SQL Stored Programs

Procedures, functions, triggers

A *stored function* is a stored program that returns a value.

FUNCTIONS

Stored Functions

- Stored functions are stored procedures that must always return a value.
- They differ from procedures in the following ways:
  - The parameter list of a function may contain only IN parameters. OUT and INOUT parameters are not allowed. Specifying the IN keyword is neither required nor allowed.
  - The function itself must return a single value, whose type is defined in the header of the function.
    - While stored procedures may return values via OUT or INOUT variables, a function must return data via a single RETURN value.

Stored Functions

- Functions can be called from within SQL statements.
  - Stored functions can be used in expressions wherever you can use a built-in function of the same return data type and can be used inside of SQL statements such as SELECT, UPDATE, DELETE, and INSERT
  - A function may not return a result set. The use of stored functions can improve the readability and maintainability of stored program code by encapsulating commonly used business rules or formulas.
- Stored function return values can be used to control the overall program flow.
- Using stored functions in standard SQL statements can simplify the syntax of the SQL by hiding complex calculations and avoiding the repetitive coding.

Creating Stored Functions

- CREATE FUNCTION *function_name* ([*parameter[,...]*])
  RETURNS *datatype*
  [LANGUAGE SQL]
  [[NOT] DETERMINISTIC]
  [{ CONTAINS SQL|NO SQL|MODIFIES SQL DATA|READS SQL DATA}] [SQL SECURITY {DEFINER|INVOKER}]
  [COMMENT comment_string]
  function_statements

Creating Stored Functions

- Most of the options for the CREATE FUNCTION statement also apply to CREATE PROCEDURE. However, the following are unique to stored functions:
  - The RETURNS clause is mandatory and defines the data type that the function will return.
  - You cannot specify the IN, OUT, or INOUT modifiers to parameters. All parameters are implicitly IN parameters.
  - The function body must contain one or more RETURN statements, which terminate function execution and return the specified result to the calling program.

Example 10-1. Simple stored function with multiple RETURN statements

```
CREATE FUNCTION cust_status(in_status CHAR(1))
    RETURNS VARCHAR(20)
BEGIN
    IF in_status = 'O' THEN
        RETURN('Overdue');
    ELSEIF in_status = 'U' THEN
        RETURN('Up to date');
    ELSEIF in_status = 'N' THEN
        RETURN('New');
    END IF;
END;
```

```
CREATE FUNCTION f_discount_price
      (normal_price NUMERIC(8,2))
   RETURNS NUMERIC(8,2)
BEGIN

    DECLARE discount_price NUMERIC(8,2);

    IF (normal_price>500) THEN
       SET discount_price=normal_price*.8;

    ELSEIF (normal_price>100) THEN
       SET discount_price=normal_price*.9;

    ELSE
       SET discount_price=normal_price;

    END IF;

    RETURN(discount_price);

END;
```

The RETURN Statement

- The RETURN statement terminates stored function execution and returns the specified value to the calling program.
- You can have as many RETURN statements in your stored function as makes sense.

Example 10-1. Simple stored function with multiple RETURN statements

```
CREATE FUNCTION cust_status(in_status CHAR(1))
    RETURNS VARCHAR(20)
BEGIN
    IF in_status = 'O' THEN
        RETURN('Overdue');
    ELSEIF in_status = 'U' THEN
        RETURN('Up to date');
    ELSEIF in_status = 'N' THEN
        RETURN('New');
    END IF;
END;
```

The RETURN Statement

- Avoid the situation in which none of the RETURN statements gets executed.
- “Falling out” of a function, rather than exiting cleanly via a RETURN statement, will cause a runtime error.
- Consider including only a single RETURN statement (“one way in and one way out”), and to use variable assignments within conditional statements to change the return value.

Example 10-3. Simple stored function with single RETURN statement

```
CREATE FUNCTION cust_status(in_status CHAR(1))
    RETURNS VARCHAR(20)
BEGIN
    DECLARE long_status VARCHAR(20);

    IF in_status = 'O' THEN
        SET long_status='Overdue';
    ELSEIF in_status = 'U' THEN
        SET long_status='Up to date';
    ELSEIF in_status = 'N' THEN
        SET long_status='New';
    END IF;

    RETURN(long_status);
END;
```

DETERMINISTIC and SQL Clauses

- When binary logging is enabled, MySQL needs to know if a stored function that modifies SQL is deterministic—that is, if it always performs the same actions and returns the same results when provided with the same inputs
- The default for stored programs is NOT DETERMINISTIC CONTAINS SQL
  - You need to explicitly set the appropriate keywords in order for the function to compile when binary logging is enabled
  - This requirement relates to the need to ensure that changes made in the stored function can be correctly replicated to another server
  - If the actions performed by the function are nondeterministic, then correct replication cannot be assured

DETERMINISTIC and SQL Clauses

- If you declare a stored function without one of the SQL mode clauses NO SQL or READS SQL, and if you have not specified the DETERMINISTIC clause, *and* if the binary log is enabled, you may receive the following error:
  - ERROR 1418 (HY000): This function has none of DETERMINISTIC, NO SQL, or READS SQL DATA in its declaration and binary logging is enabled (you *might* want to use the less safe log_bin_trust_function_creators variable)
- To avoid this error, you must do one of the following:
  - Specify one or more of the DETERMINISTIC, NO SQL, and/or READS SQL DATA key- words in your stored function definition.
  - Set the value of log_bin_trust_routine_creators to 1 (SET GLOBAL log_bin_trust_ routine_creators = 1)

SQL Statements in Stored Functions

- You can include SQL statements within stored functions, although you should be very careful about including SQL statements in a stored function that might itself be used inside a SQL statement.
- You cannot return a result set from a stored function: trying to create a stored function that contains a SELECT statement without an INTO clause will result in a 1415 error.

Example 10-5. Stored functions cannot return result sets

```
mysql> CREATE FUNCTION test_func( )
    -> RETURNS INT
    -> BEGIN
    -> SELECT 'Hello World';
    -> RETURN 1;
    -> END;$$
ERROR 1415 (OA000): Not allowed to return a result set from a function
```

Calling Stored Functions

- A function can be called by specifying its name and parameter list wherever an expression of the appropriate data type may be used.

Calling a function from command line

Example 10-7. Calling a stored function from the MySQL command line

```
mysql> SET @x=isodd(42);
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @x;
+------+
| @x   |
+------+
| 0    |
+------+
1 row in set (0.02 sec)

mysql> SELECT isodd(42)
    -> ;
+-----------+
| isodd(42) |
+-----------+
|         0 |
+-----------+
```

Calling a function from procedure

Example 10-8. Calling a stored function from within a stored procedure

```
SET l_isodd=isodd(aNumber);

IF (isodd(aNumber)) THEN
    SELECT CONCAT(aNumber," is odd") as isodd;
ELSE
    SELECT CONCAT(aNumber," is even") AS isodd;
END IF;
```

Using Stored Functions in SQL

- Stored functions have an additional and significant role to play: as *user-defined functions* (UDFs) within SQL statements.

Example 10-11. SQL statement with multiple CASE statements

```
SELECT CASE customer_status
            WHEN 'U' THEN 'Up to Date'
            WHEN 'N' THEN 'New'
            WHEN 'O' THEN 'Overdue'
       END  as Status, count(*) as Count
  FROM customers
 GROUP BY customer_status
 ORDER BY CASE customer_status
            WHEN 'U' THEN 'Up to Date'
            WHEN 'N' THEN 'New'
            WHEN 'O' THEN 'Overdue'
       END
```

Example 10-12. Stored function for use in our SQL statement

```
CREATE FUNCTION cust_status(IN in_status CHAR(1))
    RETURNS VARCHAR(20)
BEGIN
    DECLARE long_status VARCHAR(20);

    IF in_status = 'O' THEN
         SET long_status='Overdue';
    ELSEIF in_status = 'U' THEN
         SET long_status='Up to date';
    ELSEIF in_status = 'N' THEN
         SET long_status='New';
    END IF;

     RETURN(long_status);
END;
```

Calling a function from SQL statement

Example 10-13. Stored function in a SQL statement

```
SELECT cust_status(customer_status) as Status, count(*) as Count
  FROM customers
 GROUP BY customer_status
 ORDER BY cust_status(customer_status);
```

Using SQL in Stored Functions

- SQL statements can be included inside of stored functions that are themselves used within SQL statements as user-defined functions.
- However, functions calling SQL inside of SQL statements can lead to unpredictable and often poor performance.

Example 10-14. Stored function to return customer count for a sales rep

```
CREATE FUNCTION customers_for_rep(in_rep_id INT)
    RETURNS INT
    READS SQL DATA
BEGIN
    DECLARE customer_count INT;

    SELECT COUNT(*)
      INTO customer_count
      FROM customers
      WHERE sales_rep_id=in_rep_id;

    RETURN(customer_count);

END;
```

Example 10-15. Using the sales rep function in a stored program

```
IF customers_for_rep(in_employee_id) > 0 THEN
     CALL calc_sales_rep_bonus(in_employee_id);
ELSE
     CALL calc_nonrep_bonus(in_employee_id);
END IF;
```

Example 10-16. Standard SQL to retrieve sales reps with more than 10 customers

```
SELECT employee_id,COUNT(*)
  FROM employees JOIN customers
    ON (employee_id=sales_rep_id)
 GROUP BY employee_id
 HAVING COUNT(*) > 10
 ORDER BY COUNT(*) desc;
```

Example 10-17. Function-based query to retrieve sales reps with more than 10 customers

```
SELECT employee_id,customers_for_rep(employee_id)
  FROM employees
 WHERE customers_for_rep(employee_id)>10
 ORDER BY customers_for_rep(employee_id) desc
```

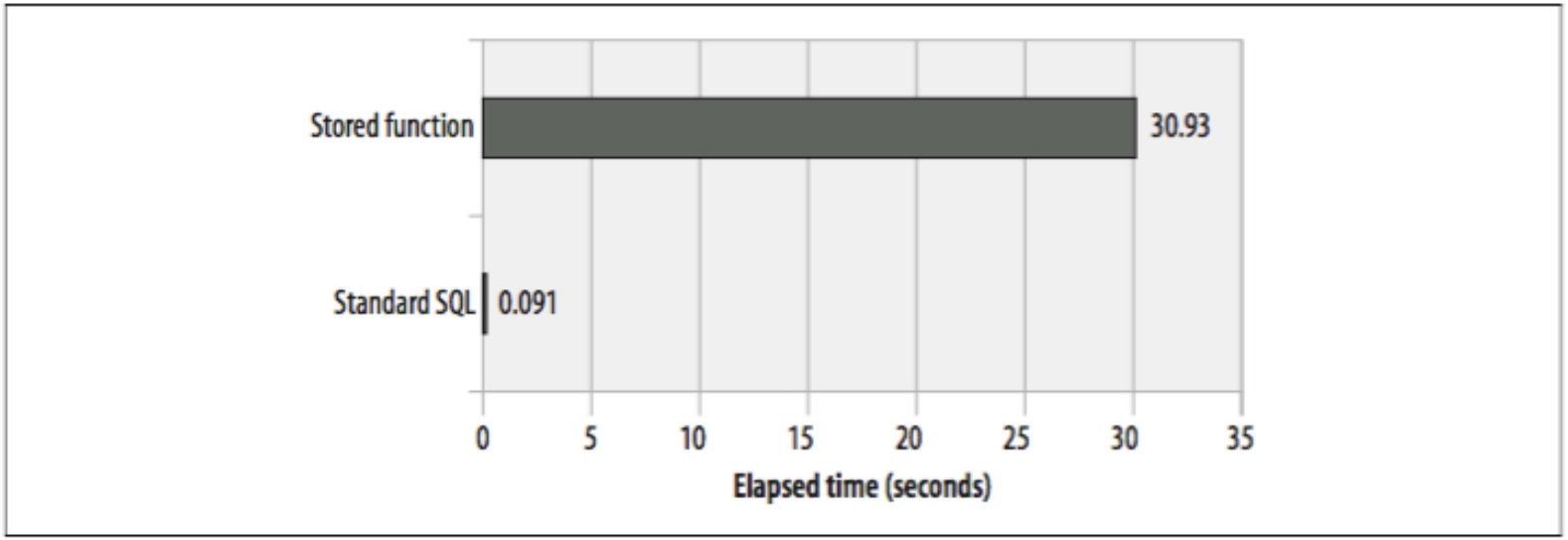

Figure 10-1. Comparison of performance between standard SQL and SQL using a stored function containing embedded SQL

Database *triggers* are stored programs that are executed in response to some kind of event that occurs within the database.

TRIGGERS

Triggers

- A trigger is a special type of stored program that fires when a table is modified by an INSERT, UPDATE, or DELETE (DML) statement.
- Triggers are a powerful mechanism for ensuring the integrity of data, as well as a useful means of automating certain operations in the database.

Creating Triggers

- CREATE [DEFINER = { *user*|CURRENT_USER }] TRIGGER *trigger_name* {BEFORE|AFTER} {UPDATE|INSERT|DELETE} ON *table_name* FOR EACH ROW *trigger_statements*

Triggers

- DEFINER
  - This optional clause specifies the security privileges that the trigger code will assume when it is invoked. The default CURRENT_USER setting results in the trigger executing with the privileges of the account that executes the CREATE TRIGGER statement. Specifying a *user* allows the trigger to execute with the privileges of another account.
- *trigger_name*
  - The trigger name follows the normal conventions for MySQL's naming of database objects. A sensible convention might result in triggers being given names such as table_name_bu (for a BEFORE UPDATE trigger) or table_name_ai (for an AFTER INSERT trigger).
- BEFORE or AFTER
  - These clauses control the sequence in which the trigger will fire—either before or after the triggering DML statement is executed.

Triggers

- UPDATE, INSERT, *or* DELETE
  - These clauses specify the type of DML statement that will cause the trigger to be invoked.
- ON *table_name*
  - Associates the trigger with a specific table.
- FOR EACH ROW
  - This clause is mandatory in the initial MySQL implementation. It indicates that the trigger will be executed once for every row affected by the DML statement. The ANSI standard also provides for a FOR EACH STATEMENT mode, which might be supported in an upcoming version of MySQL.
- *trigger_statements*
  - This code can be one or more stored program language statements. If more than one statement is specified, they must all be contained within a BEGIN-END block.

```
CREATE TRIGGER sales_bi_trg
     BEFORE INSERT ON sales
     FOR EACH ROW
BEGIN
     IF NEW.sale_value > 500 THEN
        SET NEW.free_shipping='Y';
     ELSE
        SET NEW.free_shipping='N';
     END IF;
     IF NEW.sale_value > 1000 THEN
        SET NEW.discount=NEW.sale_value*.15;
         ELSE
        SET NEW.discount=0;
     END IF;
END;
```

Referring to Column Values Within the Trigger

- Trigger statements can include references to the values of the columns being affected by the trigger.
  - You can access and sometimes modify the values of these columns.
- To distinguish between the values of the columns “before” and “after” the relevant DML has fired, use the NEW and OLD modifiers.
  - For instance, in a BEFORE UPDATE trigger, the value of the column mycolumn before the update is applied is OLD. mycolumn, and the value after modification is NEW.mycolumn.
  - If the trigger is an INSERT trigger, only the NEW value is available (there is no OLD value). Within a DELETE trigger, only the OLD value is available (there is no NEW value).
- Within BEFORE triggers you can modify a NEW value with a SET statement—thus changing the effect of the DML.

Triggering Actions

- Triggers will normally execute in response to the DML statements matching their specification—for instance, BEFORE INSERT will always be invoked in response to an INSERT statement.
- However, triggers also fire in response to implicit—as well as explicit—DML.
  - For instance, an INSERT statement that contains an ON DUPLICATE KEY UPDATE clause can issue an implicit UPDATE statement causing BEFORE UPDATE or AFTER UPDATE triggers to fire.
  - Cascaded foreign key actions do not activate triggers.

BEFORE and AFTER Triggers

- The most significant difference between BEFORE and AFTER triggers is that in an AFTER trigger you are not able to modify the values about to be inserted into or updated with the table in question
  - The DML has executed, and it is too late to try to change what the DML is going to do.

- IF you try to modify a NEW value in an AFTER trigger, you will encounter an error

- Use AFTER triggers for activities that logically should occur in a transaction after a DML has successfully executed.
  - Auditing activities, for example, are best executed in an AFTER trigger, since you will first want to make sure that the DML succeeded.

Using Triggers

- Triggers can be used to implement a variety of useful requirements, such as automating the maintenance of denormalized or derived data, implementing logging, and validating data.
- EXAMPLE: Maintaining derived data
  - We often need to maintain redundant "denormalized" information in our tables to optimize critical SQL queries.
  - The code to perform this denormalization *could* be placed within the application code, but then you would have to make sure that any and every application module that modifies the table also performs the denormalization.
  - If you want to *guarantee* that this code is run whenever a change is made to the table, you can attach that functionality to the table itself, via a trigger.

Example 11-2. Using triggers to maintain denormalized data

```
DELIMITER $$

CREATE TRIGGER sales_bi_trg
  BEFORE INSERT ON sales
  FOR EACH ROW
BEGIN
  DECLARE row_count INTEGER;

  SELECT COUNT(*)
    INTO row_count
    FROM customer_sales_totals
   WHERE customer_id=NEW.customer_id;

  IF row_count > 0 THEN
    UPDATE customer_sales_totals
       SET sale_value=sale_value+NEW.sale_value
     WHERE customer_id=NEW.customer_id;
  ELSE
    INSERT INTO customer_sales_totals
       (customer_id,sale_value)
     VALUES(NEW.customer_id,NEW.sale_value);
  END IF;

END$$
```

Example 11-2. Using triggers to maintain denormalized data (continued)

```
  BEFORE UPDATE ON sales
  FOR EACH ROW
BEGIN

  UPDATE customer_sales_totals
     SET sale_value=sale_value+(NEW.sale_value-OLD.sale_value)
   WHERE customer_id=NEW.customer_id;

END$$

CREATE  TRIGGER sales_bd_trg
  BEFORE DELETE ON sales
  FOR EACH ROW
BEGIN

  UPDATE customer_sales_totals
     SET sale_value=sale_value-OLD.sale_value
   WHERE customer_id=OLD.customer_id;

END$$
```

Using Triggers

- **Implementing Logging**

Example 11-3. Using triggers to implement audit logging

```
CREATE TRIGGER account_balance_au
 AFTER UPDATE ON account_balance FOR EACH ROW
 BEGIN
        INSERT into transaction_log
               (user_id, description)
        VALUES (user( ),
                CONCAT('Adjusted account ',
                   NEW.account_id,' from ',OLD.balance,
                   ' to ', NEW.balance));
END;
```

Validating Data with Triggers

- A typical and traditional use of triggers in relational databases is to validate data or implement business rules to ensure that the data in the database is logically consistent and does not violate the rules of the business or the application.
- Data validation triggers may perform tasks such as:
  - *Implementing checks on allowable values for columns*
    - For instance, a percentage value must fall between 0 and 100, a date of birth cannot be greater than today's date, and so on.
  - *Performing cross-column or cross-table validations*
    - For example, an employee cannot be his own manager, a sales person must have an associated quota, and seafood pizzas cannot include anchovies.
  - *Performing advanced referential integrity*
    - Referential constraints are usually best implemented using foreign key constraints; sometimes, however, you may have some advanced referential integrity that can only be implemented using triggers. For instance, a foreign key column may be required to match a primary key in one of a number of tables (an *arc* relationship).
- A data validation trigger typically prevents a DML operation from completing if it would result in some kind of validation check failing.

Example 11-4. ANSI-standard trigger to enforce a business rule

```
CREATE TRIGGER account_balance_bu
    BEFORE UPDATE
     ON account_balance
    FOR EACH ROW
BEGIN
     -- The account balance cannot be set to a negative value.
     IF (NEW.balance < 0) THEN

         SIGNAL SQLSTATE '80000'
             SET MESSAGE_TEXT='Account balance cannot be less than 0';
     END IF;
END;
```

To signal a generic SQLSTATE value, use '45000', which means “unhandled user-defined exception.”

Trigger Overhead

- By necessity, triggers add overhead to the DML statements to which they apply.
- The actual amount of overhead will depend upon the nature of the trigger, but—as all MySQL triggers execute FOR EACH ROW—the overhead can rapidly accumulate for statements that process large numbers of rows.
- Avoid placing any expensive SQL statements or procedural code in triggers.

Triggers

- Triggers implement functionality that must take place whenever a certain change occurs to the table.
- Because triggers are attached directly to the table, application code cannot bypass database triggers.
- Typical uses of triggers include the implementation of critical business logic, the denormalization of data for performance reasons, and the auditing of changes made to a table.
- Triggers can be defined to fire before or after a specific DML statement executes.