BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI, RAJASTHAN, 333031

Advanced Database Systems
[CS G516]

WORKING WITH USER DEFINE FUNCTIONS(UDF)

Prerequisites

1. Strong understanding of relational model concepts (relations, tuples, attributes)
2. SQL DML operations: `SELECT`, `INSERT`, `UPDATE`, `DELETE`
3. Stored procedures and basic procedural SQL concepts
4. Familiarity with normalization and transaction properties (ACID)
5. MySQL 8.0+ server with appropriate privileges (`CREATE FUNCTION`, `CREATE TRIGGER`)

Introduction

User Defined Functions (UDFs)

A User Defined Function (UDF) in MySQL is a stored program unit that encapsulates reusable computational logic and returns exactly one value to the calling SQL expression. From a relational theory perspective, a scalar UDF behaves like a mathematical function: for a given input tuple (parameter set), it deterministically produces a single output value.

UDFs are primarily used to:

- Encapsulate repetitive calculations
- Improve query readability and maintainability
- Enforce derived attribute logic consistently

Unlike stored procedures, UDFs can be invoked inside SQL queries, such as within `SELECT`, `WHERE`, `HAVING`, or `ORDER BY` clauses.

MySQL supports only Scalar UDFs. It does not support table-valued functions as found in SQL Server or PostgreSQL.

Triggers

A trigger is a database object that is automatically executed in response to a data modification event on a relation. Triggers execute as part of the same transaction that fired them, thereby fully participating in MySQL's ACID properties.

Triggers are used to:

- Enforce complex business rules not expressible via constraints
- Maintain derived or historical attributes
- Prevent invalid state transitions in data

In MySQL, triggers are defined at the row level (`FOR EACH ROW`) and support two timings:

- `BEFORE` – executed prior to the DML operation
- `AFTER` – executed after the DML operation

MySQL Syntax and Implementation

Scalar User Defined Functions

Sample Schema Used in This Lab

The following relations will be used throughout this lab.

```
CREATE TABLE vendors (
vendorid CHAR(5) PRIMARY KEY,
companyname VARCHAR(50) NOT NULL
);
```

```
CREATE TABLE ingredients (
ingredientid CHAR(5) PRIMARY KEY,
name VARCHAR(50) NOT NULL,
unitprice DECIMAL(5,2) NOT NULL,
vendorid CHAR(5),
FOREIGN KEY (vendorid) REFERENCES vendors(vendorid)
);
```

Sample Data

```
INSERT INTO vendors VALUES
('VGRUS', 'Veggies_R_Us'),
```

```
('DNDRY', 'Dairy_N_Delight');

INSERT INTO ingredients VALUES
('I001', 'Tomato', 0.40, 'VGRUS'),
('I002', 'Potato', 0.30, 'VGRUS'),
('I003', 'Cheese', 0.85, 'DNDRY');
```

General Syntax (MySQL)

```
DELIMITER $$

CREATE FUNCTION function_name (
        param_name datatype
)
RETURNS return_datatype
DETERMINISTIC
BEGIN
RETURN expression;
END$$

DELIMITER ;
```

Example 1: Scalar Function – Total Ingredient Cost by Vendor

Problem Statement: Compute the total unit price of all ingredients supplied by a specific vendor. This function models an **aggregate over a relation restricted by a predicate**.

```
DELIMITER $$

CREATE FUNCTION fn_sumprice(v_vendorId CHAR(5))
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
        DECLARE total_sum DECIMAL(10,2);

        SELECT IFNULL(SUM(unitprice), 0)
        INTO total_sum
        FROM ingredients
        WHERE vendorid = v_vendorId;
```

```
        RETURN total_sum;
END$$

DELIMITER ;
```

Execution Examples:

```
SELECT fn_sumprice('VGRUS') AS total_price;
SELECT fn_sumprice('DNDRY') AS total_price;
```

Built-in Scalar Functions (MySQL)

Function	Description
UPPER()	Convert text to uppercase
LOWER()	Convert text to lowercase
SUBSTRING()	Extract substring
LENGTH()	Length of string
ROUND()	Numeric rounding
NOW()	Current timestamp
FORMAT()	Format numeric output

Table-Valued Logic (MySQL-Compatible Alternatives)

MySQL does **not** support RETURNS TABLE functions.

Alternative 1: Using a View

Requirement: List all ingredients supplied by a given vendor.

```
CREATE VIEW vw_vendor_items AS
SELECT i.ingredientid, i.name, v.companyname
FROM ingredients i
JOIN vendors v ON v.vendorid = i.vendorid;
```

Usage:

```
SELECT ingredientid, name
```

```
FROM vw_vendor_items
WHERE companyname = 'Veggies_R_Us';
```

Alternative 2: Stored Procedure with Conditional Logic

Important: This procedure depends on the existence of a menuitems relation. The error you encountered occurs when this relation is missing.

Required Relation: menuitems

```
CREATE TABLE menuitems (
menuitemid CHAR(5) PRIMARY KEY,
name VARCHAR(50) NOT NULL,
price DECIMAL(6,2) NOT NULL
);
```

Sample Data

```
INSERT INTO menuitems VALUES
('MI001', 'Veg Burger', 40.00),
('MI002', 'Cheese Pizza', 80.00),
('MI003', 'Pasta', 60.00);
```

Stored Procedure Definition

```
CALL sp_discount_items(50.00, 0.10);
```

Triggers in MySQL

Additional Schema for Trigger Examples

```
CREATE TABLE items (
        itemid CHAR(5) PRIMARY KEY,
        name VARCHAR(50),
        price DECIMAL(6,2),
        prevprice DECIMAL(6,2)
);
```

```
CREATE TABLE madewith (
        itemid CHAR(5),
        ingredientid CHAR(5),
        quantity INT,
        PRIMARY KEY (itemid, ingredientid),
        FOREIGN KEY (itemid) REFERENCES items(itemid),
        FOREIGN KEY (ingredientid) REFERENCES ingredients(ingredientid)
);
```

Sample Data

```
INSERT INTO items VALUES
```

```
('M001', 'Veg Sandwich', 20.00, NULL);

INSERT INTO madewith VALUES
('M001', 'I001', 2),
('M001', 'I002', 3);
```

General Syntax

```
DELIMITER $$

CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON table_name
FOR EACH ROW
BEGIN
	-- trigger logic
END$$

DELIMITER ;
```

Example 1: Automatic Price Markup Trigger

Business Rule: Item price is always twice the sum of ingredient costs.

```
DELIMITER $$

CREATE TRIGGER trg_markup_price
AFTER UPDATE ON ingredients
FOR EACH ROW
BEGIN
	UPDATE items it
	SET it.price = (
		SELECT 2 * SUM(m.quantity * i.unitprice)
		FROM madewith m
		JOIN ingredients i ON m.ingredientid = i.ingredientid
		WHERE m.itemid = it.itemid
	);
END$$

DELIMITER ;
```

Example 2: Validate Inserted Ingredient Price

Constraint: Ingredient unit price must not exceed 0.9.

```
DELIMITER $$
```

```
CREATE TRIGGER trg_check_unitprice
BEFORE INSERT ON ingredients
FOR EACH ROW
BEGIN
	IF NEW.unitprice > 0.9 THEN
		SIGNAL SQLSTATE '45000'
		SET MESSAGE_TEXT = 'Unit price cannot be greater than 0.9';
	END IF;
END$$

DELIMITER ;
```

Example 3: Backup Old Price Before Update

Schema Modification:

```
ALTER TABLE items ADD prevprice DECIMAL(5,2);
```

Trigger Definition:

```
DELIMITER $$

CREATE TRIGGER trg_backup_item_price
BEFORE UPDATE ON items
FOR EACH ROW
BEGIN
	IF NEW.price = 0 THEN
		SIGNAL SQLSTATE '45000'
		SET MESSAGE_TEXT = 'Price cannot be zero';
	END IF;

	SET NEW.prevprice = OLD.price;
END$$

DELIMITER ;
```

Dropping a Trigger

```
DROP TRIGGER trigger_name;
```

Exercise

Create the following tables

1. **VENDORS**(**vendorid**:char(5),**companyname**:varchar(50),**city**:varchar(40))

2. **INGREDIENTS**(**ingredientid**:char(5),**name**:varchar(50),**unitprice**:decimal(5,2), **vendorid**:char(5))
3. **ITEMS**(**itemid**:char(5),**name**:varchar(50),**price**:decimal(6,2),**prevprice**:decimal(6,2))
4. **MADEWITH**(**itemid**:char(5),**ingredientid**:char(5),**quantity**:numeric)

Enter records from Lab2.2.sql in these tables and maintain all the foreign key constraints. Solve the following problems.

1. Create a scalar user defined function that returns the final price of a specific meal after applying a given discount percentage.
2. Create a scalar user defined function that returns the average number of ingredients used per meal across all meals.
3. Create a database object (view or equivalent MySQL alternative) that returns a relation containing the meal(s) with the highest price and the lowest price.
4. Create a stored program unit that accepts an integer `n` and returns the top `n` highest priced meals and top `n` lowest priced meals in a single result set.
5. Create a trigger on the `items` table that allows insertion of a new meal only if Average Meal Price−3≤New Meal Price≤Average Meal Price+3
6. Create a trigger on the `madewith` table that automatically recalculates and updates the corresponding meal price whenever the ingredient quantity is updated.
7. Create a trigger on the `ingredients` table that prevents insertion or update of an ingredient if its unit price exceeds `0.90`.
8. Create a scalar user defined function that returns the total ingredient cost of a given meal, computed using ingredient quantity and unit price.
9. Create a trigger on the `ingredients` table that updates meal prices automatically whenever an ingredient's unit price is modified, assuming a 100% markup policy.
10. Create a scalar user defined function that returns the vendor-wise total ingredient supply cost for a given vendor identifier.

11. Create a trigger on the `items` table that stores the previous price of a meal before any price update and rejects the update if the new price is zero.
12. Create a stored procedure that lists all meals along with their discounted prices, where meals priced above a given threshold receive a specified discount.
13. Create a trigger that prevents deletion of a meal if it is referenced in the `madewith` relation.
14. Create a scalar user defined function that returns the count of vendors supplying ingredients to a given meal.
15. Create a trigger-based audit mechanism that logs every meal price change with old price, new price, and timestamp into a separate audit table.

References

1. **MySQL Stored Objects Overview**
   https://dev.mysql.com/doc/refman/8.0/en/stored-objects.html
2. **CREATE FUNCTION (User-Defined Functions)**
   https://dev.mysql.com/doc/refman/8.0/en/create-function.html
3. **CREATE PROCEDURE / Stored Procedures**
   https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html
4. **CREATE TRIGGER**
   https://dev.mysql.com/doc/refman/8.0/en/create-trigger.html
5. **CREATE EVENT (Scheduled Tasks)**
   https://dev.mysql.com/doc/refman/8.0/en/create-event.html
6. **CREATE VIEW**
   https://dev.mysql.com/doc/refman/8.0/en/create-view.html
7. **Flow Control Statements (IF, CASE, LOOP, WHILE)**
   https://dev.mysql.com/doc/refman/8.0/en/flow-control-statements.html
8. **DECLARE, Variables, Handlers**
   https://dev.mysql.com/doc/refman/8.0/en/declare.html

9. **Stored Procedures & Functions**

   https://www.mysqltutorial.org/mysql-stored-procedure/

   https://www.mysqltutorial.org/mysql-user-defined-functions/

10. **Triggers**

    https://www.mysqltutorial.org/mysql-triggers/

11. **Views**

    https://www.mysqltutorial.org/mysql-views/