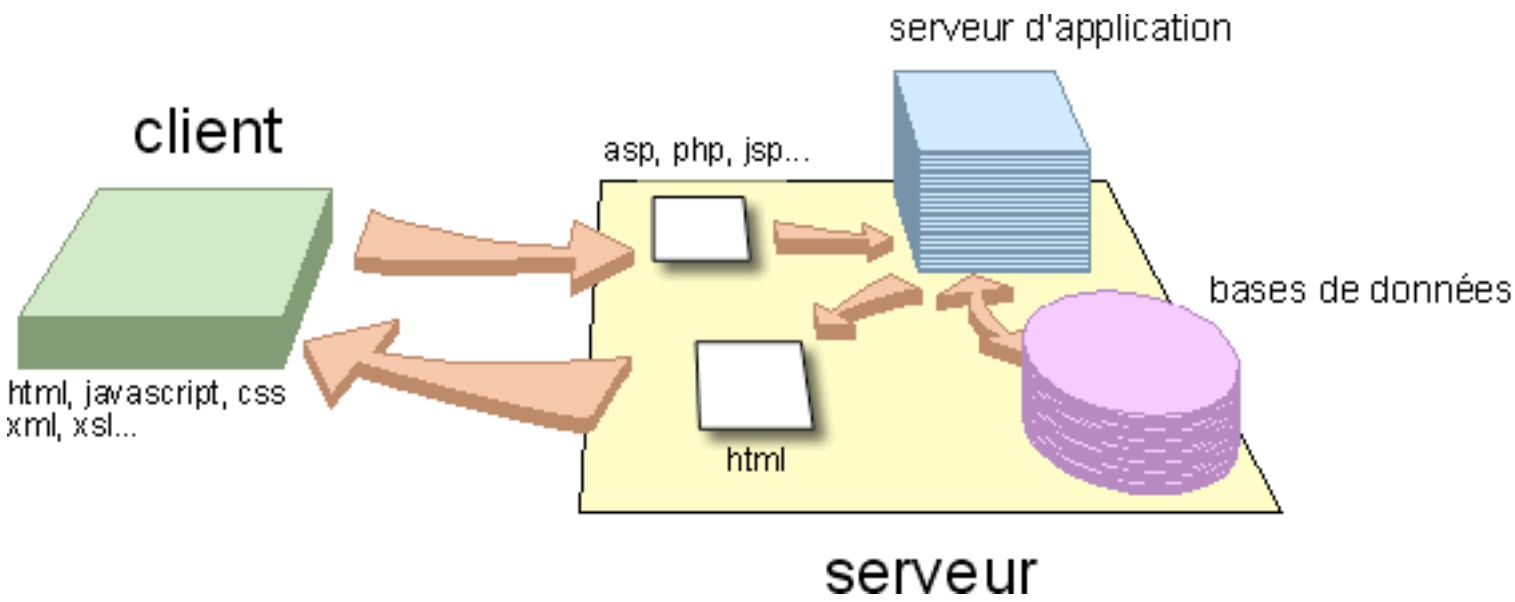
Programmation côté serveur

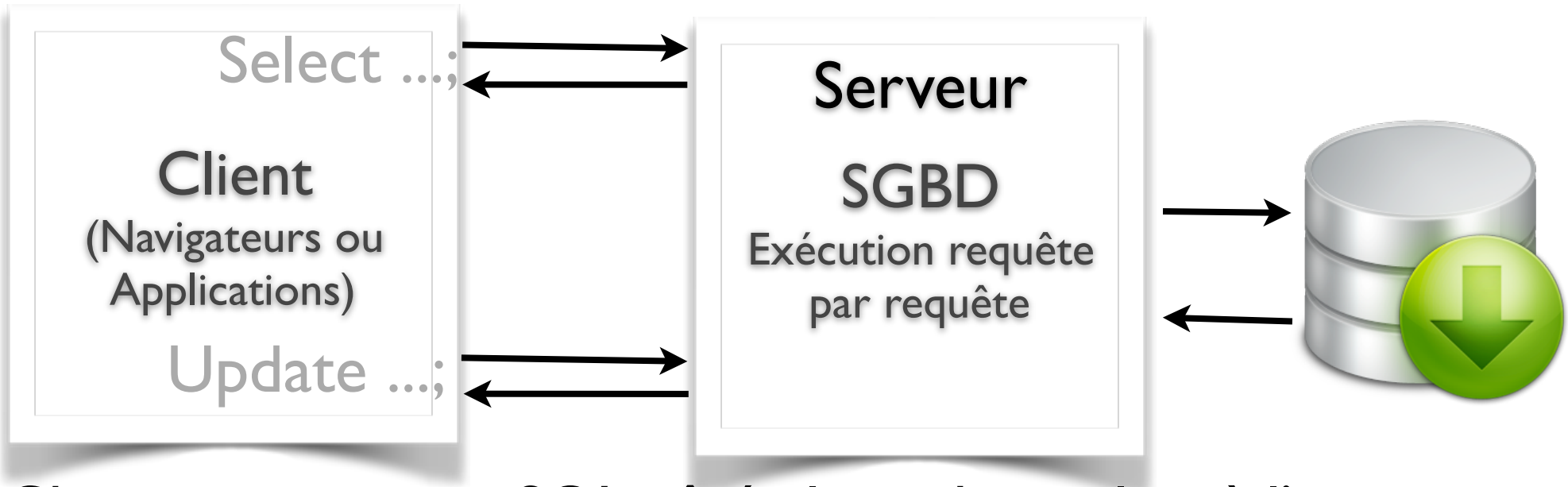
Dans l'environnement MySQL

BTS SIO 2ème année : SLAM 3

Client-Serveur

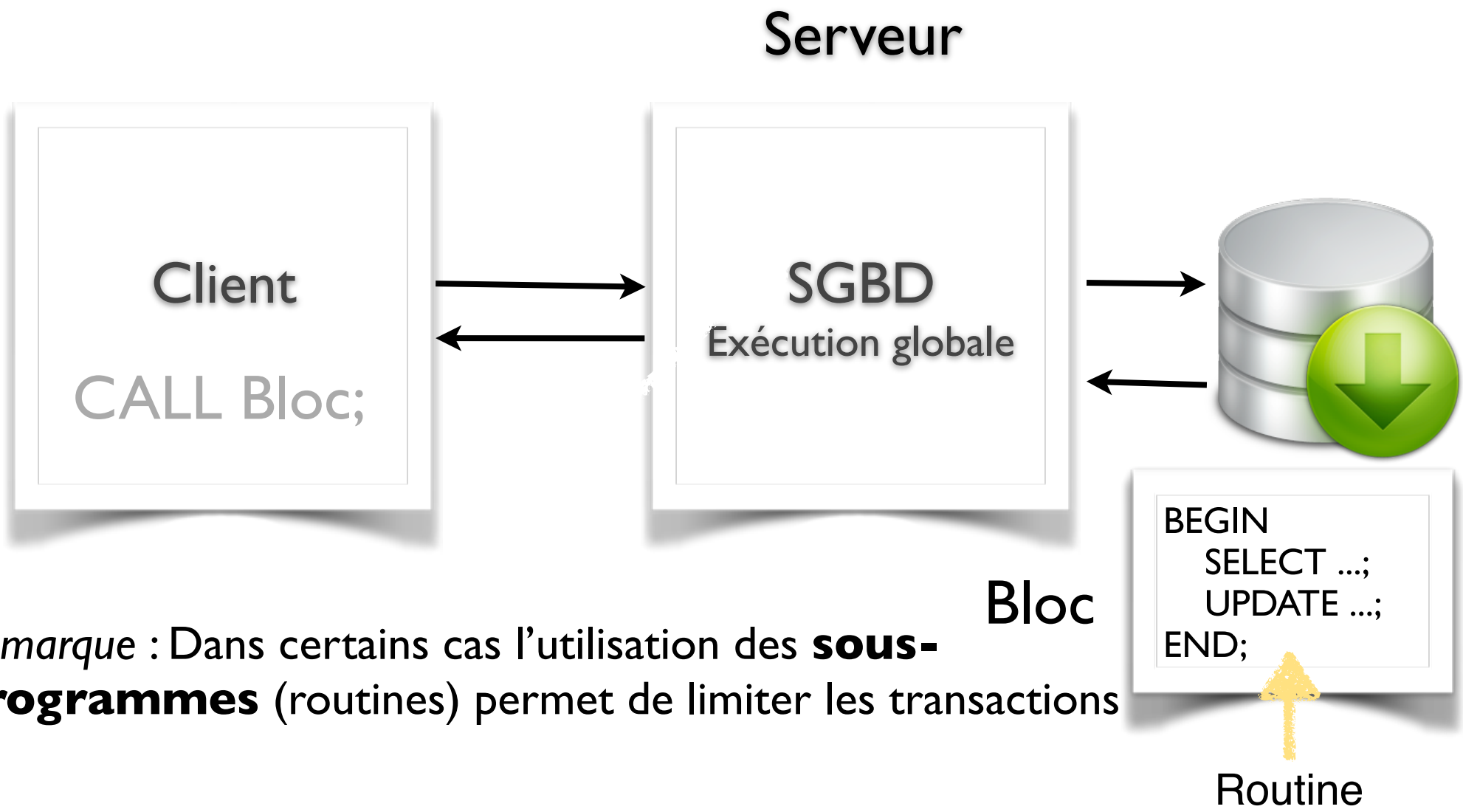


Client-Serveur



Chaque instruction SQL côté client donne lieu à l'envoi d'une demande puis d'une réponse du serveur.
Cela peut rendre l'**application plus lente** et **encombrer le trafic réseau**.

Client-Serveur



Remarque : Dans certains cas l'utilisation des **sous-programmes** (routines) permet de limiter les transactions

Avantages

- **Limite le nombre d'échange** et donc l'encombrement du trafic réseau.
- **Factorisation du code** (appel identique pour PHP, Java, etc.)
- **Modularité** : un sous-programme peut lui-même être décomposé en sous-programme. Il peut également appeler un autre sous-programme.
- **Portabilité** : il est indépendant du S.E. qui héberge MySQL
- **Puissance** : On peut utiliser toutes les instructions SQL + les types de données.
- **Sécurité** : les sous-programmes s'exécutent dans le SGBD (a priori sécurisé)

MySQL

Routines

Bases du langage

Le langage procédural de MySQL est une extension de SQL, car il permet de faire cohabiter les habituelles structures de contrôle (si, pour et tant que pour les plus connues) avec des instructions SQL (Select, Insert, Update, Delete)

Les routines stockés

Les SGBD-R en général, et MySQL en particulier, permettent d'écrire des procédures de programmation impérative classique.

Cette possibilité est apparue avec la version 5.0.

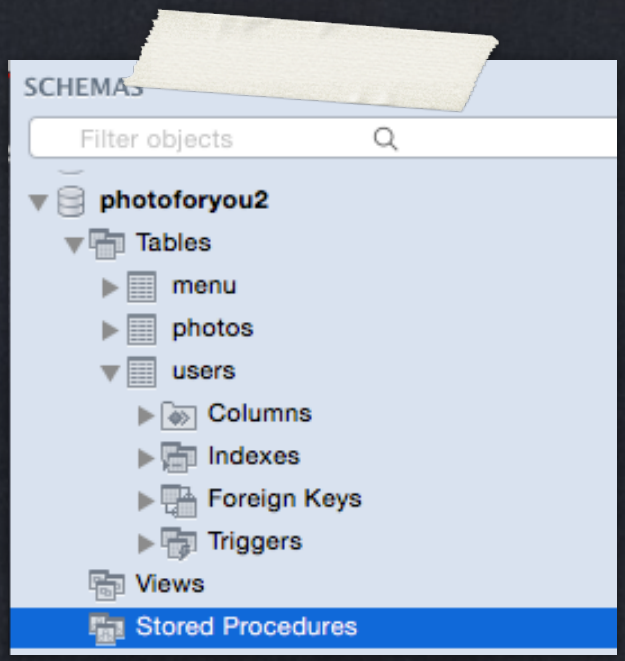
Les routines sont des bouts de code : fonctions, procédures, etc.

Les routines stockés

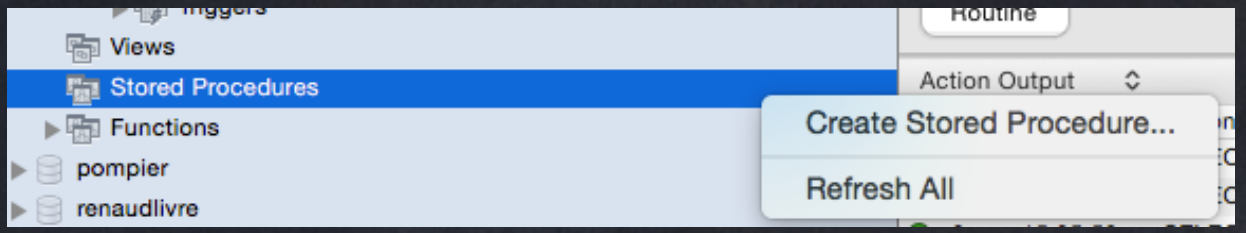
Le langage est rudimentaire mais fonctionnel :

- Variables (DECLARE, SET)
- Opérateurs (=, AND, LIKE)
- Fonctions de contrôle (IF, CASE, REPEAT, LOOP...)
- Curseur

Sous MySQLWorkbench

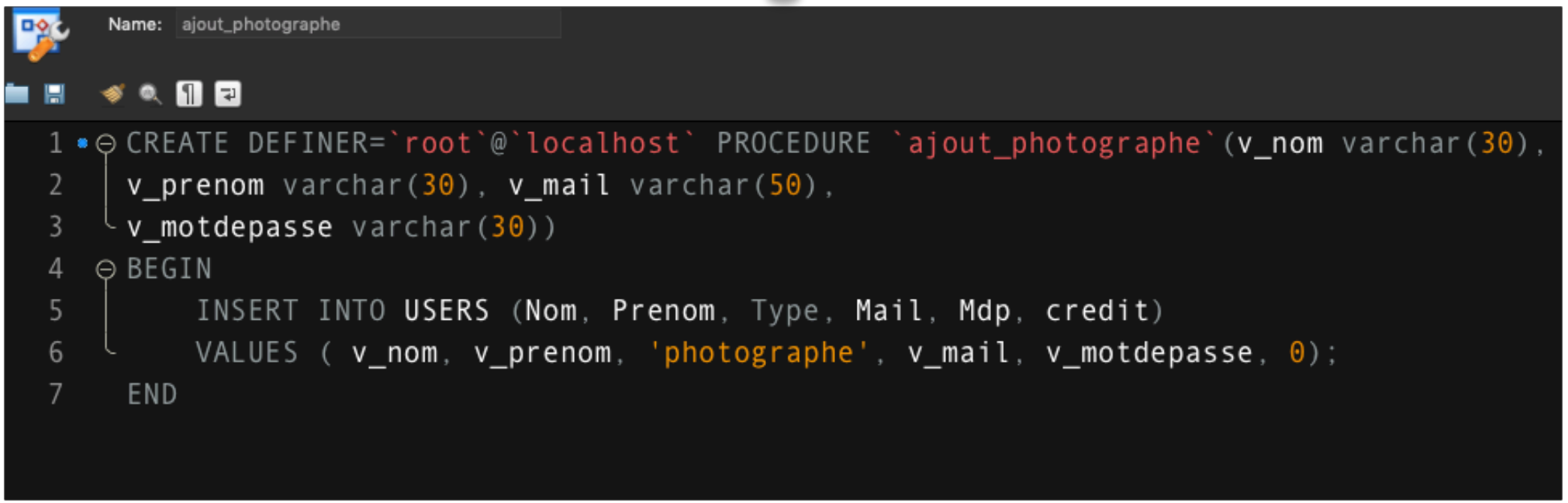


Click droit



```
1  CREATE PROCEDURE `new_procedure` ()
2  BEGIN
3
4  END
5
```

Sous MySQLWorkbench



call ajout_photographe('testNom2', 'testPrenom2', 'testMail2', ‘testMotdepasse2');

Gestion des procédures stockées

Afficher les procédures existantes :

```
Show procedure status ;
```

Afficher le code d'une procédure :

```
Show create procedure insertemp ;
```

Supprimer une procédure :

```
Drop procedure insertemp ;
```

Créer une procédure :

```
CREATE PROCEDURE `new_procedure` ()
```

Variables et déclaration

```
drop procedure if exists testVariables;
delimiter //
create procedure testVariables ( )
begin
     declare my_int int;
     declare my_bigint bigint;
     declare my_num numeric(8,2);
     declare my_pi float default 3.1415926;
     declare my_text text;
     declare my_date date default '2008-02-01';
     declare my_varchar varchar(30) default 'bonjour';
     set my_int=20;
     set my_bigint = power(my_int,3);
     select my_varchar, my_int, my_bigint, my_pi, my_date,
my_num, my_text;
end ;
//
delimiter ;
```

```
mysql> call testVariables;
+------------+--------+-----------+---------+------------+--------+---------+
| my_varchar | my_int | my_bigint | my_pi   | my_date    | my_num | my_text |
+------------+--------+-----------+---------+------------+--------+---------+
| bonjour    |     20 |      8000 | 3.14159 | 2008-02-01 |   NULL | NULL    |
+------------+--------+-----------+---------+------------+--------+---------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.04 sec)
```

Paramètres : IN, OUT, INOUT

Les paramètres des procédures sont en entrées par défaut : IN par défaut.

On peut spécifier un mode de passage en sortie : OUT ou INOUT s'il est en entrée-sortie.

La variable en sortie peut être récupérée comme variable globale de MySQL.

Paramètres en sortie : OUT et INOUT

Name: racineCarre

```
1  CREATE DEFINER=`root`@`localhost` PROCEDURE `racineCarre`(IN n int, OUT racine float)
2  BEGIN
3      set racine = sqrt(n);
4  END
```

```
1  CALL racineCarre(9, @res);
2  SELECT @res;
3
```

100% 13:2

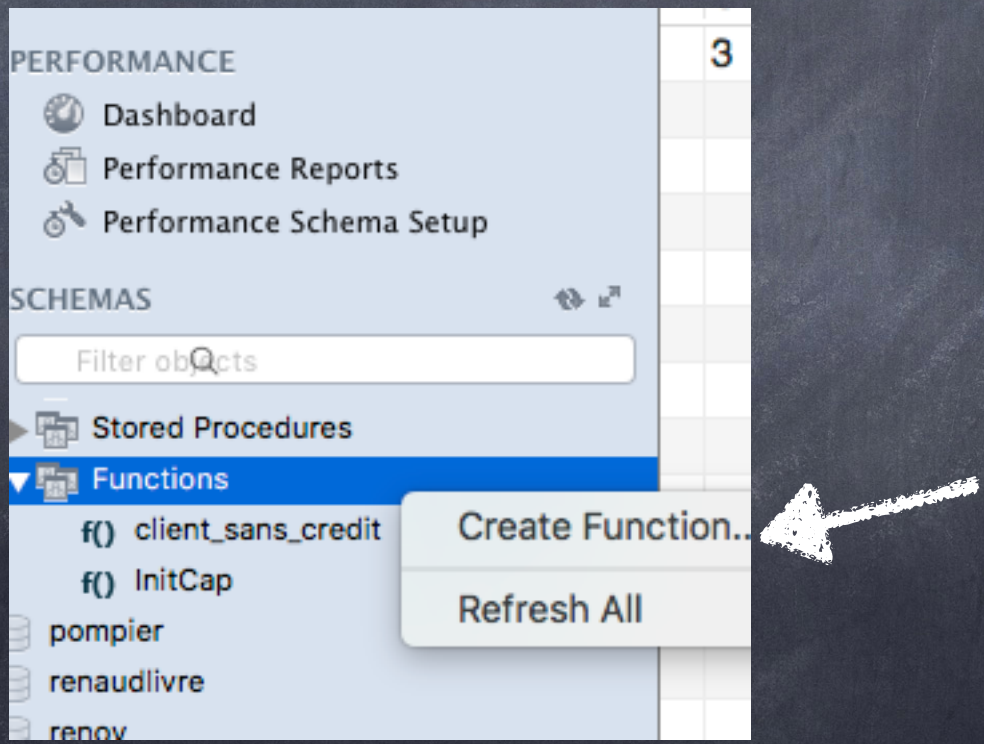
Result Grid | Filter Rows: Search | Export:

@res
3

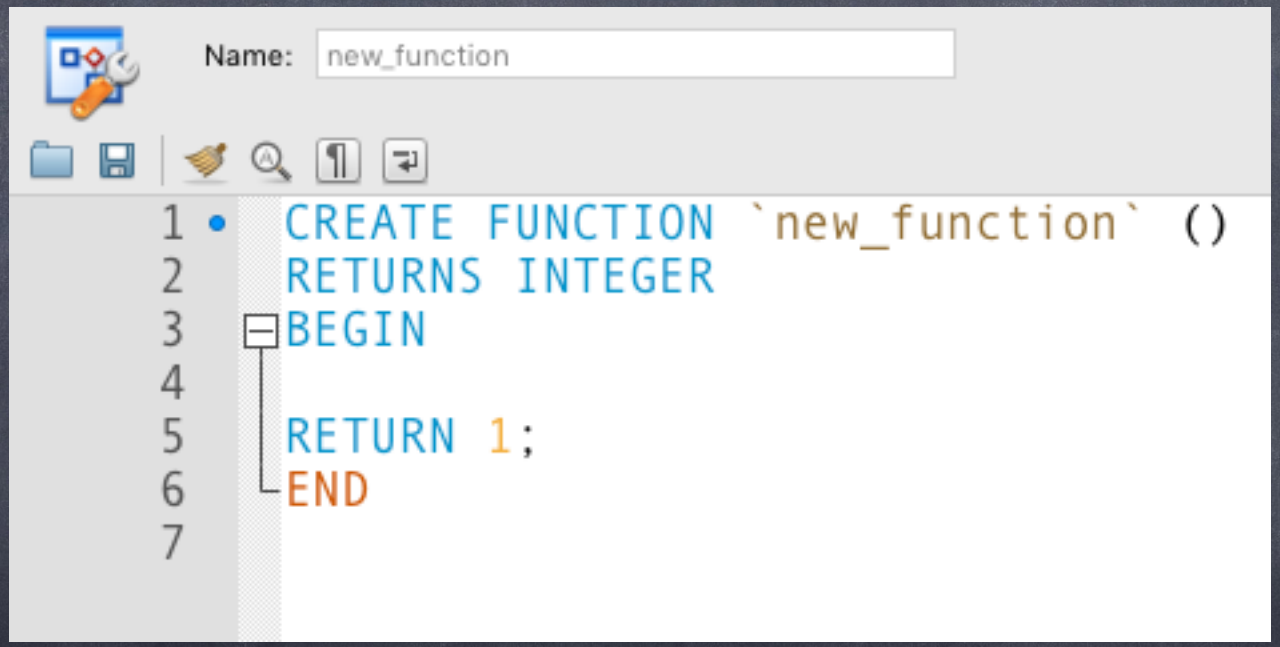
Les fonctions

Les fonctions stockées

Nous venons de voir la procédure racineCarre(). Mais lorsqu'on a besoin que d'une valeur de retour en fonction d'un ou plusieurs paramètres on utilisera les fonctions.
Les fonctions n'ont qu'un paramètre en sortie qui est renvoyé par le return. Donc, tous les paramètres de l'en-tête sont en entrées : on ne le précise pas.



Les fonctions stockées

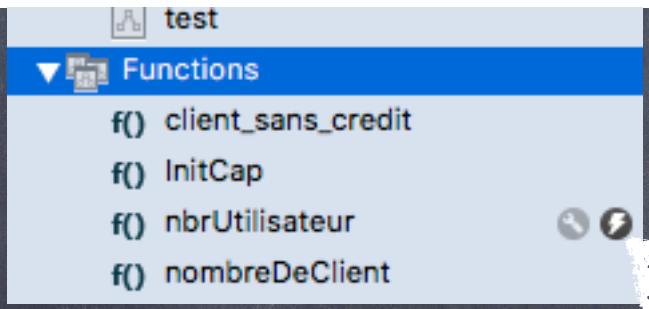


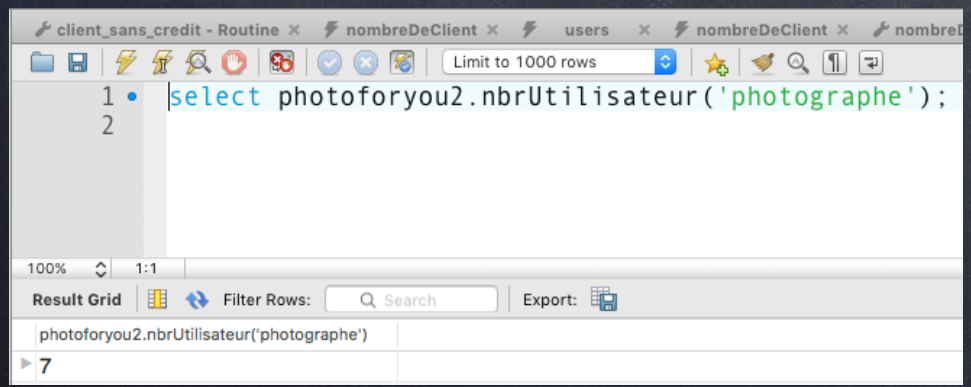
Les fonctions stockées

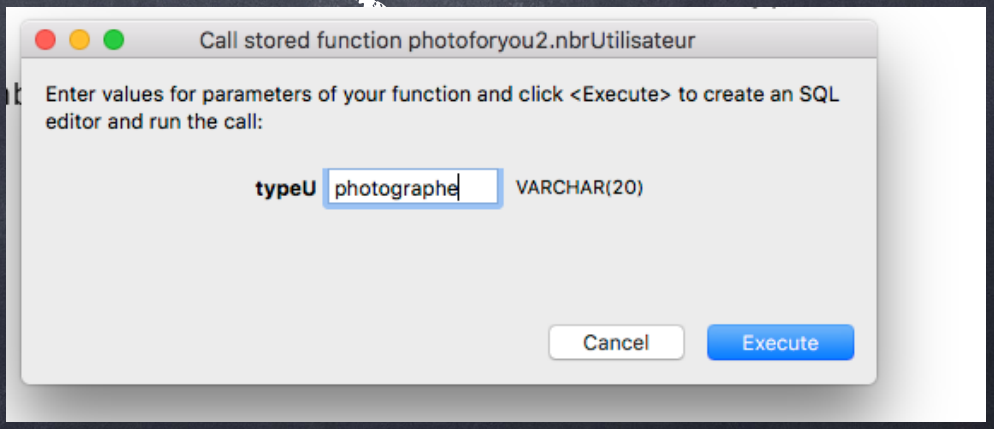
```
1  CREATE DEFINER=`root`@`localhost` FUNCTION `nombreDeClient`() RETURNS int(11)
2  BEGIN
3  DECLARE nbrClient INT;
4  SELECT COUNT(*) into nbrClient
5  FROM users
6  WHERE Type = 'client';
7  RETURN nbrClient;
8  END
```

Les fonctions stockées avec paramètre(s)

```sql
CREATE DEFINER=`root`@`localhost` FUNCTION `nbrUtilisateur`( typeU VARCHAR(20)) RETURNS int(11)
BEGIN
DECLARE nbr INT;
SELECT COUNT(*) into nbr
FROM users
WHERE Type = typeU;
RETURN nbr;
END
```







Tests

```
IF expression THEN
      instructions
[  ELSEIF   expression THEN
     instructions ]
[  ELSE
     instructions ]
END IF;
```

Tests

```
drop procedure if exists soldes;
delimiter //
create procedure soldes(prix numeric(8,2), OUT prixSolde
numeric(8,2))
begin
     if (prix > 500) then
       set prixSolde=prix * 0.8 ;
     elseif (prix >100) then
       set prixSolde = prix * 0.9 ;
     else
       set prixSolde = prix ;
     end if ;
end ;
//
delimiter ;
```

```
mysql> call soldes(1000, @nouveauPrix);
Query OK, 0 rows affected (0.03 sec)

mysql> select @nouveauPrix;
+---------------+
| @nouveauPrix |
+---------------+
| 800.00        |
+---------------+
1 row in set (0.02 sec)
```

Boucles

```
    drop procedure if exists testWhile;
    delimiter //
    create procedure testWhile(n int, OUT somme int)
comment 'testWhile'
begin
          declare i int;
          set somme =0;
          set i = 1;
          while i <=n do
             set somme = somme + i;
             set i = i+1;
          end while;
    end;
    //
    delimiter ;
```

Boucles

```
[ label : ] WHILE expression DO
        instructions
END WHILE [ label ] ;
```

```
[ label : ] REPEAT
        instructions
UNTIL expression END REPEAT [ label ] ;
```

Boucle sans fin

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

Boucles

Boucle sans fin

```
[ monLabel : ] DEBUT_DE_BOUCLE
    LEAVE monLabel ;
FIN_DE_BOUCLE [monLabel ] ;
```

```
drop procedure if exists testBoucle;
delimiter //
create procedure testBoucle(n int, OUT somme int)
begin
  declare i int;
  set somme =0;
  set i = 1;
  maboucle : loop
    if ( i>n ) then
      leave maboucle;
    end if;
    set somme = somme + i;
    set i = i+1;
  end loop maboucle;
end ;
// delimiter ;
```

On sort de la boucle avec LEAVE

Récupération dans la BD

```
drop procedure if exists testSelectInto;
delimiter //
create procedure testSelectInto(my_job varchar(9))
begin
     declare somSal float(7,2);
     select sum(sal) into somSal
     from emp
     where job = my_job;
     select somSal;
end ;
//
delimiter ;
```

Récupération dans la BD : résultat en sortie

```
drop procedure if exists testSelectInto;
delimiter //
create procedure testSelectInto(my_job varchar(9), somSal
float(7,2))
begin
    select sum(sal) into somSal
    from emp
    where job = my_job;
end ;
//
delimiter ;
```

Affichage de valeur de la BD

```
drop procedure if exists testSelect;
delimiter //
create procedure testSelect(my_job varchar(9))
begin
    select my_job, avg(sal)
    from emp
    where job = my_job;
end ;
//
delimiter ;
```

Utilisation de commandes du DDL et du DML

```
drop procedure if exists insertemp;
delimiter //

create procedure testDDML ()
begin
     drop table if exists test;
     create table test (num integer);
     insert into test values(1), (2), (3);
     select * from test;
end ;
//
delimiter ;
```

On peut passer toutes les commandes du DDL, du DML et du DCL (grant, revoke, commit, rollback) à une procédure.

Les curseurs

Le mécanisme de curseur permet une programmation itérative.

Le langage SQL préconise une programmation ensembliste.

Les curseurs doivent donc être utilisés à bon escient

Les curseurs

Un curseur est une zone mémoire côté serveur (mise en cache) qui permet de traiter individuellement chaque ligne renvoyée par un SELECT.

Les curseurs doivent être déclarés après les variables et avant les exceptions.

Les curseurs

Les instructions sont :

-DECLARE nomCurseur CURSOR FOR requête SQL;
Pour la déclaration du curseur

-OPEN nomCurseur;
Ouverture du curseur et chargement des lignes.
Précision: aucune exception n'est levée si la requête ne ramène aucune ligne.

Les curseurs

Les instructions sont :

-FETCH nomCurseur INTO listeVariables;
Positionnement sur la ligne suivante et chargement de l'enregistrement courant dans une ou plusieurs variables.

-CLOSE nomCurseur;
Ferme le curseur.

Les curseurs

```sql
CREATE DEFINER=`root`@`localhost` PROCEDURE `parcours_photographes`()
BEGIN
    DECLARE r_nom, r_prenom VARCHAR(30);
    DECLARE fin TINYINT DEFAULT 0;

    DECLARE curs_photographes CURSOR
        FOR SELECT nom, prenom
        FROM users
        WHERE type = "photographe"
        ORDER BY nom, prenom;
        -- On trie les photographes par oredre alphabétique

    -- Gestion d'erreur quand on arrive à la fin du curseur
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin = 1;

    OPEN curs_photographes;

    loop_curseur: LOOP
        FETCH curs_photographes INTO r_nom, r_prenom;
        IF fin = 1 THEN
            LEAVE loop_curseur;
        END IF;
        SELECT CONCAT(r_prenom, ' ', r_nom) AS 'Photographe';
    END LOOP;

    CLOSE curs_photographes;

END
```

Les curseurs : résultats

La déclaration d'une variable de type "continue handler for not found" : cette variable prendra la valeur 1 (vrai) quand on ne trouvera plus de ligne (tuple) dans la table associée au curseur ;

```
-- Gestion d'erreur quand on arrive à la fin du curseur
DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin = 1;
```

Initialisation de la variable « vide » à 0 : faux.

```
DECLARE fin TINYINT DEFAULT 0;
```

Les curseurs : résultats

Le fetch positionne le curseur sur la ligne suivante pour le fetch suivant. Si on fait un fetch alors que le curseur est positionné sur la fin de la table, alors le « continue handler for not found » prend la valeur prévue dans la déclaration : ici vide passe à 1 (vrai).

Close curseur : ça libère l'accès aux données pour d'autres utilisateurs.

La gestion d'erreur

On va créer une base avec une table

```
/*Create Employee database for demo */
CREATE DATABASE Employee;
/*Create sample EmployeeDetails table.*/
CREATE TABLE Employee.tbl_EmployeeDetails
(
      EmpID INTEGER
     ,EmpName VARCHAR(50)
     ,EmailAddress VARCHAR(50)
     ,CONSTRAINT pk_tbl_EmployeeDetails_EmpID PRIMARY KEY (EmpID)
)ENGINE = InnoDB;
```

Syntaxe d'une exception

```
DECLARE handler_action HANDLER FOR condition_value ... statement
```

Il y a trois types d'action possible :

- CONTINUE
- EXIT
- UNDO

On peut capter les erreurs suivantes :

- mysql_error_code
- sqlstate_value
- SQLWarning
- NotFound

Quelques exemples

```
DECLARE handler_action HANDLER FOR condition_value ... statement
```

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SELECT 'Error occured';
```

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET IsError=1;
```

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION SET IsError=1;
```

```
DECLARE EXIT HANDLER FOR SQLSTATE '23000' SET IsError = 1;
```

Un exemple complet

```
1  CREATE DEFINER=`root`@`localhost` PROCEDURE `InsertEmployee`(
2          InputEmpID INTEGER,
3          InputEmpName VARCHAR(50),
4          InputEmailAddress VARCHAR(50)
5  )
6  BEGIN
7      DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SELECT 'Une erreur a eu lieu';
8      INSERT INTO Employee.tbl_EmployeeDetails
9      (
10         EmpID,
11         EmpName,
12         EmailAddress
13     )
14     VALUES
15     (
16         InputEmpID,
17         InputEmpName,
18         InputEmailAddress
19     );
20 END
```

CALL InsertEmployee (1, 'Pierre', 'pierre@sio.edu');

CALL InsertEmployee (1, 'Jacques', 'jacques@sio.edu');

Un exemple complet

CALL InsertEmployee (1, 'Pierre', 'pierre@sio.edu');

CALL InsertEmployee (1, 'Jacques', 'jacques@sio.edu');