



Stored Procedures and Triggers in MySQL 5.0

Matthew Montgomery
Sr. Support Engineer
MySQL Support, Database Group
matt.montgomery@sun.com



Agenda

- Stored Procedures & Functions
- Stored Routines in External Languages
- Triggers
- Using Triggers to Emulate Foreign Key Constraints

What is a Stored Procedure

- A stored procedure is a set of SQL statements that are stored in the server and executed as a block.
- Follows SQL:2003 (PSM) syntax also used by IBM DB2 and PostgreSQL 8.4+
- May return many results for one set of input values

Why might I use a Stored Procedure

- Abstract business logic away from clients
- Reduce Rework - Client applications in different languages need to perform the same operations.
- 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.

Stored Procedure Limitations

- Some SQL commands are not allowed:
 - LOAD DATA, (UN)LOCK TABLES, CHECK, REPAIR
- Stored Procedures have all limitations of Stored Functions if called from within a stored function
- UNDO handlers are not supported.
- FOR loops are not supported.
- Identifiers cannot be passed in variables.

Stored Procedure Limitations

- There are no stored routine debugging facilities.
- SQL_MODE changes must be done outside of CREATE PROCEDURE/FUNCTION.

Stored Procedure Syntax

CREATE

```
[DEFINER = { user | CURRENT_USER }] PROCEDURE sp_name ([proc_parameter[,...]])  
[characteristic ...] routine_body
```

proc_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 } | COMMENT 'string'
```

routine_body:

Valid SQL procedure statement

Executing a Stored Procedure

- The CALL statement invokes a procedure that was defined previously with CREATE PROCEDURE.
- Syntax : CALL sp_name([parameter[, ...]]);

ex. CALL kill_run_aways(10);

BEGIN... END syntax

[begin_label:] BEGIN

[statement_list]

END [end_label]

- BEGIN ... END syntax is used for writing compound statements, which can appear within stored routines and triggers.
 - Each statement within statement_list must be terminated by a semicolon (;) statement delimiter.

The DELIMITER command 1/2

- Use DELIMITER or \d to control the which characters the CLI recognizes as command delimiters.
- This allows you to group SQL commands into a single block that is sent to the server all at once.
- \g and \G are always interpreted by the CLI as command delimiters
- The ; is always interpreted by the server as the command delimiter.

The DELIMITER command 2/2

```
mysql> delimiter //  
mysql> show tables like 't1';  
-> show tables like 't2';  
-> //
```

```
+-----+  
| Tables_in_test (t1) |  
+-----+  
| t1                   |  
+-----+  
1 row in set (0.00 sec)
```

```
+-----+  
| Tables_in_test (t2) |  
+-----+  
| t2                   |  
+-----+  
1 row in set (0.00 sec)
```

```
mysql> \d ;  
mysql>
```

Variables in Stored Routines

Local Variables

DECLARE *var_name* *type*

ex. DECLARE name VARCHAR(20);

ex2. DECLARE v1 INT UNSIGNED DEFAULT 10;

Variable SET Statement

SET *a* = *expression* [, *b* = *expression*]...

ex. SET name = 'bob';

ex2. SET v1 = v1+5;

SELECT ... INTO Statement

SELECT *col_name*[,...] INTO *var_name*[,...] *table_expr*

ex. SELECT fname,age INTO name,v1 FROM users
WHERE ... LIMIT 1;

DECLARE Handlers 1/2

The DECLARE... CONDITION FOR statement specifies conditions that need specific handling. It associates a name with a specified error condition.

The DECLARE ... HANDLER statement specifies handlers that each may deal with one or more conditions. If one of these conditions occurs, the specified statement is executed. statement can be a simple statement (for example, SET var_name = value), or it can be a compound statement written using BEGIN and END

DECLARE Handlers 2/2

```
DECLARE out_of_range CONDITION FOR  
SQLSTATE '22003'
```

```
DECLARE CONTINUE HANDLER FOR  
out_of_range SHOW WARNINGS;
```

```
DECLARE CONTINUE HANDLER FOR NOT  
FOUND SET done = 1
```

Cursors

Cursors allow you to iterate over a result set

```
DECLARE cursor_name CURSOR FOR  
    select_statement;
```

```
OPEN cursor_name;
```

```
FETCH cursor_name INTO var_name [,var_name...];
```

```
CLOSE cursor_name;
```

Flow Control 1/2

IF:

```
IF search_condition THEN statement_list  
  [ELSEIF search_condition THEN statement_list] ...  
  [ELSE statement_list]  
END IF
```

CASE:

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

Or:

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

```
END CASE
```

Flow Control 2/2

LOOP:

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

LEAVE:

```
LEAVE label
```

REPEAT:

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

ITERATE:

```
ITERATE label
```

WHILE:

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

Example Stored Procedure

```
DELIMITER //
CREATE PROCEDURE `kill_run_aways` ( IN runtime TINYINT UNSIGNED )
    LANGUAGE SQL
    NOT DETERMINISTIC
    BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE connid INT UNSIGNED;
    DECLARE cur_1 CURSOR FOR SELECT ID FROM `information_schema`.`PROCESSLIST` where CONCAT("'",
USER, "'@'", SUBSTRING_INDEX(HOST,':',1), "'") in ( SELECT `USER_PRIVILEGES`.`GRANTEE` AS
`GRANTEE` from `information_schema`.`USER_PRIVILEGES` where not(`USER_PRIVILEGES`.`GRANTEE` in
(select GRANTEE from `information_schema`.`USER_PRIVILEGES` where
`USER_PRIVILEGES`.`PRIVILEGE_TYPE` = 'SUPER' group by `USER_PRIVILEGES`.`GRANTEE`))) AND COMMAND
='Query' and TIME >= runtime;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
    OPEN cur_1;
    REPEAT
        FETCH cur_1 INTO connid;
        IF NOT done THEN
            KILL connid;
        END IF;
    UNTIL done END REPEAT;
    CLOSE cur_1;
    END //
DELIMITER ;
```

What is a Stored Function

- A stored function is a set of SQL statements that are stored in the server and executed as a block like a stored routine but with some additional limitations.
- A stored function may return only one result for each input value.
- A stored function can be called from within other SQL commands such as SELECT.

Stored Function Limitations 1/2

- All Stored Procedure limitations plus...
- No statements that perform explicit or implicit COMMIT or ROLLBACK like ALTER, CREATE, DROP, LOAD, TRUNCATE, RENAME
 - See: <http://dev.mysql.com/doc/refman/5.0/en/implicit-commit.html>
- Statements that return a result set (i.e. SELECT without a “INTO var_list” clause)
- A function can process a result set if you use a cursor and FETCH statements.

Stored Function Limitations 2/2

- Recursive statements,
 - Functions cannot call themselves.
- Functions are not permitted to modify a table that is already being used (for reading or writing) by the statement that invoked the function or trigger.

Stored Function Syntax

CREATE

```
[DEFINER = { user | CURRENT_USER }]
FUNCTION sp_name ([func_parameter[,...]])
RETURNS type
[characteristic ...] routine_body
```

func_parameter:

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 } | COMMENT '*string*'

routine_body:

Valid SQL procedure statement

RETURN Statement

RETURN *expr*

The RETURN statement terminates execution of a stored function and returns the value *expr* to the function caller. There must be at least one RETURN statement in a stored function. There may be more than one if the function has multiple exit points.

This statement is not used in stored procedures or triggers.

Example Function

```
mysql> CREATE FUNCTION ip_to_hex ( ip_address CHAR(15) )
/* Encode IP into abbreviated HEX format */
RETURNS BINARY(8) NO SQL RETURN CONCAT(
    HEX( CHAR( SUBSTRING_INDEX( SUBSTRING_INDEX(ip_address, '.', 1) , '.', -1))),
    HEX( CHAR( SUBSTRING_INDEX( SUBSTRING_INDEX(ip_address, '.', 2) , '.', -1))),
    HEX( CHAR( SUBSTRING_INDEX( SUBSTRING_INDEX(ip_address, '.', 3) , '.', -1))),
    HEX( CHAR( SUBSTRING_INDEX( SUBSTRING_INDEX(ip_address, '.', 4) , '.', -1))));
```

Query OK, 0 rows affected (0.08 sec)

```
mysql> SELECT ip_to_hex('192.168.23.10');
```

```
+-----+
| ip_to_hex('192.168.23.10') |
+-----+
| C0A8170A                    |
+-----+
```

1 row in set (0.00 sec)

Stored Procedures in Ext. Languages

- Extends MySQL to support external (non-SQL) stored procedures and functions. Support for external language routines are by the use of plugins. Plugins developed include the support for stored procedures in C, Java , Perl, Legacy UDF and XML RPC.
- <http://forge.mysql.com/projects/project.php?id=239>
- Authors:
 - Eric Herman <Eric.Herman@sun.com>
 - Antony Curtis <antony.curtis@ieee.org>

General Purpose Stored Routines Lib.

- This routine library collects general purpose MySQL 5 stored procedures and functions, implementing arrays, for-each loops, syntax helpers, named parameters, unit testing and more.
- <https://sourceforge.net/projects/mysql-sr-lib/>
- Author: Giuseppe Maxia <giuseppe@mysql.com>

Triggers

- A trigger is a named database object that is associated with a table and that is activated when a particular event occurs for the table.
- MySQL triggers are activated by SQL statements only. They are not activated by changes in tables made by APIs that do not transmit SQL statements to the MySQL Server; For example, they are not activated by updates made using the NDB API.

Trigger Limitations 1/2

- Same limitations as for Stored Functions plus...
- Triggers currently are not activated by foreign key actions.
- The RETURN statement is disallowed in triggers, which cannot return a value. To exit a trigger immediately, use the LEAVE statement.
- Triggers are not allowed on tables in the mysql database.

Trigger Limitations 2/2

- There cannot be two triggers for a given table that have the same trigger action time and event. For example, you cannot have two BEFORE UPDATE triggers for a table. But you can have a BEFORE UPDATE and a BEFORE INSERT trigger, or a BEFORE UPDATE and an AFTER UPDATE trigger.

Trigger Syntax

CREATE

[DEFINER = { user | CURRENT_USER }]

TRIGGER trigger_name trigger_time trigger_event

ON tbl_name FOR EACH ROW trigger_stmt

DEFINER = user whose permissions the trigger is executed with

trigger_time = BEFORE | AFTER

trigger_event = INSERT (includes LOAD DATA, REPLACE) | UPDATE |
DELETE (includes REPLACE but not DROP or TRUNCATE)

A potentially confusing example of this is the INSERT INTO ... ON DUPLICATE KEY UPDATE ... syntax: a BEFORE INSERT trigger will activate for every row, followed by either an AFTER INSERT trigger or both the BEFORE UPDATE and AFTER UPDATE triggers, depending on whether there was a duplicate key for the row.

Trigger Permissions Checking

- At CREATE TRIGGER time, the user that issues the statement must have the SUPER privilege.
- At trigger activation time, DEFINER user must have these privileges.
 - The SUPER privilege.
 - The SELECT privilege for the subject table if references to table columns occur via OLD.col_name or NEW.col_name in the trigger definition.
 - The UPDATE privilege for the subject table if table columns are targets of SET NEW.col_name = value assignments in the trigger definition.
 - Whatever other privileges normally are required for the statements executed by the trigger.

Using Triggers to Implement Fks 1/4

```
# Trigger on insertion into child table (MATCH SIMPLE)
delimiter |
create trigger child_on_insert before insert on child for each row
begin
    if not ((new.fk1 is null) or (new.fk2 is null)) then
        if not exists (select * from parent where pk1 = new.fk1 and pk2 =
new.fk2) then
            call my_error_fk_violation();
        end if;
    end if;
end;|
```

Using Triggers to Implement Fks 2/4

```
# The same trigger in case of MATCH FULL
delimiter |
create trigger child_on_insert before insert on child for each row
begin
    if not ((new.fk1 is null) or (new.fk2 is null)) then
        if not exists (select * from parent where pk1 = new.fk1 and pk2 =
new.fk2) then
            call my_error_fk_violation();
        end if;
    elseif not ((new.fk1 is null) and (new.fk2 is null)) then
        call my_error_fk_violation();
    end if;
end;|
```

Using Triggers to Implement Fks 3/4

```
# Trigger on deletion from parent table in case of ON DELETE RESTRICT
delimiter |
create trigger parent_on_delete before delete on parent for each row
begin
    if exists(select * from child where fk1 = old.pk1 and fk2 = old.pk2)
        then
            call my_error_fk_violation();
        end if;
end;|

# Similar trigger for updating parent table (ON UPDATE RESTRICT)
delimiter |
create trigger parent_on_update before update on parent for each row
begin
    if exists (select * from child where fk1 = old.pk1 and fk2 = old.pk2)
        then
            call my_error_fk_violation();
        end if;
end;|
```

Using Triggers to Implement Fks 4/4

```
# Trigger on deletion from parent table in case of ON DELETE SET DEFAULT  
delimiter |
```

```
create trigger parent_on_delete before delete on parent for each row  
begin  
    update child set fk1= default(fk1), fk2= default(fk2) where fk1 =  
        old.pk1 and fk2 = old.pk2;  
end;|
```

```
# Trigger for update for ON UPDATE CASCADE  
delimiter |
```

```
create trigger parent_on_update before update on parent for each row  
begin  
    update child set fk1= new.pk1, fk2= new.pk2 where fk1 = old.pk1 and  
        fk2 = old.pk2;  
end;|
```

Viewing Stored Routines

* No SHOW ROUTINES command

```
SELECT ROUTINE_NAME, ROUTINE_TYPE FROM  
information_schema.ROUTINES WHERE  
ROUTINE_SCHEMA = 'db_name';
```

```
SHOW CREATE PROCEDURE|FUNCTION  
routine_name\G
```

Viewing Stored Triggers

* No SHOW [CREATE] TRIGGER command.

```
SELECT EVENT_OBJECT_TABLE,  
       TRIGGER_NAME, ACTION_TIMING,  
       EVENT_MANIPULATION FROM  
information_schema.TRIGGERS where  
TRIGGER_SCHEMA='db_name';
```

```
SELECT * FROM information_schema.TRIGGERS  
WHERE TRIGGER_SCHEMA='db_name' and  
EVENT_OBJECT_TABLE='table_name';
```