs ne sont pas sensibles à la casse
- Une variable utilisateur n'existe que :
  - dans la session dans laquelle elle a été créée
  - durant la session dans laquelle elle a été créée

Programmation – Variables

- Pour créer une variable, nous utiliserons la syntaxe suivante :

```
SET @nom_variable = valeur;
-- ou
SET @nom_variable := valeur;  -- à utiliser dans les
                              -- requêtes
```

- Vous pouvez y accéder via un `SELECT` :

```
SELECT @nom_variable;
```

Programmation – Variables

Exemple

```
SET @var1 := 10;
SET @var2 := 20;

SELECT @var1, @var2;

SET @var1 = @var2;
SELECT @var1, @var2;

SET @var1 := @var2 * 10;
SELECT @var1, @var2;

SET @row := 0;
SELECT @row:=@row+1 AS 'Num', idcommande, idclient,datecom
FROM commande;
```

Programmation – Variables

- On peut également assigner une variable au travers d'un select :

```
SELECT count(*) INTO @nb_clients
FROM client;
```

- Vous pouvez y accéder via un `SELECT` :

```
SELECT @nb_clients;
```

Programmation – Variables

- Idem avec plusieurs variables :

```
SELECT nom, prenom INTO @nom, @prenom
FROM client
WHERE idclient=5;
```

- Vous pouvez y accéder via un `SELECT` :

```
SELECT @prenom, @nom;
```

Programmation – Variables

- Il faut que :
  - le `SELECT` ne retourne qu'une seule ligne
  - le nombre de colonnes soit égal au nombre des variables

Programmation – Requêtes préparées

- Les requêtes préparées doivent être vues comme des requêtes paramétrables
- Nous pouvons utiliser un ou plusieurs paramètres
- Ces paramètres sont symbolisés par le caractère `?`
- Lorsqu'on crée une requête préparée, elle est envoyée et stockée dans le SGBDR.
- Pour toutes les exécutions, seuls les éventuels paramètres sont envoyés par le client
- Technique très utilisée au travers de l'API MySQL
- Permet de se protéger des injections SQL
- Nous la testerons dans la console

Programmation – Requêtes préparées

On pourrait imaginer une requête permettant de visualiser les commandes passées par un client

Afin de ne pas réécrire cette requête pour chaque utilisation, nous pouvons la préparer avec le nom du client comme paramètre.

Programmation – Requêtes préparées

- Pour définir une requête, nous devons lui donner un nom et utiliser la syntaxe suivante :

```
PREPARE nom_requete_preparee
FROM 'requête SQL';
```

- Exemple

```
PREPARE select_clients
FROM
  'SELECT * FROM client';
```

Programmation – Requêtes préparées

- Pour l'exécuter, nous utiliserons la syntaxe suivante :

```
EXECUTE nom_requete_preparee;
```

- Exemple

```
EXECUTE select_clients;
```

Programmation – Requêtes préparées

- Pour ajouter un ou plusieurs paramètres (uniquement des valeurs, pas des noms de colonne ou de table), nous placerons un `?` à la place de la valeur attendue:

```
PREPARE select_clients
FROM
  'SELECT * FROM client WHERE nom=?';
```

- Pour l'exécuter, il est nécessaire de créer une variable utilisateur pour chaque paramètre (si plusieurs paramètres, ils sont séparés par une virgule) :

```
SET @nom := 'gillet';
EXECUTE select_clients USING @nom;
```

Programmation – Requêtes préparées

- En revanche, la requête elle-même peut être passée sous forme d'un paramètre :

```
SET @colonnes := 'nom, prenom, localite';
SET @query := CONCAT_WS(' ','SELECT',@colonnes,'FROM
client WHERE nom=?');
PREPARE select_clients
FROM @query;
```

- Pour l'exécuter, il est nécessaire de créer une variable utilisateur pour chaque paramètre :

```
SET @nom := 'merciers';
EXECUTE select_clients USING @nom;
```

Programmation – Requêtes préparées

- Pour supprimer une requête préparée :

```
DEALLOCATE PREPARE nom_requete_preparee;
```

- Exemple

```
DEALLOCATE PREPARE select_clients;
```

Programmation – Routines stockées

- Les routines stockées peuvent être des procédures ou des fonctions qui sont attachées à la DB (elles ne sont pas liées à une session ou une transaction et persistent donc jusqu'à une éventuelle suppression volontaire)
- Elles prennent des paramètres en entrée et, en tant que fonctions, retournent des données
- Elles permettent :
  - De simplifier le code
  - De minimiser les données échangées entre le client et le serveur lors de l'exécution d'une requête
  - D'augmenter la vitesse d'exécution car la requête, aussi complexe soit-elle, est « précompilée » (analyse syntaxique et sémantique) du côté du serveur

Programmation – Procédures stockées

- La requête minimale de création d'une procédure stockée est la suivante :

```
CREATE PROCEDURE nom_procedure()
BEGIN

END;
```

- Exemple

```
CREATE PROCEDURE liste_clients()
BEGIN
   SELECT * FROM client;
END;
```

Programmation – Procédures stockées

- La dernière requête ne fonctionne pas car le premier « ; » rencontré est considéré comme fin de l'instruction de la création de la procédure stockée.
- Nous devons donc, le temps de la rédaction de celle-ci, modifier le délimiteur de fin d'instruction :

```
DELIMITER |
CREATE PROCEDURE liste_clients()
BEGIN
   SELECT * FROM client;
END |
DELIMITER ;
```

Programmation – Procédures stockées

- Pour appeler la procédure stockée, nous utiliserons l'instruction `CALL` :

```
CALL liste_clients();
```

Programmation – Procédures stockées

- La requête complète de création d'une procédure stockée est la suivante :

```
CREATE PROCEDURE nom_procedure([param1,[param2,…]])
BEGIN
-- corps de la procédure
END;
```

Programmation – Procédures stockées

Un paramètre est composé :

1) D'une option pour indiquer le sens du paramètre :
   - `IN` : paramètre entrant, utilisé en lecture (par défaut si rien n'est indiqué)
   - `OUT` : paramètre sortant, utilisé comme résultat
   - `INOUT` : paramètre entrant–sortant, utilisé en lecture et comme résultat
2) De son nom : par convention, ajoutez un préfixe afin de différencier les paramètres des éléments SQL (`pi_`, `po_` ou `pio_`)
3) De son type : voir types des colonnes

Exemple : `IN pi_nom_client VARCHAR(30)`

Programmation – Procédures stockées

- Requête de création d'une procédure stockée de sélection de clients par leurs noms :

```
DELIMITER |
CREATE PROCEDURE select_clients_par_nom(IN pi_nom VARCHAR(30))
BEGIN
   SELECT * FROM client WHERE nom=pi_nom;
END |
DELIMITER ;
```

Programmation – Procédures stockées

- Appel de la procédure créée :

```
CALL select_clients_par_nom('gillet');
-- ou
SET @nom := 'hansenne';
CALL select_clients_par_nom(@nom);
```

Programmation – Procédures stockées

- Pour supprimer une procédure stockée :

```
DROP PROCEDURE [IF EXISTS] nom_procedure;
```

- Pour modifier une procédure stockée, il faut d'abord la supprimer puis la recréer
- L'instruction `ALTER PROCEDURE` ne permet pas une modification des paramètres ou du corps de la procédure, seulement de ses caractéristiques

Programmation – Procédures stockées

- Pour afficher le code d'une procédure stockée :

```
SHOW CREATE PROCEDURE nom_procedure \G;
```

- Pour afficher la liste des procédures stockées :

```
SHOW PROCEDURE STATUS;
-- ou
SHOW PROCEDURE STATUS WHERE db='nom_db';
```

Programmation – Fonctions stockées

- Une fonction sera utilisée pour retourner une valeur (exécuter une formule, liste de valeurs discrètes, etc.)
- Elle pourra être utilisée dans les procédures mais également dans les requêtes SQL comme n'importe quelle autre fonction

Programmation – Fonctions stockées

- La requête minimale de création d'une fonction est la suivante :

```
CREATE FUNCTION nom_fonction() RETURNS type_donnée_retour
DETERMINISTIC
BEGIN
   RETURN;
END;
```

- Exemple

```
DELIMITER |
CREATE FUNCTION test() RETURNS INT DETERMINISTIC
BEGIN
   RETURN 1;
END |
DELIMITER ;
```

Programmation – Fonctions stockées

- Pour appeler une fonction :

```
SELECT nom_fonction();
```

- Pour supprimer une fonction :

```
DROP FUNCTION [IF EXISTS] nom_fonction;
```

Programmation – Fonctions stockées

- Pour afficher le code d'une fonction stockée :

```
SHOW CREATE FUNCTION nom_fonction \G;
```

- Pour afficher la liste des fonctions stockées :

```
SHOW FUNCTION STATUS;
-- ou
SHOW FUNCTION STATUS WHERE db='nom_db';
-- ou
SELECT name, db, type FROM mysql.proc;
```

Programmation – Privilèges

- Les privilèges associés aux procédures sont :
  - `CREATE ROUTINE` : création de la routine
  - `ALTER ROUTINE` : suppression de la routine et modification des caractéristiques
  - `EXECUTE` : exécution de la routine (`CALL` ou `SELECT`)

Programmation – Exercices

Exercices 1

Vous devez créer les procédures suivantes :

1. Sélection des clients en fonction de la localité
2. Sélection des commandes d'un client en fonction de son nom
3. Sélection des articles commandés par les clients en fonction d'une valeur minimale du compte

Programmation – Instructions

Nous allons maintenant lister les instructions du langage de programmation implémenté par MySQL :

- Variables
- Structures conditionnelles
- Structures de répétition

Programmation – Variables

Pour déclarer une variable, nous utiliserons l'instruction suivante :

```
DECLARE nom_variable type_variable [DEFAULT val_defaut];
```

Les paramètres `nom_variable` et `type_variable` sont similaires aux variables utilisateurs.

Par convention, on préfixe les variables locales de « `l_` » (L minuscule suivi d'un underscore)

Si pas de valeur par défaut, la variable vaut `NULL`

Programmation – Variables

Exemple :

```
DECLARE l_compteur INT DEFAULT 0;
```

Programmation – Variables

Modification de la valeur d'une variable :

```
SET nom_variable := valeur;
```

Programmation – Variables

On peut déclarer différents blocs afin de structurer notre code à l'aide des instructions suivantes :

```
BEGIN


END;
```

La portée d'une variable locale correspond au bloc dans lequel elle a été déclarée, ainsi que ses sous-blocs.

La durée de vie est la même que le bloc dans lequel elle a été déclarée.

Programmation – Variables

Exemple

```
DELIMITER |
CREATE PROCEDURE test_variable()
BEGIN
  DECLARE l_niv_1 INT DEFAULT 1;
  SELECT l_niv_1 AS 'Niveau 1';
  BEGIN
    DECLARE l_niv_1 INT DEFAULT 11;
    DECLARE l_niv_2 INT DEFAULT 2;
    SELECT l_niv_1 AS 'Niveau 1', l_niv_2 AS 'Niveau 2';
    SET l_niv_1 := 12;
    SELECT l_niv_1 AS 'Niveau 1', l_niv_2 AS 'Niveau 2';
  END;
  SELECT l_niv_1 AS 'Niveau 1';
  SELECT l_niv_1 AS 'Niveau 1', l_niv_2 AS 'Niveau 2';
END |
DELIMITER ;
```

Programmation – Structures conditionnelles

Il existe quatre types de structures conditionnelles :

1. La sélection
2. L'alternative
3. La suite d'alternatives
4. Le choix multiple

1. La sélection

Similaire au `if` en C.

La structure de base est la suivante :

```
IF condition(s) THEN
  Bloc d'instruction(s)
END IF;
```

Programmation – Structures conditionnelles

1. La sélection – exemple :

```
DELIMITER |
CREATE PROCEDURE test_if(IN pi_compte INT)
BEGIN
  IF pi_compte > 0 THEN
    SELECT pi_compte AS 'Le compte est positif';
  END IF;
END |
DELIMITER ;
```

2. L'alternative

Similaire au `if...else` en C.

La structure de base est la suivante :

```
IF condition(s) THEN
  Bloc d'instruction(s) 1
ELSE
  Bloc d'instruction(s) 2
END IF;
```

2. L'alternative – exemple :

```
DELIMITER |
CREATE PROCEDURE test_if_else(IN pi_compte INT)
BEGIN
  IF pi_compte >= 0 THEN
    SELECT pi_compte AS 'Le compte est positif';
  ELSE
    SELECT pi_compte AS 'Le compte est négatif';
  END IF;
END |
DELIMITER ;
```

3. La suite d'alternatives

Similaire à des imbrications de `if...else` en C.

La structure de base est la suivante :

```
IF condition(s) 1 THEN
  Bloc d'instruction(s) 1
ELSEIF condition(s) 2 THEN
  Bloc d'instruction(s) 2
ELSE
  Bloc d'instruction(s) 3
END IF;
```

3. La suite d'alternatives – exemple :

```
DELIMITER |
CREATE PROCEDURE test_if_elseif_else(IN pi_compte INT)
BEGIN
  IF pi_compte > 0 THEN
    SELECT pi_compte AS 'Le compte est positif';
  ELSEIF pi_compte < 0 THEN
    SELECT pi_compte AS 'Le compte est négatif';
  ELSE
    SELECT pi_compte AS 'Le compte est nul';
  END IF;
END |
DELIMITER ;
```

Programmation – Structures conditionnelles

Exercices 2

1. Sur base d'un paramètre représentant l'id d'un client, vous afficherez s'il existe des commandes ainsi que leur nombre
2. Sur base d'un paramètre représentant l'id d'un client, vous déterminerez et afficherez si l'état de son compte est supérieur, égal ou inférieur à la moyenne des comptes, que vous afficherez également
3. Sur base d'un paramètre représentant l'id d'un produit, vous afficherez le libellé et la quantité totale commandée

4. Le choix multiple

Similaire au `switch...case` en C.

Il existe deux types de choix multiple :

1) Basé sur des valeurs. Équivalent à comparer l'égalité d'une variable à des valeurs spécifiques
2) Basé sur des conditions

4. Le choix multiple

Basée sur des valeurs, la structure de base est la suivante :

```
CASE variable
   WHEN valeur 1 THEN
       instruction(s) 1
   WHEN valeur 2 THEN
      instruction(s) 2
   …
   ELSE -- équivalent du default en C
      instruction(s) n
END CASE;
```

4. Le choix multiple – valeurs – exemple :

```
DELIMITER |
CREATE PROCEDURE test_case_valeurs(IN pi_cat CHAR(2))
BEGIN
  CASE pi_cat
    WHEN 'A0' THEN
      SELECT pi_cat AS 'Client de base';
    WHEN 'B1' THEN
      SELECT pi_cat AS 'Client confirmé';
    WHEN 'C2' THEN
      SELECT pi_cat AS 'Client premium';
    ELSE
      SELECT pi_cat AS 'Nouveau client';
  END CASE;
END |
DELIMITER ;
```

4. Le choix multiple

Basée sur des conditions, la structure de base est la suivante :

```
CASE
   WHEN condition(s) 1 THEN
       instruction(s) 1
   WHEN condition(s) 2 THEN
      instruction(s) 2
   …
   ELSE -- équivalent du default en C
      instruction(s) n
END CASE;
```

4. Le choix multiple – conditions – exemple :

```
DELIMITER |
CREATE PROCEDURE test_case_conditions(IN pi_cat CHAR(2))
BEGIN
  CASE
    WHEN pi_cat LIKE 'A%' THEN
      SELECT pi_cat AS 'Client de base';
    WHEN pi_cat LIKE 'B%' THEN
      SELECT pi_cat AS 'Client confirmé';
    WHEN pi_cat LIKE 'C%' THEN
      SELECT pi_cat AS 'Client premium';
    ELSE
      SELECT pi_cat AS 'Nouveau client';
  END CASE;
END |
DELIMITER ;
```

4. Le choix multiple

Le cas « `else` » étant facultatif, en l'absence de celui-ci, si aucun cas n'est vrai, une erreur est déclenchée.

Si on prévoit un cas, mais qu'il ne contient aucune instruction, il faut placer un bloc vide `BEGIN END;`

Programmation – Structures conditionnelles

Exercices 3

1. Sur base d'un paramètre représentant l'id d'un client, vous afficherez, en fonction de son compte et selon les gammes suivantes, le message correspondant :
   - <= 0 : Gamme 0
   - 1 – 2000 : Gamme 1
   - 2001 – 4000 : Gamme 2
   - 4001 – 6000 : Gamme 3
   - >= 6001 : Gamme 4

Programmation – Structures de répétition

Il existe trois types de boucles :

1. À conditions avec test en entrée de boucle
2. À conditions avec test en fin de boucle
3. Infinie

Programmation – Structures de répétition

Avant d'aborder les différentes boucles, deux instructions sont à voir :

- `LEAVE` : équivalent au `break` en C. Cette instruction est accompagnée du label du bloc dont on doit sortir.
- `ITERATE` : équivalent au `continue` en C. Cette instruction est accompagnée du label de la boucle à répéter.

1. La boucle à conditions en entrée de boucle

Il s'agit d'une boucle qui sera exécutée tant qu'une (ou des) condition est vraie.

Elle est semblable à la boucle `while` en C.

La syntaxe est la suivante :

```
[label_de_boucle: ]WHILE condition(s) DO
  instruction(s)
END WHILE [label_de_boucle];
```

1. La boucle à conditions en entrée de boucle – exemple :

```
DELIMITER |
CREATE PROCEDURE test_while()
BEGIN
  DECLARE l_max INT DEFAULT 10;
  DECLARE l_compteur INT DEFAULT 1;
  WHILE l_compteur <= l_max DO
    SELECT l_compteur AS 'Compteur';
    SET l_compteur := l_compteur + 1;
  END WHILE;
END |
DELIMITER ;
```

2. La boucle à conditions en sortie de boucle

Il s'agit d'une boucle qui sera exécutée jusqu'à ce qu'une (ou des) condition soit vraie, avec au minimum une itération.

Elle est semblable à la boucle `do...while` avec une condition inverse en C.

La syntaxe est la suivante :

```
[label_de_boucle: ]REPEAT
  instruction(s)
UNTIL condition(s)
END REPEAT [label_de_boucle];
```

2. La boucle à conditions en sortie de boucle – exemple :

```
DELIMITER |
CREATE PROCEDURE test_repeat()
BEGIN
  DECLARE l_max INT DEFAULT 10;
  DECLARE l_compteur INT DEFAULT 1;
  REPEAT
    SELECT l_compteur AS 'Compteur';
    SET l_compteur := l_compteur + 1;
  UNTIL l_compteur > l_max
  END REPEAT;
END |
DELIMITER ;
```

3. La boucle infinie

Il s'agit d'une boucle qui s'exécute indéfiniment.
On en sort sur base d'un test qui fera appel à un `LEAVE`.
La syntaxe est la suivante :

```
[label_de_boucle: ]LOOP
  instruction(s)
END LOOP [label_de_boucle];
```

3. La boucle infinie – exemple :

```
DELIMITER |
CREATE PROCEDURE test_loop()
BEGIN
  DECLARE l_max INT DEFAULT 10;
  DECLARE l_compteur INT DEFAULT 1;
  label_loop: LOOP
    SELECT l_compteur AS 'Compteur';
    SET l_compteur := l_compteur+1;
    IF l_compteur>l_max THEN
      LEAVE label_loop;
    END IF;
  END LOOP;
END |
DELIMITER ;
```

Exercices 4

1. Avec chaque type de boucle, affichez les dix premiers éléments de la table de multiplication d'un nombre passé en paramètre
2. Affichez, pour les dix premiers idclient, le nom et le prénom si le client existe, « Vide » sinon.
3. Testez les différents cas de figure d'utilisation de `LEAVE` et `ITERATE` :
   - Label de boucle
   - Label d'une boucle externe
   - Label placé sur le bloc global