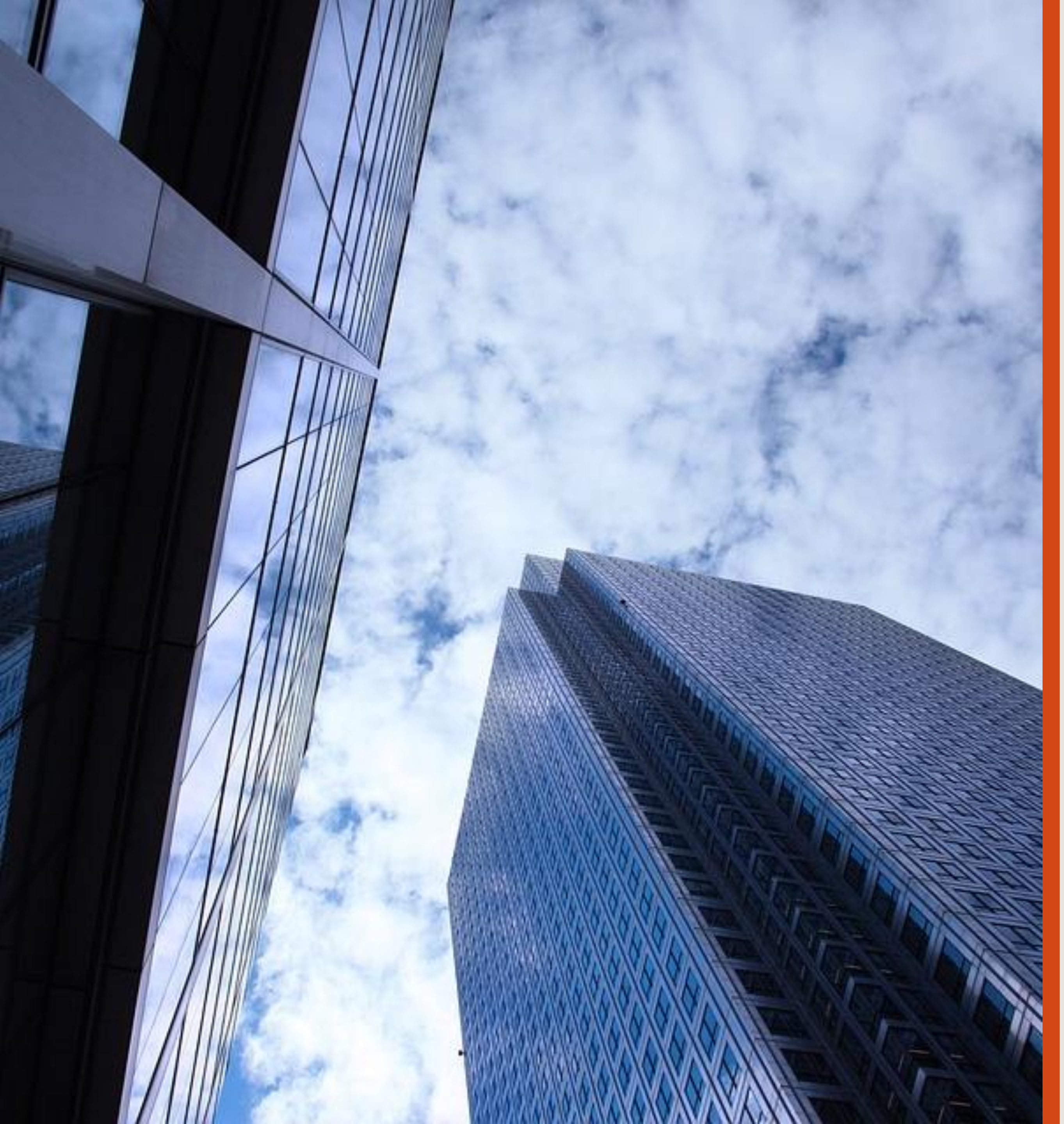


SQL
PART 07

Jayathma Chathurangani
ejc@ucsc.cmb.ac.lk

OUTLINE

- ✓ **Stored Procedures**
  - **What**
  - **Parameters and Variables**
  - **Conditional Statements and Loops**
  - **Error Handling**
  - **Cursors and Prepared Statements**
  - **Nested Procedures**
- ✓ **Triggers**

1

INTRODUCTION

1.1 INTRODUCTION

- Typically, when working with a relational database, you issue individual Structured Query Language (SQL) queries to retrieve or manipulate data, like SELECT, INSERT, UPDATE or DELETE, directly from within your application code.
- Those statements work on and manipulate underlying database tables directly.
- If the same statements or group of statements are used within multiple applications accessing the same database, they are often duplicated in individual applications.
- MySQL, similar to many other relational database management systems, supports the use of stored procedures.
- **Stored procedures help group one or multiple SQL statements for reuse under a common name, encapsulating common business logic within the database itself. Such a procedure can be called from the application that accesses the database to retrieve or manipulate data in a consistent way.**
- Using stored procedures, you can create reusable routines for common tasks to be used across multiple applications, provide data validation, or deliver an additional layer of data access security by restricting database users from accessing the underlying tables directly and issuing arbitrary queries.

1.2 WHAT?

- Stored procedures in MySQL (It was first introduced in MySQL version 5) and in many other relational database systems are named objects that contain one or more instructions laid out and then executed by the database in a sequence when called.
- In the most basic example, a stored procedure can save a common statement under a reusable routine, such as retrieving data from the database with often-used filters.

  For example, If we consider the enterprise application, we always need to perform specific tasks such as database cleanup, processing payroll, and many more on the database regularly. Such tasks involve multiple SQL statements for executing each task. This process might easy if we group these tasks into a single task. We can fulfill this requirement in MySQL by creating a stored procedure in our database.

- A procedure is called a recursive stored procedure when it calls itself. Most database systems support recursive stored procedures. But, it is not supported well in MySQL.

1.3 STORED PROCEDURE FEATURES

- **Stored Procedure increases the performance of the applications.** Once stored procedures are created, they are compiled and stored in the database.
- **Stored procedure reduces the traffic between application and database server.** Because the application has to send only the stored procedure's name and parameters instead of sending multiple SQL statements. Stored procedures are executed directly on the database server, performing all computations locally and returning results to the calling user only when finished.
- Stored procedures **are reusable and transparent** to any applications.
- A procedure is always **secure**. The database administrator can grant permissions to applications that access stored procedures in the database without giving any permissions on the database tables.
- When the procedure is called by its name, the database engine executes it as defined, **instruction by instruction**.

1.4 MORE..

- Parameters passed to the stored procedure or returned through it.
- Declared variables to process retrieved data directly within the procedure code.
- Conditional statements, which allow the execution of parts of the stored procedure code depending on certain conditions, such as IF or CASE instructions.
- Loops, such as WHILE, LOOP, and REPEAT, allow executing parts of the code multiple times, such as for each row in a retrieved data set.
- Error handling instructions, such as returning error messages to the database users accessing the procedure.
- Calls to other stored procedures in the database.

1.5 HOW TO CREATE A PROCEDURE?

```
DELIMITER &&
CREATE PROCEDURE procedure_name [[IN | OUT | INOUT] parameter_name
datatype [, parameter datatype]) ]
BEGIN
    Declaration_section
    Executable_section
END &&
DELIMITER ;
```

- procedure_name: It represents the name of the stored procedure.
- Parameter: It represents the number of parameters. It can be one or more than one.
- Declaration_section: It represents the declarations of all variables.
- Executable_section: It represents the code for the function execution.

1.6 SHOW AND DROP PROCEDURES

How to show or list stored procedures in MySQL?

- SHOW PROCEDURE STATUS [LIKE 'pattern' | WHERE search_condition]
- SHOW PROCEDURE STATUS WHERE db = 'mystudentdb';

How to delete/drop stored procedures in MySQL?

- DROP PROCEDURE [IF EXISTS] procedure_name;
- DROP PROCEDURE display_marks;

2

PARAMETERS AND VARIABLES

2.1 INTRODUCTION- MODES IN PROCEDURE PARAMETERS

IN parameter

- It is the default mode. It takes a parameter as input, such as an attribute. When we define it, the calling program has to pass an argument to the stored procedure. This parameter's value is always protected

OUT parameters

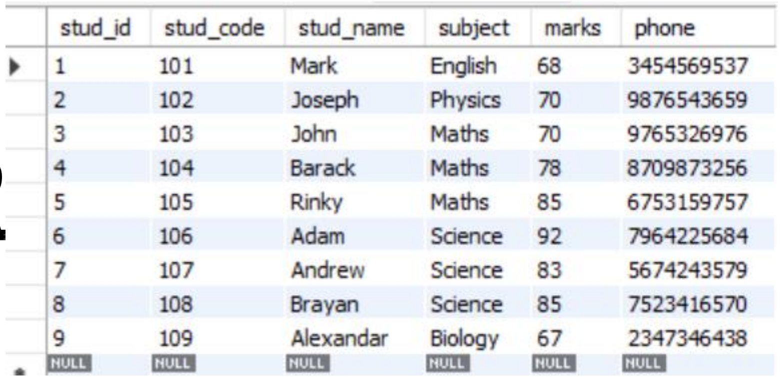
- It is used to pass a parameter as output. Its value can be changed inside the stored procedure, and the changed (new) value is passed back to the calling program. It is noted that a procedure cannot access the OUT parameter's initial value when it starts.

INOUT parameters

- It is a combination of IN and OUT parameters. It means the calling program can pass the argument, and the procedure can modify the INOUT parameter, and then passes the new value back to the calling program.

How to call a Procedure?

CALL procedure_name (parameter(s))

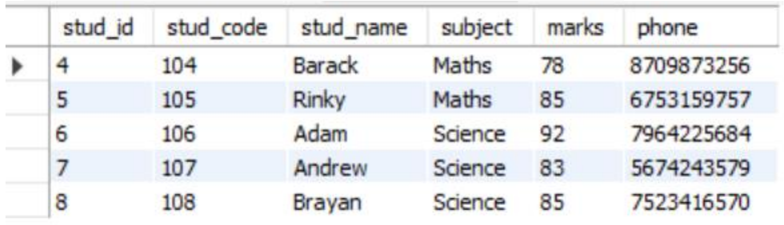


stud_id	stud_code	stud_name	subject	marks	phone
1	101	Mark	English	68	3454569537
2	102	Joseph	Physics	70	9876543659
3	103	John	Maths	70	9765326976
4	104	Barack	Maths	78	8709873256
5	105	Rinky	Maths	85	6753159757
6	106	Adam	Science	92	7964225684
7	107	Andrew	Science	83	5674243579
8	108	Brayan	Science	85	7523416570
9	109	Alexandar	Biology	67	2347346438
NULL	NULL	NULL	NULL	NULL	NULL

2.1.1 PROCEDURE WITHOUT PARAMETER

```
DELIMITER &&
CREATE PROCEDURE get_merit_student ()
BEGIN
    SELECT * FROM student_info WHERE marks > 70;
END &&
DELIMITER ;
```

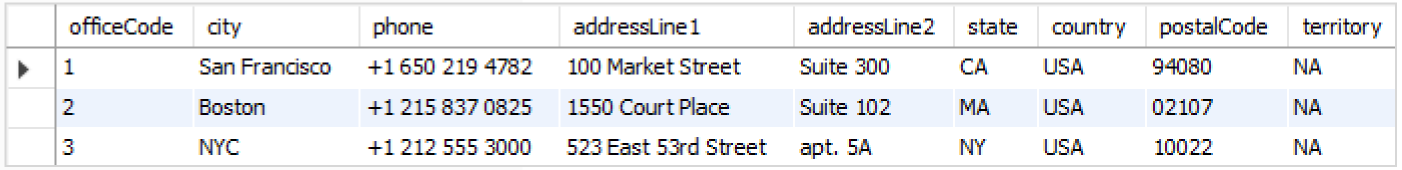
CALL get_merit_student();



stud_id	stud_code	stud_name	subject	marks	phone
4	104	Barack	Maths	78	8709873256
5	105	Rinky	Maths	85	6753159757
6	106	Adam	Science	92	7964225684
7	107	Andrew	Science	83	5674243579
8	108	Brayan	Science	85	7523416570

2.1.2 PROCEDURES WITH IN PARAMETER

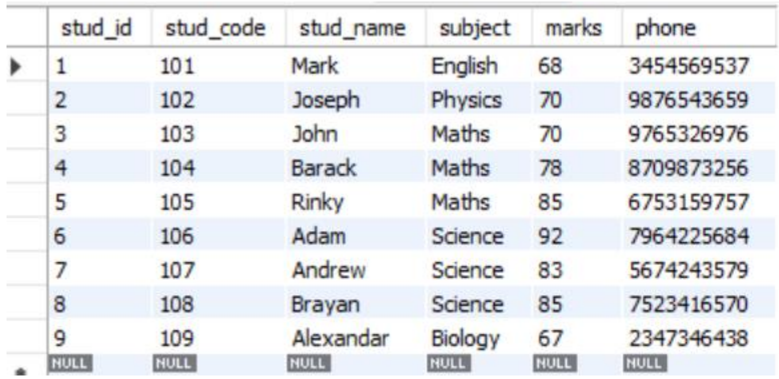
```
DELIMITER //

CREATE PROCEDURE GetOfficeByCountry(
    IN countryName VARCHAR(255)
)
BEGIN
    SELECT *
    FROM offices
    WHERE country = countryName;
END //

DELIMITER ;
```



officeCode	city	phone	addressLine1	addressLine2	state	country	postalCode	territory
1	San Francisco	+1 650 219 4782	100 Market Street	Suite 300	CA	USA	94080	NA
2	Boston	+1 215 837 0825	1550 Court Place	Suite 102	MA	USA	02107	NA
3	NYC	+1 212 555 3000	523 East 53rd Street	apt. 5A	NY	USA	10022	NA

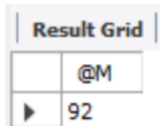
CALL GetOfficeByCountry('USA');

2.1.3 PROCEDURES WITH OUT PARAMETER



stud_id	stud_code	stud_name	subject	marks	phone
1	101	Mark	English	68	3454569537
2	102	Joseph	Physics	70	9876543659
3	103	John	Maths	70	9765326976
4	104	Barack	Maths	78	8709873256
5	105	Rinky	Maths	85	6753159757
6	106	Adam	Science	92	7964225684
7	107	Andrew	Science	83	5674243579
8	108	Brayan	Science	85	7523416570
9	109	Alexandar	Biology	67	2347346438
NULL	NULL	NULL	NULL	NULL	NULL

```
DELIMITER &&
CREATE PROCEDURE display_max_mark (OUT highestmark INT)
BEGIN
    SELECT MAX(marks) INTO highestmark FROM student_info;
END &&
DELIMITER ;
```



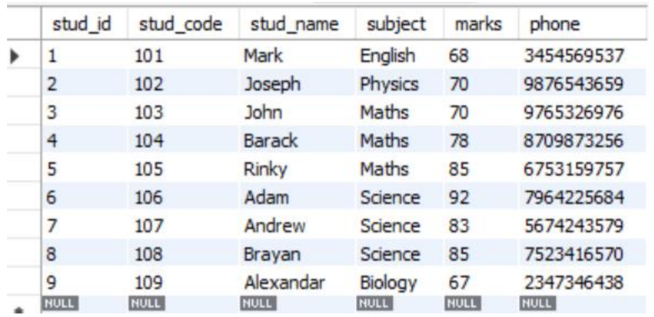
Result Grid

@M
92

```
CALL display_max_mark(@M);
SELECT @M;
```

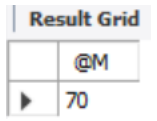
When we call the procedure, the OUT parameter tells the database systems that its value goes out from the procedures. Now, we will pass its value to a session variable **@M** in the CALL statement as given.

2.1.4 PROCEDURES WITH INOUT PARAMETER



stud_id	stud_code	stud_name	subject	marks	phone
1	101	Mark	English	68	3454569537
2	102	Joseph	Physics	70	9876543659
3	103	John	Maths	70	9765326976
4	104	Barack	Maths	78	8709873256
5	105	Rinky	Maths	85	6753159757
6	106	Adam	Science	92	7964225684
7	107	Andrew	Science	83	5674243579
8	108	Brayan	Science	85	7523416570
9	109	Alexandar	Biology	67	2347346438
NULL	NULL	NULL	NULL	NULL	NULL

```
DELIMITER &&
CREATE PROCEDURE display_marks (INOUT var1 INT)
BEGIN
    SELECT marks INTO var1 FROM student_info WHERE stud_id = var1;
END &&
DELIMITER ;
```



```
Result Grid
@M
70
```

```
SET @M = '3';
CALL display_marks(@M);
SELECT @M;
```

In this procedure, we have used the INOUT parameter as 'var1' of integer type. Its body part first fetches the marks from the table with the specified id and then stores it into the same variable var1. The var1 first acts as the IN parameter and then OUT parameter. Therefore, we can call it the INOUT parameter mode.

2.2 VARIABLES

- First, specify the name of the variable after the DECLARE keyword. Ensure the variable name adheres to the database table column naming rules.
- Second, define the data type and length of the variable. Variables can have any relevant data type relevant to its database, such as INT, VARCHAR , and DATETIME.
- Third, assign a default value to the variable using the DEFAULT option. If you declare a variable without specifying a default value, its default value is NULL.

```
DECLARE variable_name datatype(size) [DEFAULT default_value];
```

- *Examples*

```
DECLARE totalSale DEC(10,2) DEFAULT 0.0;
DECLARE totalQty, stockCount INT DEFAULT 0;
DECLARE currency CHAR(3) DEFAULT 'USD';
DECLARE totalSale DEC(10,2);
```

2.2.1 ASSIGNING VARIABLES

- To assign a variable a value, you use the SET statement:

Examples

```
DECLARE total INT
```

- To assign a variable a value, use the SELECT INTO statement to assign the result of a query to a variable:

Examples

```
DECLARE productCount INT DEFAULT 0;


SELECT COUNT(*)

INTO productCount

FROM products;
```

```
DELIMITER $$

CREATE PROCEDURE GetTotalOrder()
BEGIN
	DECLARE totalOrder INT DEFAULT 0;

	SELECT
		COUNT(*)
	INTO totalOrder FROM
		orders;

	SELECT totalOrder;
END$$

DELIMITER ;
```

2.2.2 VARIABLE SCOPES

- A variable has its own scope, which determines its lifetime. If you declare a variable inside a stored procedure, it will be out of scope when the END statement of the stored procedure is reached.
- When you declare a variable inside the BEGIN...END block, it goes out of scope once the END is reached.
- MySQL allows you to declare two or more variables that share the same name in different scopes because a variable is only effective within its scope.
- However, declaring variables with the same name in different scopes is not considered good programming practice.
- ***A variable whose name begins with the @ sign is a session variable, available and accessible until the session ends.***

3

CONDITIONAL STATEMENTS AND LOOPS

3.1.1 CONDITIONAL STATEMENTS - IF

- The IF statement allows you to evaluate one or more conditions and execute the corresponding code block if the condition is true.
- The IF statement has three forms:
  - IF…THEN statement: Evaluate one condition and execute a code block if the condition is true.
  - IF…THEN…ELSE statement: Evaluate one condition and execute a code block if the condition is true; otherwise, execute another code block.
  - IF…THEN…ELSEIF…ELSE statement: Evaluate multiple conditions and execute a code block if a condition is true. If all conditions are false, execute the code block in the ELSE branch.

```
DELIMITER $$

CREATE PROCEDURE GetCustomerLevel(
   IN  pCustomerNumber INT,
   OUT pCustomerLevel  VARCHAR(20))
BEGIN
   DECLARE credit DECIMAL DEFAULT 0;

   SELECT creditLimit
   INTO credit
   FROM customers
   WHERE customerNumber = pCustomerNumber;

   IF credit > 50000 THEN
      SET pCustomerLevel = 'PLATINUM';
   ELSEIF credit <= 50000 AND credit > 10000 THEN
      SET pCustomerLevel = 'GOLD';
   ELSE
      SET pCustomerLevel = 'SILVER';
   END IF;
END $$

DELIMITER ;
```

```
CALL GetCustomerLevel(447, @level);
SELECT @level;
```

```
+--------+
| @level |
+--------+
| GOLD   |
+--------+
1 row in set (0.00 sec)
```

3.1.2 CONDITIONAL STATEMENTS - CASE

- Besides the IF statement, MySQL provides an alternative conditional statement called the CASE statement used in stored procedures. The CASE statements make the code more readable and efficient.
- The CASE statement has two forms:

  - Simple CASE statement   - Searched CASE statement.

```
DELIMITER $$
CREATE PROCEDURE GetCustomerShipping(
  IN pCustomerNumber INT,
  OUT pShipping VARCHAR(50)
)
BEGIN
        DECLARE customerCountry VARCHAR(100);
        SELECT  country INTO customerCountry FROM  customers
        WHERE  customerNumber = pCustomerNumber;
        CASE customerCountry
                WHEN 'USA' THEN
                        SET pShipping = '2-day Shipping';
                WHEN 'Canada' THEN
                        SET pShipping = '3-day Shipping';
                ELSE
                        SET pShipping = '5-day Shipping';
        END CASE;
END$$
DELIMITER ;
```

```
DELIMITER $$
CREATE PROCEDURE GetDeliveryStatus(
  IN pOrderNumber INT,
  OUT pDeliveryStatus VARCHAR(100))
BEGIN
        -- get the waiting day from the orders table
        DECLARE waitingDay INT DEFAULT 0;
        SELECT DATEDIFF(shippedDate, requiredDate) INTO waitingDay
        FROM orders  WHERE   orderNumber = pOrderNumber;
        -- determine delivery status
        CASE
                WHEN waitingDay < 0 THEN
                        SET pDeliveryStatus = 'Early Delivery';
                WHEN waitingDay = 0 THEN
                        SET pDeliveryStatus = 'On Time';
                WHEN waitingDay >= 1 AND waitingDay < 5 THEN
                        SET pDeliveryStatus = 'Late';
                WHEN waitingDay >= 5 THEN
                        SET pDeliveryStatus = 'Very Late';
                ELSE
                        SET pDeliveryStatus = 'No Information';
        END CASE;
END$$
DELIMITER ;
```

3.2.1 LOOPS - LOOP

- The LOOP statement allows you to execute one or more statements repeatedly.
- The LOOP can have optional labels at the beginning and end of the block.
- you can use the ITERATE statement to skip the current iteration and start a new one.

```
[begin_label:] LOOP
    statements;
END LOOP [end_label]
```

```
[label]: LOOP
    ...
    -- terminate the loop
    IF condition THEN
        LEAVE [label];
    END IF;
    ...
END LOOP;
```

```
[label]: LOOP
    ...
    -- terminate the loop
    IF condition THEN
        ITERATE [label];
    END IF;
    ...
END LOOP;
```

3.2.1 LOOPS – LOOP (CONTD)

- Example

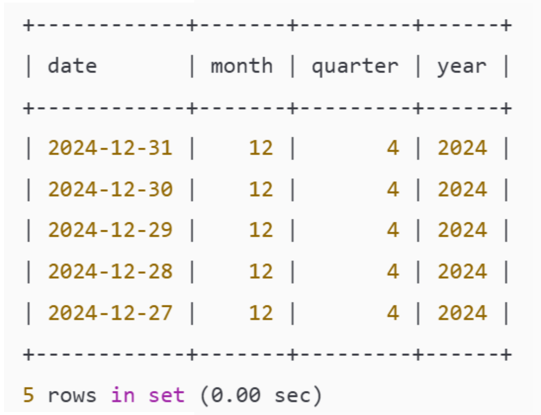
```
CREATE TABLE calendars (
    date DATE PRIMARY KEY,
    month INT NOT NULL,
    quarter INT NOT NULL,
    year INT NOT NULL
);
```

```
CALL fillDates('2024-01-01','2024-12-31');
```

```
SELECT COUNT(*) FROM calendars;
```

will give the out as 365

```
SELECT * FROM calendars ORDER BY date DESC LIMIT 5;
```



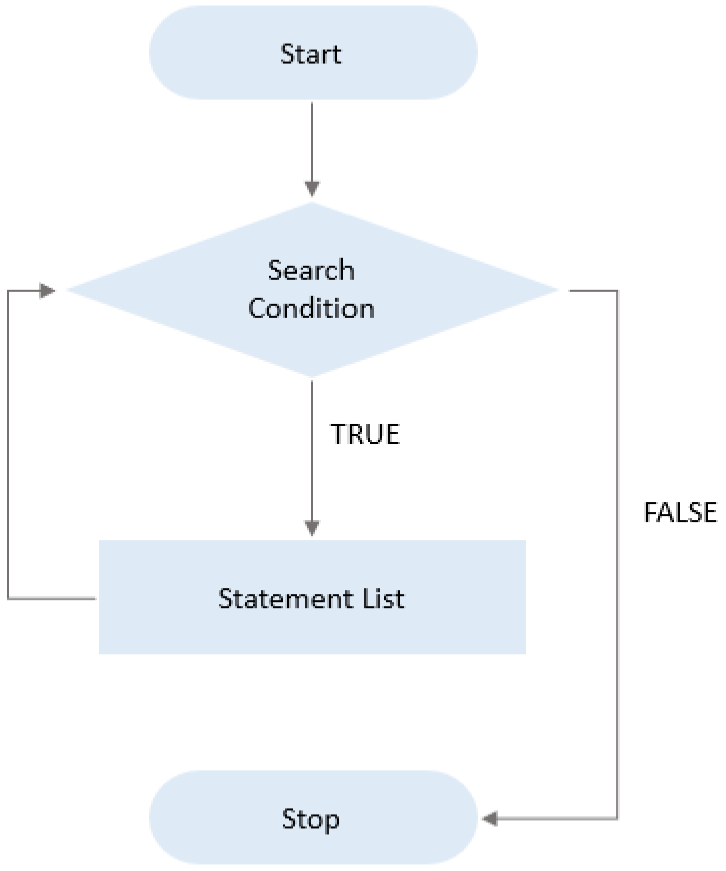
```
+------------+-------+---------+------+
| date       | month | quarter | year |
+------------+-------+---------+------+
| 2024-12-31 |    12 |       4 | 2024 |
| 2024-12-30 |    12 |       4 | 2024 |
| 2024-12-29 |    12 |       4 | 2024 |
| 2024-12-28 |    12 |       4 | 2024 |
| 2024-12-27 |    12 |       4 | 2024 |
+------------+-------+---------+------+
5 rows in set (0.00 sec)
```

```
DELIMITER //

CREATE PROCEDURE fillDates(
        IN startDate DATE,
   IN endDate DATE
)
BEGIN
        DECLARE currentDate DATE DEFAULT startDate;

     insert_date: LOOP
                -- increase date by one day
        SET currentDate = DATE_ADD(currentDate, INTERVAL 1 DAY);

      -- leave the loop if the current date is after the end date
      IF currentDate > endDate THEN
                    LEAVE insert_date;
      END IF;

      -- insert date into the table
      INSERT INTO calendars(date, month, quarter, year)
      VALUES(currentDate, MONTH(currentDate),
        QUARTER(currentDate), YEAR(currentDate));

   END LOOP;
END //

DELIMITER ;
```

3.2.2 LOOPS - WHILE

- The WHILE loop is a loop statement that executes a block of code repeatedly as long as a condition is true.
- Here is the basic syntax of the WHILE statement:

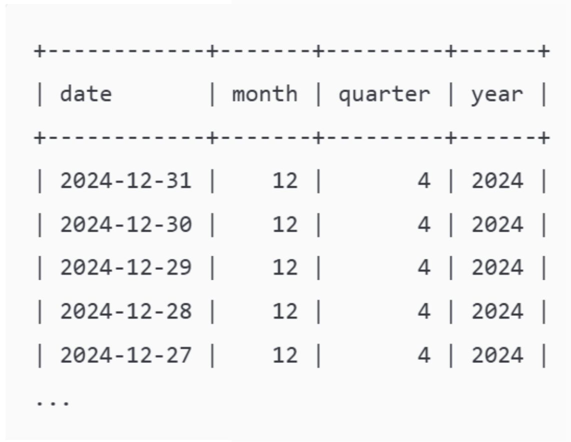
```
[begin_label:] WHILE
search_condition DO
   statement_list
END WHILE [end_label]
```



3.2.2 LOOPS – WHILE (CONTD)

- Example

```
CREATE TABLE calendars(
    date DATE PRIMARY KEY,
    month INT NOT NULL,
    quarter INT NOT NULL,
    year INT NOT NULL
);
```



```
+------------+-------+---------+------+
| date       | month | quarter | year |
+------------+-------+---------+------+
| 2024-12-31 |    12 |       4 | 2024 |
| 2024-12-30 |    12 |       4 | 2024 |
| 2024-12-29 |    12 |       4 | 2024 |
| 2024-12-28 |    12 |       4 | 2024 |
| 2024-12-27 |    12 |       4 | 2024 |
...
```

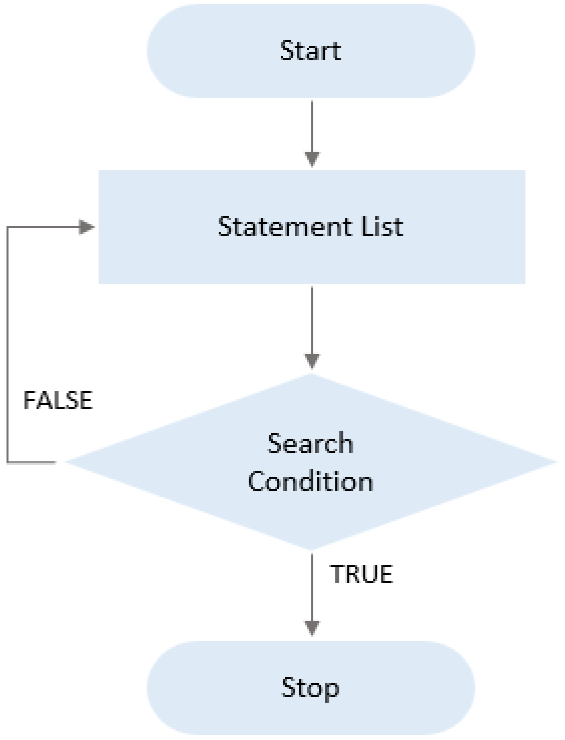
```
CALL loadDates('2024-01-01',365);

SELECT * FROM calendars ORDER BY date DESC ;
```

```
DELIMITER $$

CREATE PROCEDURE loadDates(
    startDate DATE,
    day INT
)
BEGIN

    DECLARE counter INT DEFAULT 0;
    DECLARE currentDate DATE DEFAULT startDate;

    WHILE counter <= day DO
        CALL InsertCalendar(currentDate);
        SET counter = counter + 1;
        SET currentDate = DATE_ADD(currentDate ,INTERVAL 1 day);
    END WHILE;

END$$

DELIMITER ;
```

3.2.3 LOOPS - REPEAT

- The REPEAT statement creates a loop that repeatedly executes a block of statements until a condition is true. Here is the basic syntax of the WHILE statement:

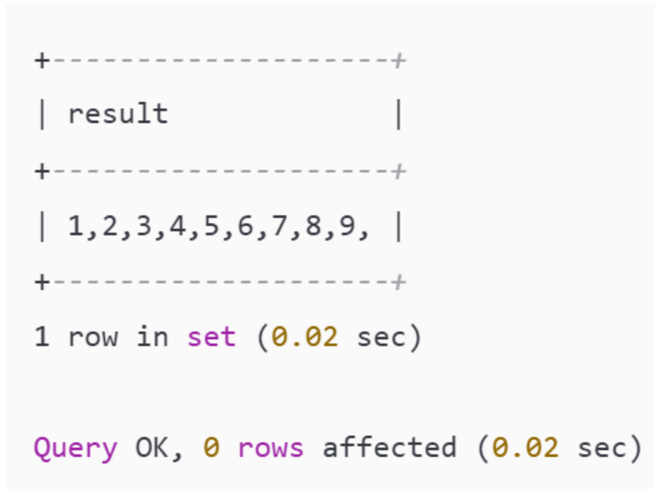
```
[begin_label:] REPEAT
    statement;
UNTIL condition
END REPEAT [end_label]
```



3.2.3 LOOPS – REPEAT (CONTD)

- Example

```
CALL RepeatDemo();
```



```
+--------------------+
| result             |
+--------------------+
| 1,2,3,4,5,6,7,8,9, |
+--------------------+
1 row in set (0.02 sec)

Query OK, 0 rows affected (0.02 sec)
```

```
DELIMITER $$

CREATE PROCEDURE RepeatDemo()
BEGIN
    DECLARE counter INT DEFAULT 1;
    DECLARE result VARCHAR(100) DEFAULT '';

    REPEAT
        SET result = CONCAT(result,counter,',');
        SET counter = counter + 1;
    UNTIL counter >= 10
    END REPEAT;

    -- display result
    SELECT result;
END$$

DELIMITER ;
```

3.3 LEAVE STATEMENT

- The LEAVE statement exits the flow control that has a given label.
- The following shows the basic syntax of the LEAVE statement:

```
LEAVE label;
```

- In this syntax, you specify the label of the block that you want to exit after the LEAVE keyword.

```
CREATE PROCEDURE sp_name()
sp: BEGIN
   IF condition THEN
      LEAVE sp;
   END IF;
   -- other statement
END$$
```

**Terminate the procedure*

```
[label]: LOOP
   IF condition THEN
      LEAVE [label];
   END IF;
   -- statements
END LOOP [label];
```

```
[label:] REPEAT
   IF condition THEN
      LEAVE [label];
   END IF;
   -- statements
UNTIL
search_condition
END REPEAT [label];
```

```
[label:] WHILE
search_condition DO
   IF condition THEN
      LEAVE [label];
   END IF;
   -- statements
END WHILE [label];
```

**Causes the current loop specified by the label to be terminated.*

4

ERROR HANDLING

4.1 SHOW WARNINGS

- When you execute a statement and encounter an error or warning, you can use the **SHOW WARNINGS** statement to display detailed information.
- To show the total number of errors,

```
SHOW COUNT(*) WARNINGS;  OR SELECT @@warning_count;
```

- ***Example:*** the following statement attempts to drop the table abc that doesn't exist:

```
DROP TABLE IF EXISTS abc;
```

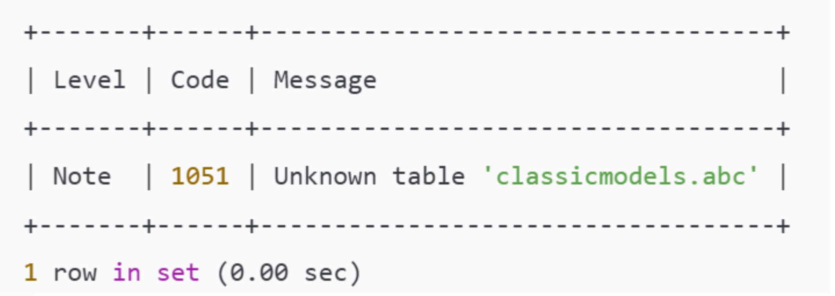
- MySQL returns a message indicating that it encountered a warning:

```
Query OK, 0 rows affected, 1 warning (0.01 sec)
```

- To display the warning, you use the SHOW WARNINGS statement:

```
SHOW WARNINGS;
```

- The output has three columns:
  Level: It can be a Note or Error.
  Code: This is an integer that represents a MySQL error code.
  Message: This stores the detailed warning or error message.



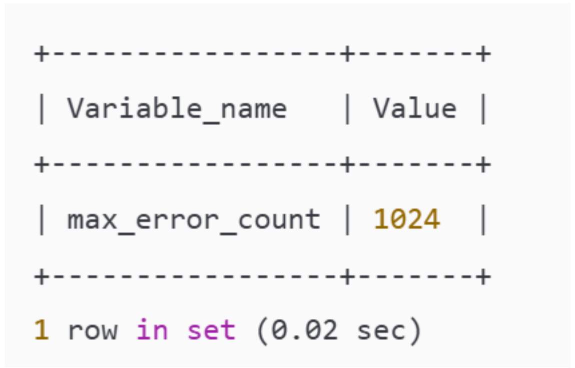
```
+-------+------+---------------------------------+
| Level | Code | Message                         |
+-------+------+---------------------------------+
| Note  | 1051 | Unknown table 'classicmodels.abc' |
+-------+------+---------------------------------+
1 row in set (0.00 sec)
```

4.1 SHOW WARNINGS (CONTD)

- MySQL uses the max_error_count system variable to control the maximum number of warnings, errors, and notes that the server can store.

- To view the value of the max_error_count system variable, you use the SHOW VARIABLES statement:

```
SHOW VARIABLES LIKE 'max_error_count';
```



```
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_error_count | 1024  |
+-----------------+-------+
1 row in set (0.02 sec)
```

- To change the value of the max_error_count variable, you use the SET statement. For example, this statement sets the max_error_count to 2048;

```
SET max_error_count=2048;
```

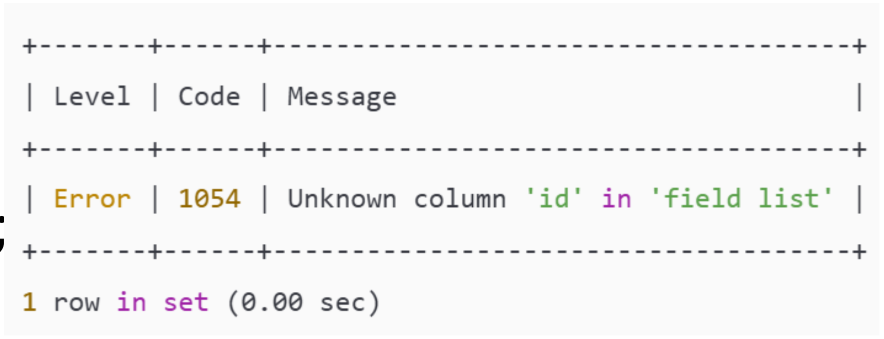
- Setting the value of the max_error_count variable to zero will disable the message storage. However, the warning_count still shows the number of errors and warnings that occurred, but the server does not store these messages.

4.2 SHOW ERRORS

- The **SHOW ERRORS** statement is used to display error information about the most recent execution of a statement or a stored procedure.
- To show the total number of errors,

```
SHOW COUNT(*) ERRORS;  OR SELECT @@error_count;
```

- The SHOW ERRORS statement works like the SHOW WARNINGS statement but it shows only errors, not warnings, and notes.



```
+-------+------+------------------------------------+
| Level | Code | Message                            |
+-------+------+------------------------------------+
| Error | 1054 | Unknown column 'id' in 'field list' |
+-------+------+------------------------------------+
1 row in set (0.00 sec)
```

- ***Example:*** Assume products table has no id column

```
SELECT id FROM products;
```

It give the below message:

```
ERROR 1054 (42S22): Unknown column 'id' in 'field list'
```

- In the message:
  - ERROR indicates that the message is an error message.
  - 1054 is an integer that represents the MySQL error code.
  - 42S22 is a five-character alphanumeric code that represents the condition of the most recently executed SQL statement.
  - "Unknown column 'id' in 'field list'" represents the detailed error message.

4.3 DECLARE ... HANDLER STATEMENT

- In MySQL, conditions refer to errors, warnings, or exceptional cases that require proper handling.

- When a condition arises during the execution of a stored procedure, you should handle it properly, such as exiting or continuing the current code block.

- To handle a condition, you declare a handler using the DECLARE ... HANDLER statement.

```
DECLARE { EXIT | CONTINUE } HANDLER
    FOR condition_value [, condition_value] ...
    statement
```

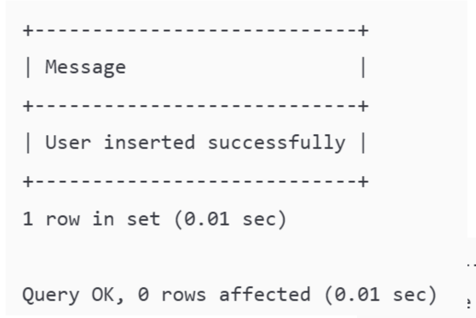
  - **DECLARE { EXIT | CONTINUE } HANDLER:** This declares a handler, instructing whether it should exit or continue the enclosing stored procedure when a specified condition occurs.
  - **EXIT:** The stored procedure will terminate.
  - **CONTINUE**: The stored procedure will continue execution.
  - **FOR condition_value [, condition_value] ...:** This specifies the conditions that activate the handler, and you can specify multiple conditions by separating them with commas.
  - **statement:** This statement or block of statements executes when the stored procedure encounters one of the specified conditions.

4.3 DECLARE … HANDLER STATEMENT (CONTD)

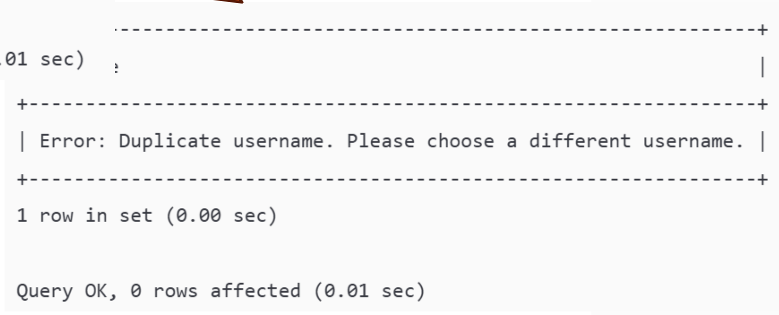
```
CREATE TABLE users (
        user_id INT AUTO_INCREMENT PRIMARY KEY,
        username VARCHAR(50) UNIQUE NOT NULL,
        email VARCHAR(50) NOT NULL
);
```

```
DELIMITER //
CREATE PROCEDURE insert_user(
        IN p_username VARCHAR(50),
        IN p_email VARCHAR(50)
)
BEGIN
        -- SQLSTATE for unique constraint violation
        DECLARE EXIT HANDLER FOR SQLSTATE '23000'
        BEGIN
                -- Handler actions when a duplicate username is detected
                SELECT 'Error: Duplicate username. Please choose a different
            username.' AS Message;
        END;

        -- Attempt to insert the user into the table
        INSERT INTO users (username, email) VALUES (p_username, p_email);

        -- If the insertion was successful, display a success message
        SELECT 'User inserted successfully' AS Message;
END //
DELIMITER ;
```

```
CALL insert_user('jane','jane@example.com');
```



```
+----------------------------+
| Message                    |
+----------------------------+
| User inserted successfully |
+----------------------------+
1 row in set (0.01 sec)

Query OK, 0 rows affected (0.01 sec)
```



```
-----------------------------------------------------------+
                                                           |
+----------------------------------------------------------------+
| Error: Duplicate username. Please choose a different username. |
+----------------------------------------------------------------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.01 sec)
```

4.4 DECLARE … CONDITION STATEMENT

- Use MySQL DECLARE … CONDITION to associate a name with a condition specified by a MySQL error code or SQLSTATE value to make the stored procedure code more readable and expressive.
- In other words, the named conditions make your store procedures code clearer and easier to maintain.
- When a condition arises during the execution of a stored procedure, you should handle it properly, such as exiting or continuing the current code block.

```
DECLARE condition_name CONDITION FOR condition_value

condition_value: {
    mysql_error_code
  | SQLSTATE [VALUE] sqlstate_value
}
```

- First, specify the condition's name after the DECLARE keyword (codition_name).
- Second, provide the condition value (condition_value) after the FOR keyword. The condition value can be a MySQL error code or a SQLSTATE value.

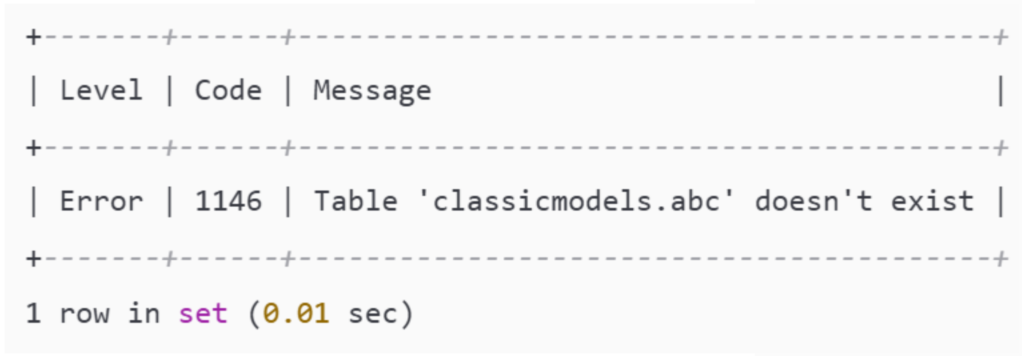
4.4 DECLARE … CONDITION STATEMENT (CONTD)

- Imagine if a table call ‘employees’ is there in the database below call will successfully return its rows

```
CALL GetData('employees’);
```

- Imagine if a table call ‘abc’ is not there in the database it will give the below error as a result.

```
CALL GetData(‘abc');
```



```
+-------+------+------------------------------------+
| Level | Code | Message                            |
+-------+------+------------------------------------+
| Error | 1146 | Table 'classicmodels.abc' doesn't exist |
+-------+------+------------------------------------+
1 row in set (0.01 sec)
```

```
DELIMITER $$

CREATE PROCEDURE GetData(
        IN tbl_name VARCHAR(255)
)
BEGIN
   DECLARE unknown_table CONDITION FOR 1051;

   DECLARE EXIT HANDLER FOR unknown_table
        BEGIN
                SHOW ERRORS;
   END;

   SET @sql_query = CONCAT('SELECT * FROM ', tbl_name);

   PREPARE stmt FROM @sql_query;
   EXECUTE stmt;
   DEALLOCATE PREPARE stmt;

END$$

DELIMITER ;
```

4.5 SIGNAL STATEMENT

- The MySQL SIGNAL statement is used to convey error information in stored procedures. It ensures that exceptions are properly handled, preventing sudden procedure termination.

Exception Handling with DECLARE … HANDLER

- You can use the DECLARE … HANDLER statement to effectively manage exceptions in MySQL. It allows you to specify how different types of exceptions should be handled within a stored procedure. By using this statement in conjunction with SIGNAL, you can enable precise control over error handling.

Customizing Error Messages

- The SIGNAL statement allows for the customization of error messages using the SET MESSAGE_TEXT command. This is helpful for modifying error messages to provide more meaningful feedback to handlers, applications, or clients.

```
SIGNAL condition_value [SET signal_information_item]
```

4.5 SIGNAL STATEMENT (CONTD)

```
-- Create a sample employee table
CREATE TABLE IF NOT EXISTS employees (
   id INT PRIMARY KEY,
   name VARCHAR(100),
   salary DECIMAL(10,2)
);

-- Insert some sample data
INSERT INTO employees (id, name, salary)
VALUES
   (1, 'John Doe', 50000),
   (2, 'Jane Smith', 75000),
   (3, 'Bob Johnson', 90000);
```

```
DELIMITER //
CREATE PROCEDURE update_salary(
         IN p_employee_id INT,
   IN p_salary DECIMAL)
BEGIN
         DECLARE employee_count INT;

   -- check if employee exists
   SELECT COUNT(*) INTO employee_count
   FROM employees
   WHERE id = p_employee_id;

   IF employee_count = 0 THEN
                  SIGNAL SQLSTATE '45000'
                  SET MESSAGE_TEXT = 'Employee not found';
   END IF;

   -- validate salary
   IF p_salary < 0 THEN
                  SIGNAL SQLSTATE '45000'
                  SET MESSAGE_TEXT = 'Salary cannot be negative';
   END IF;

   -- if every is fine, update the salary
   UPDATE employees
   SET salary = p_salary
   WHERE id = p_employee_id;
END //
DELIMITER ;
```

```
CALL update_salary(1, 7000);
```

```
ERROR 1644 (45000): Employee not found
```

```
CALL update_salary(1,-7000);
```

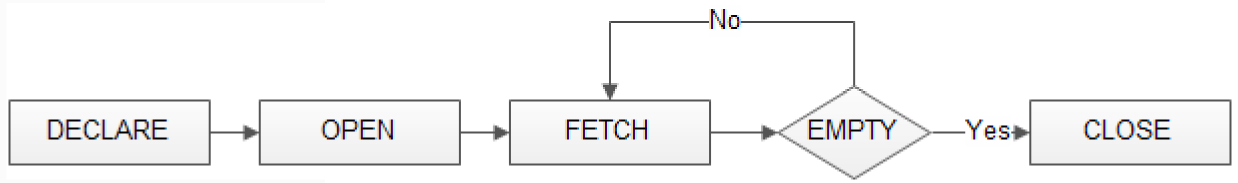
```
ERROR 1644 (45000): Salary cannot be negative
```

The SQLSTATE value should not start with '00' because it doesn't indicate an error. To signal a generic SQLSTATE value, you use '45000', which indicates an "unhandled user-defined exception"

5

CURSORS AND PREPARED STATEMENTS

5.1 WHAT?



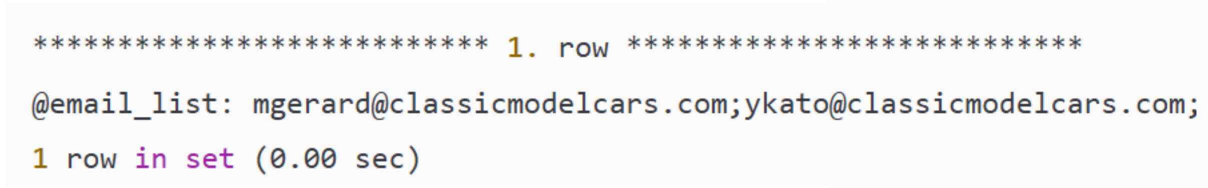
- Cursor is a database object used for iterating the result of a SELECT statement.
- **Typically, you use cursors within stored procedures, triggers, and functions where you need to process individual rows returned by a query one at a time.**
- It is a good practice to always close a cursor when it is no longer used.

- When working with MySQL cursor, you must also declare a NOT FOUND handler to manage the situation when the cursor cannot find any row.
- Each time you call the FETCH statement, the cursor attempts to read the next row in the result set. When the cursor reaches the end of the result set, it will not be able to retrieve the data, and a condition is raised. The handler is used to handle this condition.

```
-- declare a cursor
DECLARE cursor_name CURSOR FOR
SELECT column1, column2
FROM your_table
WHERE your_condition;

-- open the cursor
OPEN cursor_name;

FETCH cursor_name INTO variable1,
variable2;
-- process the data


-- close the cursor
CLOSE cursor_name;
```

5.1 EXAMPLE

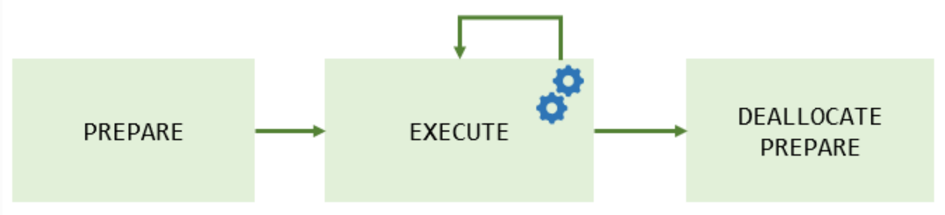
```
CALL create_email_list(@email_list);
SELECT @email_list;
```



```
*************************** 1. row ***************************
@email_list: mgerard@classicmodelcars.com;ykato@classicmodelcars.com;
1 row in set (0.00 sec)
```

```
DELIMITER $$
CREATE PROCEDURE create_email_list (
	INOUT email_list TEXT
)
BEGIN
	DECLARE done BOOL DEFAULT false;
	DECLARE email_address VARCHAR(100) DEFAULT "";

	-- declare cursor for employee email
	DECLARE cur CURSOR FOR SELECT email FROM employees;

	-- declare NOT FOUND handler
	DECLARE CONTINUE HANDLER FOR NOT FOUND
	SET done = true;

  -- open the cursor
	OPEN cur;

  SET email_list = '';

  process_email: LOOP
    FETCH cur INTO email_address;

		IF done = true THEN
			LEAVE process_email;
		END IF;

    -- concatenate the email into the emailList
		SET email_list = CONCAT(email_address,";",email_list);
	END LOOP;

  -- close the cursor
	CLOSE cur;
END$$
DELIMITER ;
```

5.2 PREPARED STATEMENTS

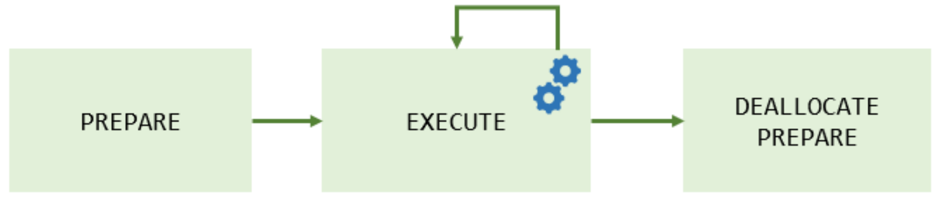


- Prepared statements are a feature that helps you enhance the security and performance of database queries.
- Prepared statements allow you to write SQL queries with placeholders for parameters, and then bind values to those parameters at runtime. They can help prevent SQL injection attacks and optimize query execution.

```
PREPARE stmt_name FROM preparable_stmt;
```

- First, specify the name of the prepared statement (stmt_name) after the PREPARE keyword.
- Second, provide the SQL statement with placeholders (?) (preparable_stmt) after the FROM keyword. The preparable_stmt represents a single SQL statement, not multiple statements.
- The preparable_stmt is sent to the MySQL server with placeholders (?) for parameters. Upon receiving the statement, the MySQL server parses, optimizes, and precompiles the query, and then creates the prepared statement.
- After creating a prepared statement, you need to initialize a set of user variables to supply values for the parameter placeholders (?) specified in the prepared statement

5.2 PREPARED STATEMENTS



```
CREATE PROCEDURE insert_user()
BEGIN
  PREPARE insert_user FROM 'INSERT INTO users (username, email) VALUES (?, ?)';
  -- Store the values in session variables
  SET @username = 'jane_doe';
  SET @email = 'jane@example.com';
  EXECUTE insert_user USING @username, @email;
  DEALLOCATE PREPARE insert_user;
END;
```

5.3 PREPARED STATEMENTS – DYNAMIC SQL

- Dynamic SQL, such as changing the table name based on a parameter, cannot be executed as a regular SQL statement. MySQL requires **prepared statements** to handle this kind of dynamic behavior.

```
DECLARE table_name VARCHAR(100) DEFAULT 'employees';

SELECT * FROM table_name WHERE employee_id = 101;  -- This won't work.
```

This will result in an error because table_name is a variable, and SQL doesn't allow variable table names in the query directly.

- It doesn't directly allow the execution of dynamic queries with table names or column names as variables, you **must prepare the query** first, as you did in the original example.
- Correct way is:

```
CREATE PROCEDURE GetEmployeeData(IN table_name VARCHAR(100), IN emp_id INT)
BEGIN
    SET @sql = CONCAT('SELECT * FROM ', table_name, ' WHERE employee_id = ?');
    PREPARE stmt FROM @sql;  -- Prepares the SQL query
    EXECUTE stmt USING emp_id;  -- Executes the prepared statement with parameter
    DEALLOCATE PREPARE stmt;  -- Cleans up the prepared statement
END;
```

5.3 PREPARED STATEMENTS – DYNAMIC SQL

```
DELIMITER //

CREATE PROCEDURE GetEmployeeData(IN table_name VARCHAR(100), IN emp_id INT)
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE emp_name VARCHAR(255);
    DECLARE modified_name VARCHAR(255);

    -- Step 1: Create a temporary table to store results
    SET @sql = CONCAT('CREATE TEMPORARY TABLE temp_result AS SELECT name FROM ', table_name, ' WHERE employee_id = ', emp_id);
    PREPARE stmt FROM @sql;
    EXECUTE stmt;
    DEALLOCATE PREPARE stmt;

    -- Step 2: Declare cursor to iterate over results
    DECLARE cur CURSOR FOR SELECT name FROM temp_result;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    OPEN cur;

    read_loop: LOOP
        FETCH cur INTO emp_name;
        IF done THEN
            LEAVE read_loop;
        END IF;

        -- Step 4: Manipulate the name (Example: Convert to UPPERCASE and add prefix)
        SET modified_name = CONCAT('Employee: ', UPPER(emp_name));

        -- Step 5: Print the manipulated name
        SELECT modified_name AS "Modified Name";
    END LOOP;

    -- Step 6: Close cursor and clean up
    CLOSE cur;
    DROP TEMPORARY TABLE IF EXISTS temp_result;
END //

DELIMITER ;
```

6

NESTED STORED PROCEDURES

6.1 WHAT?

- Nested stored procedures help to break up large amounts of SQL statements into smaller reusable pieces.
- Moving reusable logic pieces into smaller composable pieces can make your procedures more readable and easy to debug.
- Whenever you write a statement that calls a procedure X in the body of another stored procedure Y, you're nesting X in Y. The syntax looks like this:

```
CREATE PROCEDURE parent_procedure
BEGIN
      SELECT column FROM table;
      // other statements here
      //nested stored procedure
      CALL nested_procedure;
END
```

6.2 EXAMPLE

```
DELIMITER //
CREATE  PROCEDURE  film_in_stock(IN  p_film_id  INT,  IN
p_store_id INT, OUT p_film_count INT)
BEGIN
    SELECT inventory_id
    FROM inventory
    WHERE film_id = p_film_id
    AND store_id = p_store_id
    AND inventory_in_stock(inventory_id);

    SELECT COUNT(*)
    FROM inventory
    WHERE film_id = p_film_id
    AND store_id = p_store_id
    AND inventory_in_stock(inventory_id)
    INTO p_film_count;
END //

DELIMITER;
```

```
DELIMITER //
CREATE PROCEDURE film_in_stock(IN p_film_id INT, IN
p_store_id INT, OUT p_film_count INT)
BEGIN
    SELECT inventory_id
    FROM inventory
    WHERE film_id = p_film_id
    AND store_id = p_store_id
    AND inventory_in_stock(inventory_id);

    CALL count_inventory(p_film_id, p_store_id,p_film_count);
END //
DELIMITER;
```

inventory_in_stock(inventory_id).
Ignore this because it is a function.

7

TRIGGERS

7.1 INTRODUCTION

- **A trigger is a database object that is associated with a table. It will be activated when a defined action is executed for the table.**
- In a database, a trigger is a stored program invoked automatically in response to an event such as insert, update, or delete that occurs in the associated table.

  For example, you can define a trigger that is invoked automatically before a new row is inserted into a table.
- The SQL standard defines two types of triggers:
  - **A row-level trigger** is activated for each row that is inserted, updated, or deleted.

    For example, if a table has 100 rows inserted, updated, or deleted, the trigger is automatically invoked 100 times for the 100 rows affected.
  - **A statement-level trigger** is executed once for each transaction regardless of how many rows are inserted, updated, or deleted.
- MySQL supports only row-level triggers. It doesn't support statement-level triggers.

7.2 ADVANTAGES OF TRIGGERS

- Triggers provide another way to check the integrity of data.
- Triggers handle errors from the database layer.
- Triggers give an alternative way to run scheduled tasks. By using triggers, you don't have to wait for the scheduled events to run because the triggers are invoked automatically before or after a change is made to the data in a table.
- Triggers can be useful for auditing the data changes in tables.

7.3 DISADVANTAGES OF TRIGGERS

- Triggers can only provide extended validations, not all validations. For simple validations, you can use the NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints.
- Triggers can be difficult to troubleshoot because they execute automatically in the database, which may not be visible to the client applications.
- Triggers may increase the overhead.

7.4 CREATE TRIGGERS

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON table_name
FOR EACH ROW
BEGIN
     - Trigger body (SQL statements)
END;
```

```
DELIMITER $$
CREATE TRIGGER trigger_name
BEFORE INSERT
ON table_name
FOR EACH ROW BEGIN
-- statements
END$$
DELIMITER ;
```

- **trigger_name**: Name of the trigger.
- **BEFORE or AFTER**: Specifies when the trigger should be executed.
- **INSERT, UPDATE, or DELETE**: Specifies the type of operation that activates the trigger.
- **table_name**: Name of the table on which the trigger is defined.
- **FOR EACH ROW**: Indicates that the trigger should be executed once for each row affected by the triggering event.
- **BEGIN and END**: Delimit the trigger body, where you define the SQL statements to be executed.

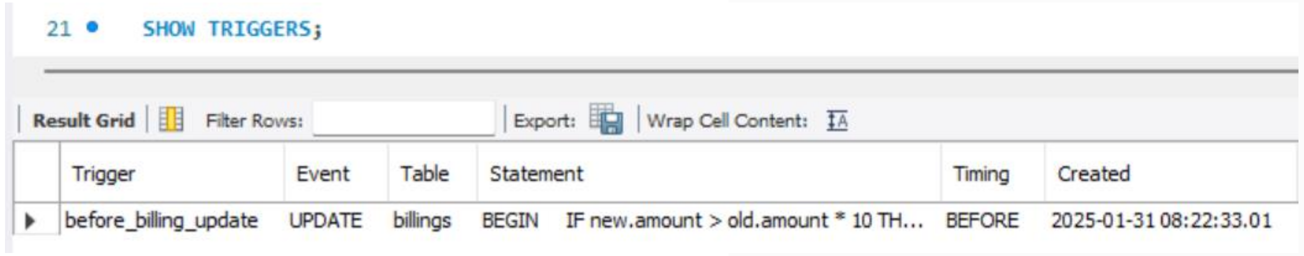
7.5 DROP TRIGGERS

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

- First, specify the name of the trigger that you want to drop after the DROP TRIGGER keywords.
- Second, specify the name of the schema to which the trigger belongs. If you skip the schema name, the statement will drop the trigger in the current database.
- Third, use IF EXISTS option to conditionally drop the trigger if the trigger exists. The IF EXISTS clause is optional.

```
SHOW TRIGGERS;
```

- Show the Triggers



```
SHOW TRIGGERS;
```

Trigger	Event	Table	Statement	Timing	Created
before_billing_update	UPDATE	billings	BEGIN IF new.amount > old.amount * 10 TH...	BEFORE	2025-01-31 08:22:33.01

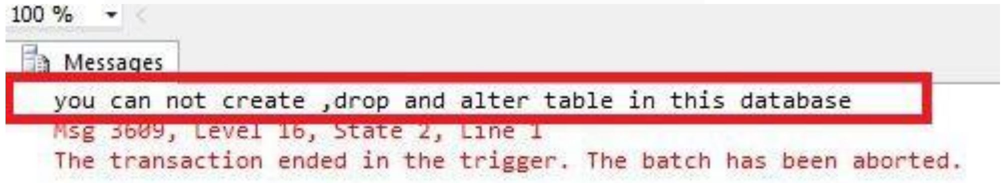
SHOW TRIGGERS
FROM <<SCHEMA_NAME>>;

7.6 TRIGGER TYPES

DDL Triggers

- The Data Definition Language (DDL) command events such as Create_table, Create_view, drop_table, Drop_view, and Alter_table cause the DDL triggers to be activated.
- They allow us to track changes in the structure of the database.
- The trigger will prevent any table creation, alteration, or deletion in the database.

```
CREATE TRIGGER prevent_table_creation
ON DATABASE
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
BEGIN
   PRINT 'you can not create, drop and alter table in this database';
   ROLLBACK;
END;
```



```
you can not create ,drop and alter table in this database
Msg 3609, Level 16, State 2, Line 1
The transaction ended in the trigger. The batch has been aborted.
```

**SQL Server Syntax*
Mysql does not support this still. An alternative solution might be to create a MySQL EVENT that runs at at a specified interval and checks whether there are any tables that don't have corresponding views, and if so, create them. (Make sure to enable the event scheduler.)

7.6 TRIGGER TYPES (CONTD)

Logon Triggers

- These triggers are fired in response to logon events. Logon triggers are useful for monitoring user sessions or restricting user access to the database.
- As a result, the PRINT statement messages and any errors generated by the trigger will all be visible in the Server error log.
- Authentication errors prevent logon triggers from being used.
- These triggers can be used to track login activity or set a limit on the number of sessions that a given login can have in order to audit and manage server sessions.

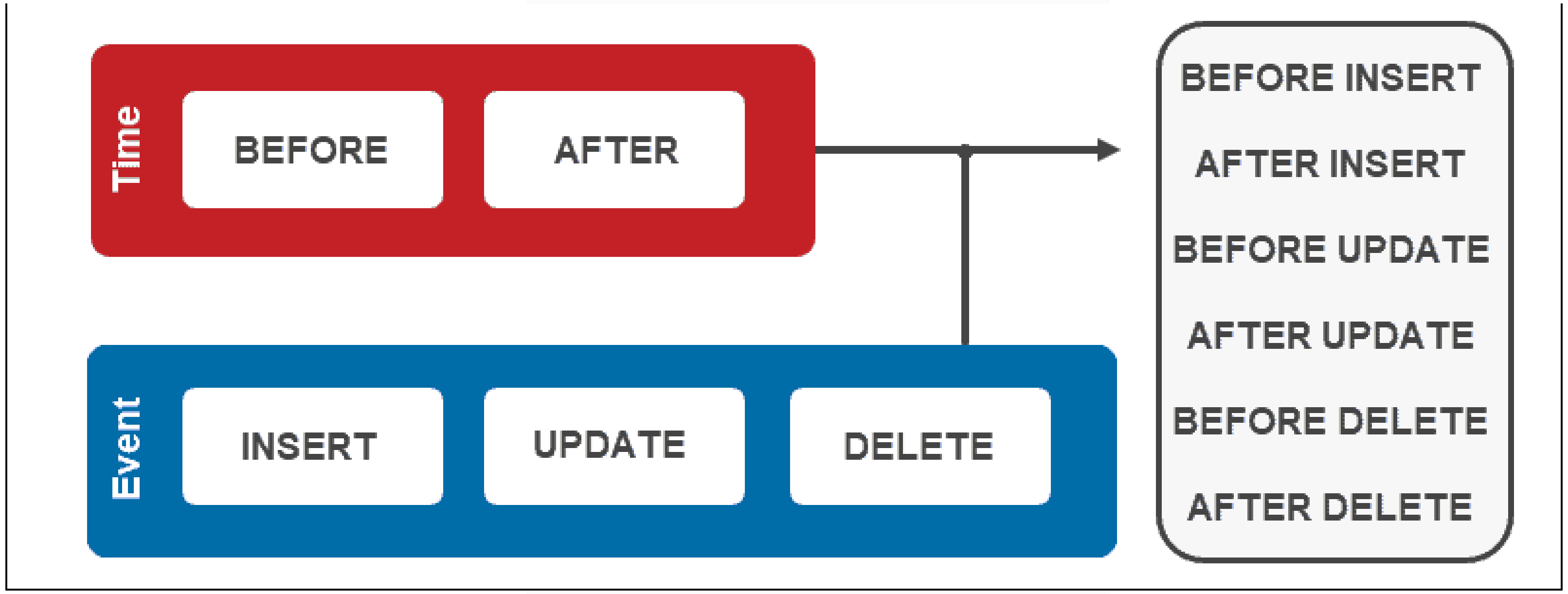
```
CREATE TRIGGER track_logon
ON LOGON
AS
BEGIN
   PRINT 'A new user has logged in.';
END;
```

**SQL Server Syntax. Mysql does not support this*

7.6 TRIGGER TYPES (CONTD)

DML Triggers

- The Data manipulation Language (DML) command events that begin with Insert, Update, and Delete set off the DML triggers.
- DML triggers are used for data validation, ensuring that modifications to a table are done under controlled conditions.



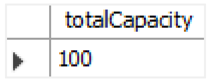
7.6.1 TYPES: BEFORE INSERT TRIGGERS

```
1)
DROP TABLE IF EXISTS WorkCenters;

CREATE TABLE WorkCenters (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    capacity INT NOT NULL
);
```

```
2)
DROP TABLE IF EXISTS WorkCenterStats;

CREATE TABLE WorkCenterStats(
    totalCapacity INT NOT NULL
);
```

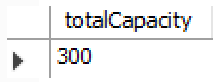
```
3)
DELIMITER $$
CREATE TRIGGER before_workcenters_insert
BEFORE INSERT
ON WorkCenters
FOR EACH ROW
BEGIN
    DECLARE rowcount INT;
    SELECT COUNT(*) INTO rowcount FROM WorkCenterStats;
    IF rowcount > 0 THEN
        UPDATE WorkCenterStats SET totalCapacity =
        totalCapacity + new.capacity;
    ELSE
        INSERT INTO WorkCenterStats(totalCapacity)
        VALUES(new.capacity);
    END IF;
END $$
DELIMITER ;
```

```
4) INSERT INTO WorkCenters(name, capacity) VALUES('Mold
Machine',100);
SELECT * FROM WorkCenterStats;
```



totalCapacity
100

```
5) INSERT INTO WorkCenters(name, capacity)
VALUES('Packing',200);

SELECT * FROM WorkCenterStats;
```



totalCapacity
300

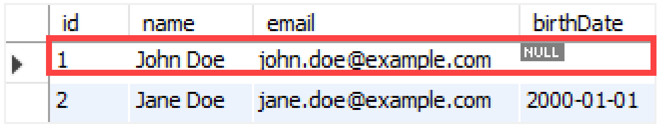
7.6.2 TYPES: AFTER INSERT TRIGGERS

```
1)
DROP TABLE IF EXISTS members;

CREATE TABLE members (
    id INT AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(255),
    birthDate DATE,
    PRIMARY KEY (id)
```

```
2)
DROP TABLE IF EXISTS reminders;

CREATE TABLE reminders (
    id INT AUTO_INCREMENT,
    memberId INT,
    message VARCHAR(255) NOT NULL,
    PRIMARY KEY (id,memberId)
);
```

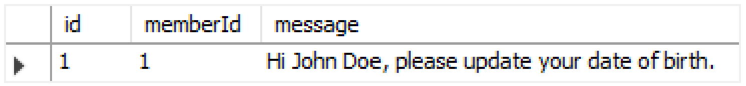
```
3)
DELIMITER $$

CREATE TRIGGER after_members_insert
AFTER INSERT
ON members FOR EACH ROW
BEGIN
    IF NEW.birthDate IS NULL THEN
        INSERT INTO reminders(memberId, message)
        VALUES(new.id,CONCAT('Hi ', NEW.name, ', please
update your date of birth.'));
    END IF;
END$$

DELIMITER ;
```

```
INSERT INTO members(name, email, birthDate)
VALUES
    ('John Doe', 'john.doe@example.com', NULL),
    ('Jane Doe', 'jane.doe@example.com','2000-01-01');
```



id	name	email	birthDate
1	John Doe	john.doe@example.com	NULL
2	Jane Doe	jane.doe@example.com	2000-01-01

```
SELECT * FROM reminders;
```



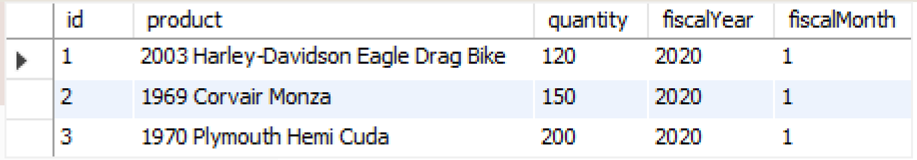
id	memberId	message
1	1	Hi John Doe, please update your date of birth.

7.6.3 Types: Before Update Triggers

```
1)
DROP TABLE IF EXISTS sales;

CREATE TABLE sales (
    id INT AUTO_INCREMENT,
    product VARCHAR(100) NOT NULL,
    quantity INT NOT NULL DEFAULT 0,
    fiscalYear SMALLINT NOT NULL,
    fiscalMonth TINYINT NOT NULL,
    CHECK(fiscalMonth >= 1 AND fiscalMonth <= 12),
    CHECK(fiscalYear BETWEEN 2000 and 2050),
    CHECK (quantity >=0),
    UNIQUE(product, fiscalYear, fiscalMonth),
    PRIMARY KEY(id)
);
```

```
2)
INSERT INTO sales(product, quantity,
fiscalYear,fiscalMonth)
VALUES
    ('2003 Harley-Davidson Eagle Drag Bike',120,2020,1)
    ('1969 Corvair Monza', 150,2020,1),
    ('1970 Plymouth Hemi Cuda', 200,2020,1);
```

```
3)
DELIMITER $$

CREATE TRIGGER before_sales_update
BEFORE UPDATE
ON sales FOR EACH ROW
BEGIN
    DECLARE errorMessage VARCHAR(255);
    SET errorMessage = CONCAT('The new quantity ',
                        NEW.quantity,
                        ' cannot be 3 times greater than
the current quantity ',
                        OLD.quantity);

    IF new.quantity > old.quantity * 3 THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = errorMessage;
    END IF;
END $$

DELIMITER ;
```



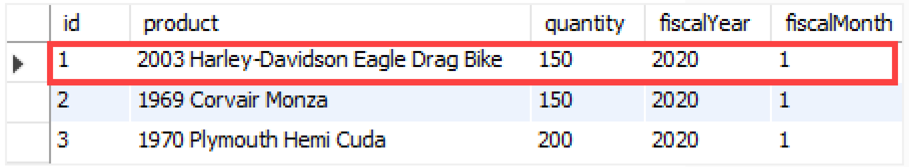
id	product	quantity	fiscalYear	fiscalMonth
1	2003 Harley-Davidson Eagle Drag Bike	120	2020	1
2	1969 Corvair Monza	150	2020	1
3	1970 Plymouth Hemi Cuda	200	2020	1

7.6.3 TYPES: BEFORE UPDATE TRIGGERS (CONTD)

```
4)
UPDATE sales
SET quantity = 150
WHERE id = 1;
```

```
Success!!
```

```
5) SELECT * FROM sales;
```



id	product	quantity	fiscalYear	fiscalMonth
1	2003 Harley-Davidson Eagle Drag Bike	150	2020	1
2	1969 Corvair Monza	150	2020	1
3	1970 Plymouth Hemi Cuda	200	2020	1

```
6)
UPDATE sales
SET quantity = 500
WHERE id = 1;
```

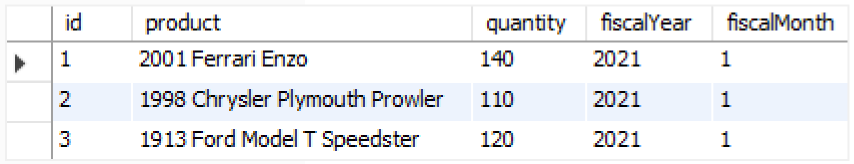
```
Error Code: 1644. The new quantity 500 cannot be 3 times greater than the current quantity 150
```

7.6.4 TYPES: AFTER UPDATE TRIGGERS

```
1)
DROP TABLE IF EXISTS Sales;

CREATE TABLE Sales (
    id INT AUTO_INCREMENT,
    product VARCHAR(100) NOT NULL,
    quantity INT NOT NULL DEFAULT 0,
    fiscalYear SMALLINT NOT NULL,
    fiscalMonth TINYINT NOT NULL,
    CHECK(fiscalMonth >= 1 AND fiscalMonth <= 12),
    CHECK(fiscalYear BETWEEN 2000 and 2050),
    CHECK (quantity >=0),
    UNIQUE(product, fiscalYear, fiscalMonth),
    PRIMARY KEY(id)
);
```

```
2)
INSERT INTO Sales(product, quantity, fiscalYear,
fiscalMonth)
VALUES
    ('2001 Ferrari Enzo',140, 2021,1),
    ('1998 Chrysler Plymouth Prowler', 110,2021,1),
    ('1913 Ford Model T Speedster', 120,2021,1);
```

```
3)
DROP TABLE IF EXISTS SalesChanges;

CREATE TABLE SalesChanges (
    id INT AUTO_INCREMENT PRIMARY KEY,
    salesId INT,
    beforeQuantity INT,
    afterQuantity INT,
    changedAt TIMESTAMP NOT NULL DEFAULT
CURRENT_TIMESTAMP
);
```



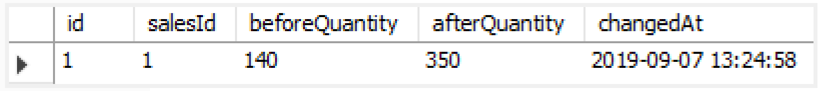
id	product	quantity	fiscalYear	fiscalMonth
1	2001 Ferrari Enzo	140	2021	1
2	1998 Chrysler Plymouth Prowler	110	2021	1
3	1913 Ford Model T Speedster	120	2021	1

7.6.4 TYPES: AFTER UPDATE TRIGGERS (CONTD)

```
4)
DELIMITER $$

CREATE TRIGGER after_sales_update
AFTER UPDATE
ON sales FOR EACH ROW
BEGIN
    IF OLD.quantity <> new.quantity THEN
        INSERT INTO SalesChanges(salesId,beforeQuantity,
afterQuantity)
        VALUES(old.id, old.quantity, new.quantity);
    END IF;
END$$

DELIMITER ;
```

```
5)
UPDATE Sales SET quantity = 350 WHERE id = 1;
```

```
6)
SELECT * FROM SalesChanges;
```



id	salesId	beforeQuantity	afterQuantity	changedAt
1	1	140	350	2019-09-07 13:24:58

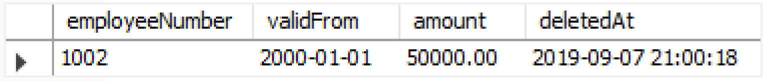
7.6.5 TYPES: BEFORE DELETE TRIGGERS

```
1)
DROP TABLE IF EXISTS Salaries;
CREATE TABLE Salaries (
    employeeNumber INT PRIMARY KEY,
    validFrom DATE NOT NULL,
    amount DEC(12 , 2 ) NOT NULL DEFAULT 0
);

2)
INSERT INTO salaries(employeeNumber,validFrom,amount)
VALUES
(1002,'2000-01-01',50000),
(1056,'2000-01-01',60000),
(1076,'2000-01-01',70000);
```

```
3)
DROP TABLE IF EXISTS SalaryArchives;

CREATE TABLE SalaryArchives (
    id INT PRIMARY KEY AUTO_INCREMENT,
    employeeNumber INT PRIMARY KEY,
    validFrom DATE NOT NULL,
    amount DEC(12 , 2 ) NOT NULL DEFAULT 0,
    deletedAt TIMESTAMP DEFAULT NOW()
);
```

```
4)
DELIMITER $$

CREATE TRIGGER before_salaries_delete
BEFORE DELETE
ON salaries FOR EACH ROW
BEGIN
    INSERT INTO
SalaryArchives(employeeNumber,validFrom,amount)
    VALUES(OLD.employeeNumber,OLD.validFrom,OLD.amount);
END$$

DELIMITER ;
```

```
5) DELETE FROM salaries
WHERE employeeNumber = 1002;

6) SELECT * FROM SalaryArchives;
```

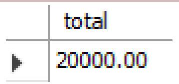


```
  employeeNumber  validFrom   amount    deletedAt
▶ 1002            2000-01-01  50000.00  2019-09-07 21:00:18
```

7.6.6 Types: After Delete Triggers

```
1)
DROP TABLE IF EXISTS Salaries;

CREATE TABLE Salaries (
    employeeNumber INT PRIMARY KEY,
    salary DECIMAL(10,2) NOT NULL DEFAULT 0
);

2)
INSERT INTO Salaries(employeeNumber,salary)
VALUES
    (1002,5000),
    (1056,7000),
    (1076,8000);
```

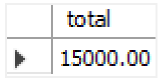
```
3)
DROP TABLE IF EXISTS SalaryBudgets;
CREATE TABLE SalaryBudgets(
    total DECIMAL(15,2) NOT NULL
);

INSERT INTO SalaryBudgets(total) SELECT SUM(salary)
FROM Salaries;
SELECT * FROM SalaryBudgets;
```



```
total
20000.00
```

```
4)
DELIMITER $$

CREATE TRIGGER after_salaries_delete
AFTER DELETE
ON Salaries FOR EACH ROW
BEGIN
    UPDATE SalaryBudgets
    SET total = total - old.salary;
END$$

DELIMITER ;
```

```
5) DELETE FROM Salaries
WHERE employeeNumber = 1002;

6) SELECT * FROM SalaryBudgets;
```



```
total
15000.00
```

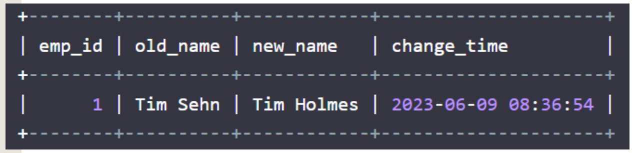
7.7 NEW AND OLD VALUES IN TRIGGERS

Trigger Event	OLD	NEW
INSERT	No	Yes
UPDATE	Yes	Yes
DELETE	Yes	No

- The trigger body can access the values of the column being affected by the operation.
- To distinguish between the value of the columns BEFORE and AFTER the event has fired, you use the NEW and OLD modifiers.
- For example, if you update the value in the description column, in the trigger body, you can access the value of the description column before the update OLD.description and the new value NEW.description.

```
1)
create table employees (
    id bigint primary key auto_increment,
    first_name varchar(100),
    last_name varchar(100)
);
insert into employees (first_name, last_name)
values ("Tim", "Sehn");

2)
create table name_changes (
    emp_id bigint,
    old_name varchar(200),
    new_name varchar(200),
    change_time timestamp default (current_timestamp())
);
```

```
3)
create trigger name_change_trigger
    after update on employees
    for each row
    insert into name_changes (emp_id, old_name, new_name) values
    (new.id,
    concat(old.first_name, concat(" ", old.last_name)),
    concat(new.first_name, concat(" ", new.last_name)));

4)
update employees set last_name = "Holmes" where first_name = "Tim";
```



```
+--------+----------+------------+---------------------+
| emp_id | old_name | new_name   | change_time         |
+--------+----------+------------+---------------------+
|      1 | Tim Sehn | Tim Holmes | 2023-06-09 08:36:54 |
+--------+----------+------------+---------------------+
```

WRAP UP
THANK YOU!