Stored procedures

Agenda

- Stored Routines and Functions
- Prepared Statements
- Conditions and Loops
- Cursors
- Error Handling

Stored Routines and Functions

Stored Routines and Functions

- Stored routines are the procedural extension to SQL in MySQL server
- Stored routines include the typical programming language features like variables, conditional statements, loops, exceptions, procedures, functions, ...
- Support for stored routines and functions was added in MySQL 5.5

Stored Routines and Functions

- Data manipulation and query statements of SQL are integral part of stored routines
- SQL queries can be combined with programming logic (like loops) in stored routines
- Stored routine blocks are also used to provide the procedural logic of triggers and events in MySQL

Stored Routines and Functions

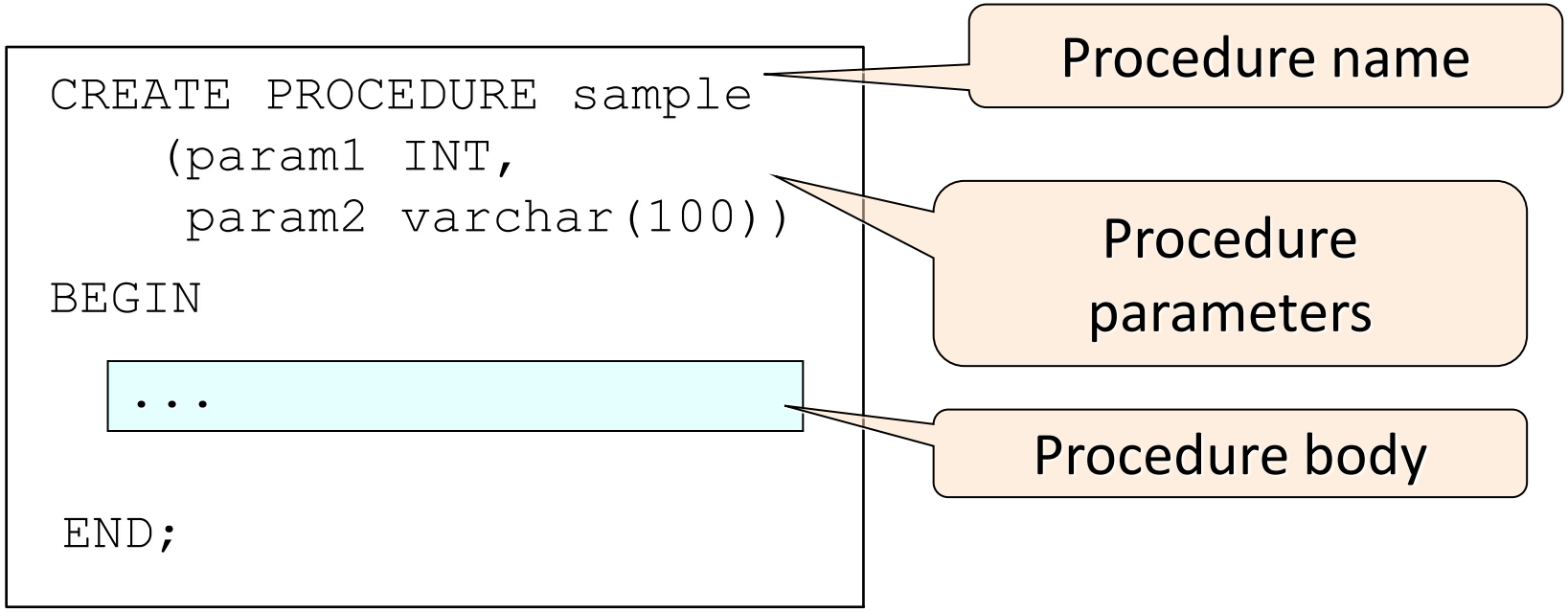
- Stored routines are named procedures that can be executed repeatedly on the database server:
  - combine SQL statements and programming logic
  - can take parameters
  - usually implement the database logic (data- related business rules)
- Functions are stored routines that return a value

Stored Routines and Functions

- Stored routines allows for improved performance:
  - data should not be moved out of the server in order to be processed
- Stored routines allows for easier maintenance, data integrity and security:
  - when the database structure is changed these changes can reflect only the stored procedures
  - applications should access the DB only through the stored routines

Stored Routines and Functions

- Stored routines are program logic and have the following structure:



Stored Routines and Functions

- Once defined routines can be called with the `CALL` command

- For example:

```
CALL sample;
```

- A stored routine can also be called from another stored routine

Stored Routines and Functions

- Parameters of the stored procedure can additionally be defined as:
  - IN - parameter has initial value when passed to the procedure but is not modified after procedure finishes
  - OUT - parameter might be modified by the procedure and used with modified value after procedure finishes
  - INOUT - parameter has an initial value and might be modified by the procedure

Stored Routines and Functions

- For example

```
CREATE DEFINER=`root`@`localhost` PROCEDURE
`sample`(IN param1 INT,
                          OUT param2 INT,
                          INOUT param3 INT)
BEGIN
    set param1 = 1;
    set param2 = 2;
    set param3 = 3;
END
```

```
set @var1 = 0;
set @var2 = 0;
set @var3 = 0;
CALL sample(@var1, @var2, @var3);
```

With SET you can define user, session and system variables

Stored Routines and Functions

- There are two ways to define variables in a stored routine:
  - using the `SET` command - defines a user variable accessible even outside the stored routine
  - using the `DECLARE` command
- The `DECLARE` command in a stored routine can be used to define:
  - local variables
  - errors conditions and handlers
  - cursors

Stored Routines and Functions

- The format for declaring a variable with `DECLARE` is:

```
DECLARE <variable_name> <datatype> [DEFAULT <value>];
```

- For example:

```
CREATE PROCEDURE sample2()
BEGIN
    DECLARE COUNTRY VARCHAR(100) DEFAULT 'Bulgaria';
    SET COUNTRY = 'GERMANY';
    SELECT COUNTRY;
END
```

Stored Routines and Functions

- SET variables are visible outside the routine procedure while DECLARE variables are visible only in the body of the procedure

- Example:

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `sample3`()
BEGIN
    set @user_defined = 'sample3 invoked';
END
```

```
call sample3;
select @user_defined;
```

Stored Routines and Functions

- Stored routines in MySQL workbench CANNOT be created directly from the editor - they are created from the **routines** folder in the object browser

- Once they are created they can be altered/removed from the **routines** folder in the object browser

- They can be called from the MySQL workbench editor with the `CALL` command

Prepared Statements

Prepared Statements

- Prepared statements can be used to created dynamic SQL queries and execute them
- Dynamic queries are created as plain text and can contain parameters that are passed when the dynamic query is executed
- Prepared statements are very useful in stored routines since queries can be created dynamically using parameters, local variables or user-defined variables

Prepared Statements

- Example:

```
SET @table = 'employees';
SET @statement = CONCAT('SELECT * FROM ', @table);
PREPARE stmt1 FROM @statement;
EXECUTE stmt1;
```

Prepared Statements

- Example:

```
PREPARE stmt2 FROM 'SELECT upper(?);';
set @var= "abc";
EXECUTE stmt2 USING @var;
```

Prepared Statements

- Prepared statements can be deleted with the `DEALLOCATE PREPARE` command

- For example:

```
DEALLOCATE PREPARE stmt2;
```

Conditions and Loops

Conditions and Loops

- Conditional and loop statement allow you to implement the logical (control) flow of your stored routines

- The `CASE` and `IF` commands we already covered can also be used in a BEGIN … END block of a stored routine in order to provide conditional logic

Conditions and Loops

- Loops are used to provide a mechanism for executing the same program logic multiple times

- Stored routines support the following types of loops:

  - using the `LOOP` command
  - using the `WHILE` command
  - using the `REPEAT` command

Conditions and Loops

- Example:

```
CREATE PROCEDURE sp_check_date(p_date datetime, out
p_state varchar(100))
begin
  declare v_hour int;
  declare v_day_of_week int;
  set v_hour := hour(p_date);
  set v_day_of_week := weekday(p_date);
  if v_hour between 8 and 18
    and v_day_of_week in (1, 2, 3, 4, 5)
  then
    set p_state = 'office hours';
  else
    set p_state = 'out of office hours';
  end if;
end
```

Conditions and Loops

- Example (cont.):

```
set @result = NULL;
call sp_check_date(now(), @result);
select @result;
```

Conditions and Loops

- LOOP statement syntax:

```
[begin_label:] LOOP
      <statement_list>
END LOOP [end_label]
```

- You can exit a loop with the LEAVE command

- You can continue to next iteration with the ITERATE command

Conditions and Loops

- Example:

```
CREATE PROCEDURE simple_loop()
BEGIN
    declare v_counter int default 5;
    declare v_person varchar(100);

    sample_loop: loop
        set v_counter = v_counter - 1;
        if(v_counter = 2)
            then iterate sample_loop; end if;
        set v_person = concat('Person', v_counter);
        select v_person;
        if v_counter = 0
            then leave sample_loop; end if;
    end loop;
END
```

Conditions and Loops

- `WHILE` statement syntax:

```
[begin_label:] WHILE <search_condition> DO
      <statement_list>
END WHILE [end_label]
```

- `REPEAT` statement syntax:

```
[begin_label:] REPEAT
      <statement_list>
UNTIL <search_condition>
END REPEAT [end_label]
```

Cursors

Cursors

- Cursors are temporary work area created in system memory when an SQL statement is executed
- They are used to iterate and manipulate data returned from an SQL statement
- A cursor can process one row at a time
- The set of rows that the cursor holds is called the active set

Cursors

- There are two types of cursors:
  - implicit - created when a DML statement such as `INSERT`, `UPDATE` or `DELETE` is called or a SELECT statement that returns one row
  - explicit - created explicitly for a `SELECT` statement

Cursors

- Implicit cursors allow to assign result of a `SELECT` statement directly to variables.

- Implicit cursors are opened with the

  `SELECT … INTO <variable_list>`

  command

Cursors

- Example:

```
CREATE PROCEDURE implicit_cursor()
BEGIN
    DECLARE V_EMAIL VARCHAR(100);
    SELECT EMAIL FROM EMPLOYEES
    WHERE NAME = 'Ivan Ivanov'
    INTO V_EMAIL;
    SELECT V_EMAIL;
END
```

Cursors

- Explicit cursors have an OPEN-FETCH-CLOSE lifecycle:
  - OPEN - opens the cursor
  - FETCH - reads the next row from the cursor into a record variable
  - CLOSE - closes the cursor

Cursors

```
CREATE PROCEDURE `sample_cursor`()
BEGIN
    DECLARE finished INT DEFAULT 0;
    DECLARE v_name VARCHAR(100);
    DECLARE v_email VARCHAR(100);
    DECLARE emp_cursor CURSOR FOR
        select name, email from employees
        where name like 'M%';
    DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET finished = 1;
    OPEN emp_cursor;
    sample_loop: LOOP
        FETCH emp_cursor INTO v_name, v_email;
        if finished = 1 then leave sample_loop; end if;
        select concat(v_name, '_', v_email);
    END LOOP sample_loop;
    CLOSE emp_cursor;
END
```

Error Handling

Error handling

- Errors can occur during the execution of stored routines - most often returned from an SQL statement executed by the stored routine

- Errors can be handled by error handlers that are basically another stored routines that provided error-handling logic

Error handling

- There are basic three types of errors in a stored routine:
  - named system errors
  - unnamed user-defined errors
  - named user-defined errors

Error handling

- Error handlers can be triggered as a result of errors and the procedure can continue execution after the handler is executed (CONTINUE handlers)

- Example:

```
DECLARE CONTINUE HANDLER FOR 1051
BEGIN
-- body of handler
END;
```

Error handling

- Error handlers can be triggered as a result of errors and the procedure can exit after the handler is executed (EXIT handlers)

- Example:

```
DECLARE EXIT HANDLER FOR 1051
BEGIN
-- body of handler
END;
```

Error handling

- Name of errors can be attached to error codes using the `DECLARE … CONDITION` command

- Example:

```
DECLARE no_such_table CONDITION FOR 1051;
DECLARE CONTINUE HANDLER FOR no_such_table
BEGIN
-- body of handler
END;
```

Questions ?