# Database programming

# INDEX

- Programming languages and databases

- Stored procedures and functions in MySQL
  - Syntax and examples
  - Parameters and variables
  - Conditional statements
  - Repetitive statements or loops
  - SQL in routines: Cursors
  - Stored routines management
  - Error handling

- Triggers
  - Management of triggers
  - Use of triggers
  - Events

# Programming languages and databases

- Increase the functionality of DBMS
- Most incorporate own languages
- Often incorporate APIs of other languages (Java, C++, perl, Ruby, php, etc.)

# Stored procedures and functions in MySQL

## Why Store Procedures or functions?

- Abstract business logic away from clients. Database-centric logic can be isolated in stored programs and implemented by programmers with more specialized, database experience.
- Reduce Rework - Client applications in different languages need to perform the same operations. Stored programs can be used to implement common routines accessible from multiple applications.
- The use of stored programs can, under some circumstances, improve the portability of your application.
- Improved Security - Applications and users may have no access to the database tables directly, but can execute only specific stored routines.
- Improved Performance – Less data and fewer commands sent between the client and server (reduce network traffic).
- Can help with database coherence.
- Can overtake SQL limitations

# Stored procedures and functions in MySQL

**Some disadvantages**

- Server overloading.
- Portability.

# Stored procedures and functions in MySQL

- A stored procedure is a set of SQL statements that are stored in the server and executed as a block.
- Takes in 0 or more args.
- May return a set of values.

# Stored procedures and functions in MySQL

## General scheme

Routine name + parameters (in, out and in/out)

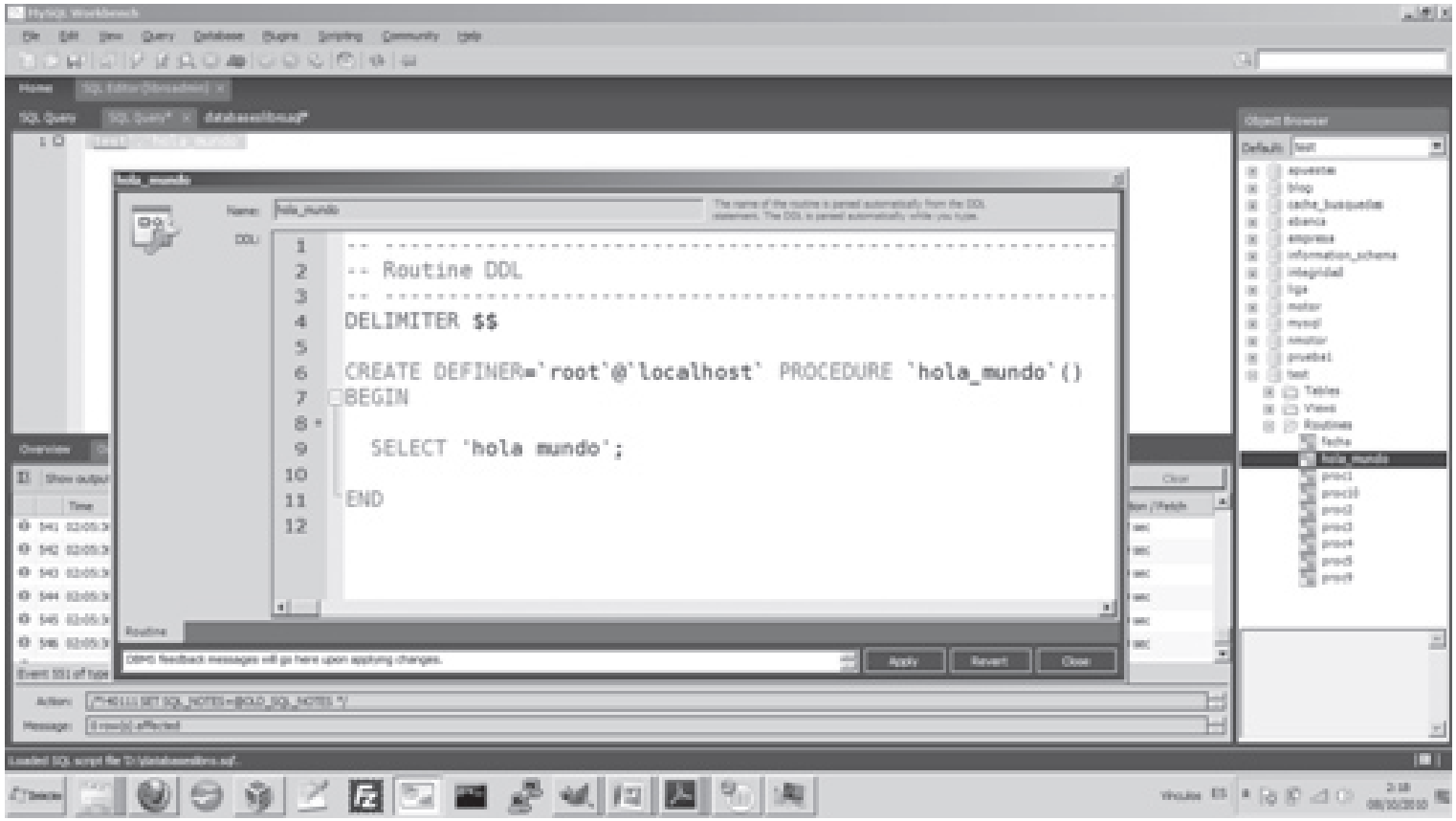
Declaration and initialization of variables

Data processing
Blocks BEGIN / END and control statements (conditional and repetitive)

End
With the RETURN statement to return a value in case of stored functions

# Stored procedures and functions in MySQL

Example 'Hello world' with Workbench



# Stored procedures and functions in MySQL

General syntax

## Function

```
CREATE FUNCTION sp_name
([parameter[,...]])
RETURNS type
[characteristic ...]
routine_body
```

## Procedure

```
CREATE PROCEDURE sp_name
([parameter[,...]])
[characteristic ...]
routine_body
```

```
parameter:
[ IN | OUT | INOUT ] param_name type

type:
Any valid MySQL data type

characteristic:
LANGUAGE SQL
| [NOT] DETERMINISTIC
| { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA } |
  SQL SECURITY { DEFINER | INVOKER }
```

# Stored procedures and functions in MySQL

- They are saved in the compiled database.
- They are executed on the server in one of their threads.
- By default they are created in the selected database, if you want to create another database, their name must be preceded by the name of the database.
- When executed, they make an implicit USE (the USE command cannot be explicitly used).
- They use the character; which is also used as a delimiter for server commands, so the delimiter needs to be changed to a different character.
- When a DROP is made from the database, all stored procedures, functions and triggers are also deleted.

# Parameters and variables I

- Parameters types:
  - IN: input
  - OUT: output
  - INOUT: input / output
- Variables
  - Types: of data
  - Scope of the variables: determined by blocks BEGIN / END

# Parameters and variables II

- Example

```
CREATE PROCEDURE proc3(OUT p INT)
       SET p = -5 $$

CALL proc3(@y)$$
SELECT @y$$
+------+
| @y |
+------+
| -5 |
+------+
```

In this case we have created a new variable @y calling the procedure, whose value is updated within the same routine because this variable is past as an output parameter

# Conditional statements I

- If-then-else

```
IF expr1 THEN
...
ELSEIF expr2 THEN
...
ELSE
...
END IF;
```

- CASE

```
CASE case_value
 WHEN when_value THEN ...
[WHEN when_value THEN ...]
[ELSE ...]
END CASE;
```

```
CASE
 WHEN search_condition THEN ...
[WHEN search_condition THEN ...]
[ELSE ...]
END CASE
```

# Conditional statements II

## Example IF-THEN-ELSE

In the following example we insert or update the t table in the test database depending on the value of input parameter

```
DELIMITER $$
CREATE PROCEDURE proc7 (IN par1 INT)
BEGIN
    DECLARE var1 INT;

    SET var1 = par1 + 1;
    IF var1 = 0 THEN
        INSERT INTO t VALUES (17);
    END IF;
    IF par1 = 0 THEN
        UPDATE t SET s1 = s1 + 1;
    ELSE
        UPDATE t SET s1 = s1 + 2;
    END IF;
END; $$
```

When the var1 value is 0, then we make an insertion, and in case the value of the input parameter was 0, we update adding 1 to the current value, else we add 2.

# Repetitive statements II

## Simple loop

[label:] **LOOP**
...
**END LOOP** [label];

## Repeat until loop

[label:] **REPEAT**
...
**UNTIL** expresion
**END REPEAT** [label]

## While loop

[label:] **WHILE** expression
**DO**
...
**END WHILE** [label]

# Repetitive statements II

In the following example we display the odd numbers from 0 to 10

```
DELIMITER $$
CREATE PROCEDURE proc10()
BEGIN
    DECLARE i int;
    SET i=0;
    loop1: REPEAT
        SET i=i+1;
        IF MOD(i,2)<>0 THEN
            SELECT CONCAT(i," es impar");
        END IF;
    UNTIL i >= 10
    END REPEAT;
END; $$
```

# SQL in routines: Cursors Definition /commands I

- A database **cursor** is a control structure that enables traversal over the records in a database.
- **Cursors** are used by database programmers to process individual rows from a set returned by database system queries.
- MySQL supports cursors inside stored programs. Cursors have these properties :
  - **Read only**: Not updatable
  - **Nonscrollable**: Can be traversed only in one direction and cannot skip rows
  - Cursor declarations must appear before handler declarations and after variable and condition declarations.

# SQL in routines: Cursors Definition /commands II

- **Cursor**: Object that refers to a data set obtained from a query

- **Cursor commands**:

  - **DECLARE**: define a new cursor

    ```
    DECLARE cursor_name CURSOR FOR SELECT_statement;
    ```

  - **OPEN**: open the cursor or its associated rows

    ```
    OPEN cursor_name
    ```

  - **FETCH**: extract the next row of a cursor

    ```
    FETCH cursor_name INTO variable list;
    ```

  - **CLOSE**: close the cursor

    ```
    CLOSE cursor_name;
    ```

# SQL in routines: Cursors Definition /commands III

Example to count the number of news using a cursor

```
CREATE PROCEDURE cursor_demo3()
BEGIN
    DECLARE tmp VARCHAR(200);
    DECLARE lrf BOOL;
    DECLARE nn INT;
    DECLARE cursor2 CURSOR FOR SELECT titulo FROM noticias;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET lrf=1;
    SET lrf=0,nn=0;
    OPEN cursor2;
        l_cursor: LOOP
            FETCH cursor2 INTO tmp;
            SET nn=nn+1;
            IF lrf=1 THEN
            LEAVE l_cursor;
            END IF;
        END LOOP l_cursor;
    CLOSE cursor2;
SELECT nn;
END; $$
```

# Stored routines management

- **Creation** (as we have seen)

- **Deletion** of routines

  To delete procedures or functions, we use the SQL DROP command with the following syntax:

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name
```

- **Check** of routines

```
SHOW CREATE {PROCEDURE | FUNCTION} sp_name

SHOW {PROCEDURE | FUNCTION} STATUS [LIKE 'pattern'];
```

# Error handling

**Syntax**:

```
DECLARE {CONTINUE | EXIT} HANDLER FOR
        {SQLSTATE sqlstate_code | MySQL error code|
         condition_name} handler_actions
```

- **Handler type**: CONTINUE or EXIT
- **Handler condition**: SQL state, own error of MySQL or error code defined by the user
- **Handler actions**: Actions to take when the handler is activated

# FULL EXAMPLE

Full example that displays the number of news published by each author. We can use two nested cursors:

```
CREATE PROCEDURE noticias_autor( ) READS SQL DATA
BEGIN
     DECLARE vautor,na_count INT;
     DECLARE fin BOOL;
     DECLARE autor_cur CURSOR FOR SELECT id_autor FROM autores;
     DECLARE noticia_cur CURSOR FOR SELECT autor_id FROM noticias WHERE autor_id=vautor;
     DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin=1;
     SET na_count=0;
     OPEN autor_cur;
     autor_loop: LOOP
          FETCH autor_cur INTO vautor;
          IF fin=1 THEN LEAVE autor_loop; END IF;
          OPEN noticia_cur;
          SET na_count=0;
          noticias_loop: LOOP
               FETCH noticia_cur INTO vautor;
               IF fin=1 THEN LEAVE autor_loop; END IF;
               SET na_count=na_count+1;
          END LOOP noticias_loop;
          CLOSE noticia_cur;
          SET fin=0;
          SELECT CONCAT('El autor',vautor,'tiene', na_count,' noticias');
     END LOOP autor_loop;
     CLOSE autor_cur;
END;$$
```

# TRIGGERS

- A trigger is an special type of stored routine that is activated or executed when an event of type INSERT, DELETE or UPDATE takes place in a table.

- The triggers implement a functionality associated with any change in a table.

```
CREATE TRIGGER insertar_movimientos BEFORE INSERT ON movimientos

FOR EACH ROW

    SET @sum = @sum + NEW.cantidad
```

# TRIGGERS

## Management of triggers

### ❑ Create trigger

```
CREATE TRIGGER trigger_name trigger_time trigger_event ON
   tbl_name FOR EACH ROW trigger_body
```

- *trigger_time* is the trigger action time. It can be **BEFORE** or **AFTER** to indicate that the trigger activates before or after each row to be modified.

- *trigger_event* indicates the kind of statement that activates the trigger. The trigger_event can be one of the following: **INSERT**, **UPDATE** or **DELETE**.

### ❑ Drop trigger

```
DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name
```

# TRIGGERS

## Management of triggers

### ❑ Alter trigger

In MySQLWorkbench, to modify an existing trigger, right-click the node of the table and choose the *Alter table* command from the context menu. Then select the tab *Trigger* to open the SQL Editor where you can modify the code of the trigger.

### ❑ Check trigger

```
SHOW TRIGGERS [{FROM | IN} db_name] [LIKE 'pattern' | WHERE
   expr]
```

NOTE: When using a LIKE clause with SHOW TRIGGERS, the expression to be matched (*expr*) is compared with the name of the table on which the trigger is declared, and not with the name of the trigger.

# TRIGGERS

## NEW and OLD keywords

Within the trigger body, the OLD and NEW keywords enable you to access columns in the rows affected by a trigger. OLD and NEW are MySQL extensions to triggers; they are not case-sensitive.

- In an INSERT trigger, only **NEW.*col_name*** can be used; there is no old row.
- In a DELETE trigger, only **OLD.*col_name*** can be used; there is no new row.
- In an UPDATE trigger, you can use **OLD.*col_name*** to refer to the columns of a row before it is updated and **NEW.*col_name*** to refer to the columns of the row after it is updated.

# TRIGGERS

## Use of triggers

- **Control of sessions**

  Sometimes it could be interesting saving some values in session variables created by the user. In the end, we can check these variables to have a resume of all we have done in that session.

- **Control of input values**

  A possible use of triggers is to control the values inserted or updated in tables.

- **Maintenance of derived fields**

  Another common use of triggers is for maintenance of derived or redundant fields, that is, fields that can be calculated from others.

# TRIGGERS

## Use of triggers

- **Statistics**

  We can register statistics of operations or values of our databases in real time using triggers.

- **Logging and auditing**

  When a lot of users access to a database, it could be necessary to update a log for an especific table every time a DELETE or UPDATE sentence is executed. In this case, we can create a trigger to save the values we want to know before and after the table is changed.

# TRIGGERS

## Raising Error Conditions with MySQL SIGNAL

- MySQL SIGNAL statement is an error handling mechanism for handling unexpected occurrences and a graceful exit from the application if need to be. Basically, it provides error information to the handler. Its basic syntax would be as follows:

```
SIGNAL condition_value
    [SET signal_information_item
    [, signal_information_item] ...]

condition_value: {
    SQLSTATE [VALUE] sqlstate_value
  | condition_name
}

signal_information_item:
    condition_information_item_name = simple_value_specification
```

# TRIGGERS

## Raising Error Conditions with MySQL SIGNAL

- The *condition_value* in a SIGNAL statement indicates the error value to be returned. It can be an SQLSTATE value (a 5-character string literal) or a *condition_name* that refers to a named condition previously defined with DECLARE ... CONDITION
- To provide the caller with information, you use the SET clause. If you want to return multiple condition information item names with values, you need to separate each name/value pair by a comma.
- The *condition_information_item_name* can be MESSAGE_TEXT, MYSQL_ERRORNO, CURSOR_NAME , etc.
- For catch-all error handling, we should assign an SQLSTATE value of '45000', which signifies an "unhandled user-defined exception"

```
DELIMITER $$

CREATE TRIGGER before_insert_studentage BEFORE INSERT ON student_age
FOR EACH ROW
BEGIN
  IF NEW.age > 150 then
    SIGNAL SQLSTATE'45000' SET MESSAGE_TEXT = 'Wrong age for a non-superman';
  END IF;
END $$
```

# TRIGGERS

**Advantages** of using SQL triggers:

- SQL triggers provide an alternative way to check the integrity of data.
- SQL triggers can catch errors in business logic in the database layer.
- SQL triggers provide an alternative way to run scheduled tasks. By using SQL triggers, you don't have to wait to run the scheduled tasks because the triggers are invoked automatically before or after a change is made to the data in the tables.
- SQL triggers are very useful to audit the changes of data in tables.

# EVENTS

In MySQL events are tasks that run according to a schedule (*events scheduled*).

Also known as *temporal triggers*, as they are conceptually similar. They differ in that a trigger is activated by a change in the data base, while an event is triggered according to a planned schedule.

An event is identified by a name and the database where is assigned.

Types:

- ✔ those who are scheduled only one time
- ✔ those who are periodically scheduled every specific interval of time

The MySQL Event Scheduler manages the scheduling and execution of events.

To see if Event Scheduler is active:

- **SHOW VARIABLES LIKE 'event_scheduler'**
- **SHOW PROCESSLIST**

# EVENTS

To turn on the Event Scheduler:

- **SET GLOBAL event_scheduler = ON;**
- **SET @@global.event_scheduler = ON;**
- **SET GLOBAL event_scheduler = 1;**
- **SET @@global.event_scheduler = 1;**

To turn off the Event Scheduler:

- **SET GLOBAL event_scheduler = OFF;**
- **SET @@global.event_scheduler = OFF;**
- **SET GLOBAL event_scheduler = 0;**
- **SET @@global.event_scheduler = 0;**

- Metadata about events can be obtained as follows:
  - Query the event table of the mysql database.
  - Query the EVENTS table of the INFORMATION_SCHEMA database.
  - Use the SHOW CREATE EVENT statement. Syntax:

    **SHOW CREATE EVENT *event_name***
  - Use the SHOW EVENTS statement. Syntax:

```
SHOW EVENTS
        [{FROM | IN} schema_name]
        [LIKE 'pattern' | WHERE expr]
```

# EVENTS

Syntax:

```
CREATE EVENT myevent
    ON SCHEDULE EVERY 5 DAY
    DO
      UPDATE myschema.mytable
         SET mycol = mycol + 1$$
```

CREATE EVENT *event_name*
ON SCHEDULE *schedule*
[ON COMPLETION [NOT] PRESERVE]
[ENABLE | DISABLE | DISABLE ON SLAVE]
[COMMENT '*comment*'] DO *event_body*;

*Schedule*:
AT *timestamp* [+ INTERVAL *interval*] …
| EVERY *interval*
[STARTS *timestamp* [+ INTERVAL *interval*] …]
[ENDS *timestamp* [+ INTERVAL *interval*] …]

*Interval*:
*quantity* {YEAR | QUARTER | MONTH | DAY | HOUR | MINUTE |
WEEK | SECOND | YEAR_MONTH | DAY_HOUR | DAY_MINUTE
DAY_SECOND | HOUR_MINUTE | HOUR_SECOND |
MINUTE_SECOND}

# EVENTS

Syntax:

```
-- We want in 'noticias' table only one month (because this
table increase a lot its size every month). So we have an
event that, every first of month, unload news created before
the last month and load them into 'arxiu_noticies'
USE motorblog $$

DELIMITER $$
CREATE EVENT arxiu_noticies
ON SCHEDULE EVERY 1 MONTH STARTS '2018-05-01 00:00:00' DO
BEGIN
  INSERT INTO historic_noticies
  SELECT * FROM noticias
   WHERE fecha <= (CURRENT_DATE() - INTERVAL 1 MONTH);

  DELETE FROM noticias
  WHERE fecha <= (CURRENT_DATE() - INTERVAL 1 MONTH);
END $$
```

# END