SUMMARY: STORED PROCEDURES AND TRIGGERS

SESSION OVERVIEW:

By the end of this session, the students will be able to

- Understand the concepts of different variants of stored procedures.
- Understand the concepts of different variants of triggers.

KEY TOPICS AND EXAMPLES:

Understanding the concepts of different variants of stored procedures:

1. **What is a Stored Procedure?**
   Stored procedures are routines that are stored in a database and executed by calling them. They can perform a wide range of operations, such as querying data, modifying data, and controlling transactions. The primary purpose of stored procedures is to enhance performance, security, and maintainability of database applications.

   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.

2. **Benefits and use cases:**
   - **Encapsulation of Logic:**
     Stored procedures encapsulate complex business logic, making it easier to manage and modify. Changes to the logic can be made in one place without affecting the application code that calls the procedure.
   - **Reusability:**
     Once a stored procedure is created, it can be reused by multiple applications and users, reducing code duplication and increasing consistency.
   - **Performance:**
     Stored procedures are precompiled and stored in the database. When executed, the database engine uses the compiled execution plan, which can lead to faster execution compared to ad-hoc SQL queries.
   - **Security:**
     Stored procedures can help enhance security by restricting direct access to the underlying tables. Users can be granted permission to execute specific procedures without granting them direct access to the tables.
   - **Maintainability:**
     By centralizing logic within stored procedures, it becomes easier to maintain and update. Changes to the procedure's logic do not require changes to the application code.

3. **Basic Syntax:**

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

NOTE:

- ***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.*

4. **Types of Stored 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 pass the new value back to the calling program.

5. **Examples:**
   To understand this we will first create a table in which it will be easier to understand.

```
CREATE TABLE student_info (
    stud_id INT PRIMARY KEY,
    stud_code INT,
    stud_name VARCHAR(50),
    subject VARCHAR(50),
    marks INT,
    phone VARCHAR(15)
);

-- Insert data
```

```
INSERT INTO student_info (stud_id, stud_code, stud_name, subject, marks,
phone) VALUES
(1, 101, 'Mark', 'English', 68, '3454569357'),
(2, 102, 'Joseph', 'Physics', 70, '9876543659'),
(3, 103, 'John', 'Maths', 70, '9765326975'),
(4, 104, 'Barack', 'Maths', 92, '87069873256'),
(5, 105, 'Rinky', 'Maths', 85, '6753159757'),
(6, 106, 'Adam', 'Science', 82, '79642256864'),
(7, 107, 'Andrew', 'Science', 83, '5674243579'),
(8, 108, 'Brayan', 'Science', 83, '7524316576'),
(9, 109, 'Alexandar', 'Biology', 67, '2347346438');
```

Create this table in your MySQL Workbench. Now we will create different types of stored Procedures in this table itself.

1. **Procedure without Parameter:**
   Suppose we want to display all records of this table whose marks are greater than 70 and count all the table rows. The following code creates a procedure named get_merit_students:

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

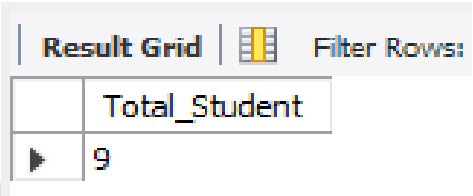
NOTE:
The provided SQL script defines a stored procedure named get_merit_student in MySQL, designed to perform two specific tasks: retrieving student records with marks greater than 70 and counting the total number of student records. The script begins by changing the statement delimiter from the default semicolon to a double ampersand (&&), allowing semicolons to be used within the procedure body without terminating the CREATE PROCEDURE statement prematurely. The CREATE PROCEDURE statement is used to define the get_merit_student procedure, which does not take any input parameters.

Within the procedure body, enclosed by the BEGIN and END statements, the first SQL query selects all columns from the student_info table where the marks column value exceeds 70. This query retrieves detailed records of students who have scored more than 70 marks, providing insight into high-performing students. Following this, the second SQL query counts the total number of student records in the student_info table, returning the result with the alias Total_Student. This query helps in determining the overall count of students in the database.

Let us call the procedure to verify the output:

```
CALL get_merit_student();
```

Output:



The above image is the output of the stored procedure that we have created above.

2. **Procedures with IN Parameter:**
   In this procedure, we have used the IN parameter as 'var1' of integer type to accept a number from users. Its body part fetches the records from the table using a SELECT statement and returns only those rows that will be supplied by the user. It also returns the total number of rows of the specified table. See the procedure code:

```
DELIMITER &&
CREATE PROCEDURE get_student (IN var1 INT)
BEGIN
    SELECT * FROM student_info LIMIT var1;
    SELECT COUNT(stud_code) AS Total_Student FROM student_info;
END &&
DELIMITER ;
```

After successful execution, we can call the procedure as follows:

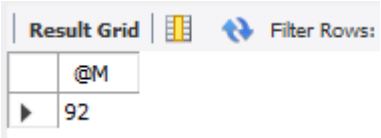
```
CALL get_student(4);
```

3. **Procedures with OUT Parameter:**
   In this procedure, we have used the OUT parameter as the 'highestmark' of integer type. Its body part fetches the maximum marks from the table using a MAX() function. See the procedure code:

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

This procedure's parameter will get the highest marks from the student_info table. 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 follows:

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



4. **Procedures with INOUT Parameter:**

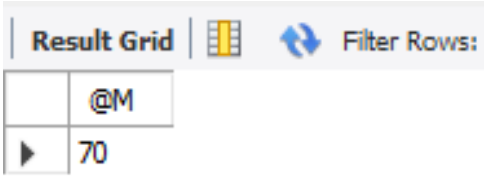
   In this procedure, we have used the INOUT parameter as 'var1' of integer type. Its body part first fetches the marks from the table with the specified id and then stores it into the same variable var1. The var1 first acts as the IN parameter and then OUT parameter. Therefore, we can call it the INOUT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

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

After successful execution, we can call the procedure as follows:

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

Output:



6. **How to show or list stored procedures in MySQL?**

   When we have several procedures in the MySQL server, it is very important to list all procedures. It is because sometimes the procedure names are the same in many databases. In that case, this query is very useful. We can list all procedure stored on the current MySQL server as follows:

```
SHOW PROCEDURE STATUS [LIKE 'pattern' | WHERE search_condition]
```

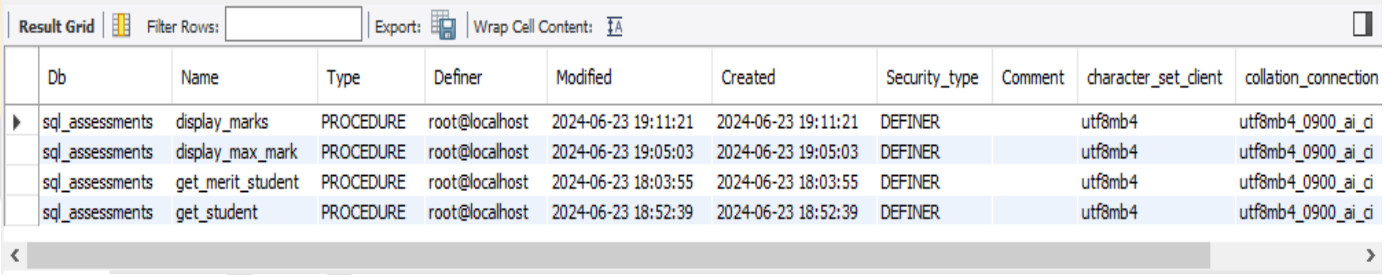
   This statement displays all stored procedure names, including their characteristics. If we want to display procedures in a particular database, we need to use the WHERE clause. In case we want to list stored procedures with a specific word, we need to use the LIKE clause.

We can list all stored procedure in the MySQL mystudentsb database using the below statement:

```
SHOW PROCEDURE STATUS WHERE db = 'sql_assessments';
```

Note: In whichever database you are working, put the name of that particular database.

Output:
It will give the below output where we can see that the mystudentdb database contains four stored procedures:



Db	Name	Type	Definer	Modified	Created	Security_type	Comment	character_set_client	collation_connection
sql_assessments	display_marks	PROCEDURE	root@localhost	2024-06-23 19:11:21	2024-06-23 19:11:21	DEFINER		utf8mb4	utf8mb4_0900_ai_ci
sql_assessments	display_max_mark	PROCEDURE	root@localhost	2024-06-23 19:05:03	2024-06-23 19:05:03	DEFINER		utf8mb4	utf8mb4_0900_ai_ci
sql_assessments	get_merit_student	PROCEDURE	root@localhost	2024-06-23 18:03:55	2024-06-23 18:03:55	DEFINER		utf8mb4	utf8mb4_0900_ai_ci
sql_assessments	get_student	PROCEDURE	root@localhost	2024-06-23 18:52:39	2024-06-23 18:52:39	DEFINER		utf8mb4	utf8mb4_0900_ai_ci

7. **How to delete/drop stored procedures in MySQL?**

   MySQL also allows a command to drop the procedure. When the procedure is dropped, it is removed from the database server also. The following statement is used to drop a stored procedure in MySQL:

```
DROP PROCEDURE [ IF EXISTS ] procedure_name;
```

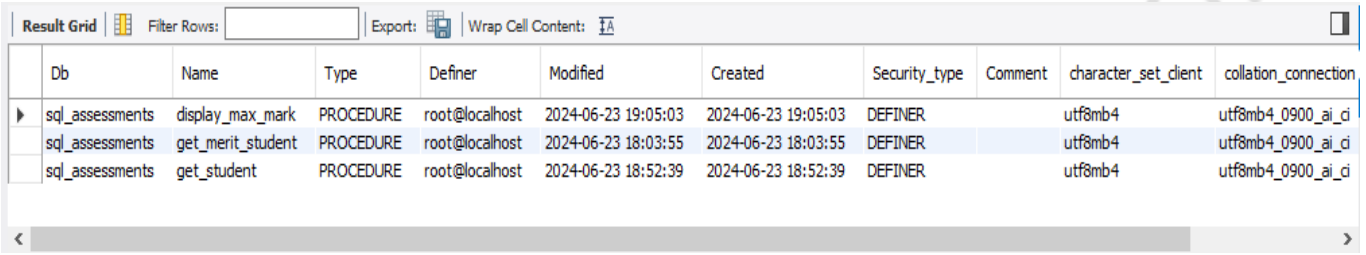
   Suppose we want to remove the procedure named display_marks from the mystudentdb database. We can do this by first selecting the database and then use the syntax as follows to remove the procedure:

```
DROP PROCEDURE display_marks;
```

   We can verify it by listing the procedure in the specified database using the SHOW PROCEDURE STATUS command. See the below output:

Output:

In this output you can notice that the procedure named as display_marks has been dropped and the output shows the other three procedures that we have created above.



Db	Name	Type	Definer	Modified	Created	Security_type	Comment	character_set_client	collation_connection
sql_assessments	display_max_mark	PROCEDURE	root@localhost	2024-06-23 19:05:03	2024-06-23 19:05:03	DEFINER		utf8mb4	utf8mb4_0900_ai_ci
sql_assessments	get_merit_student	PROCEDURE	root@localhost	2024-06-23 18:03:55	2024-06-23 18:03:55	DEFINER		utf8mb4	utf8mb4_0900_ai_ci
sql_assessments	get_student	PROCEDURE	root@localhost	2024-06-23 18:52:39	2024-06-23 18:52:39	DEFINER		utf8mb4	utf8mb4_0900_ai_ci

8. **How to alter the procedure in MySQL?**
   MySQL does not allow any command to alter the procedure in MySQL. However, it provides a command that is used to change the characteristics of a stored procedure. This command may alter more than one change in the procedure but does not modify the stored procedure's parameters or body. If we want to make such changes, we must drop and re-create the procedure using the DROP PROCEDURE and CREATE PROCEDURE statement.

   Suppose we want to add a comment to the existing procedure. In such a case, we can use the ALTER statement as follows to accomplish this task:

```
ALTER PROCEDURE get_merit_student
COMMENT 'It displays all records';
```

```
SHOW CREATE PROCEDURE get_merit_student;
```

Output:
We can notice that the procedure has been updated.

It is to note that we can alter the body of the stored procedure in MySQL using the workbench tool.

9. **Drawbacks of Stored Procedures:**

   - **Increased Memory Usage:** Using stored procedures can significantly increase memory usage for each connection that uses them.
   - **Higher CPU Usage:** Overuse of logical operations in stored procedures can lead to higher CPU usage, as the database server is not optimized for such tasks.
   - **Limited for Complex Business Logic:** Stored procedure constructs are not ideal for developing complex and flexible business logic.
   - **Debugging Challenges:** Debugging stored procedures is difficult. MySQL, in particular, lacks built-in debugging facilities.
   - **Development and Maintenance Complexity:** Developing and maintaining stored procedures can be challenging, requiring specialized skills that not all developers have, potentially leading to issues during both development and maintenance phases.

Understanding the concepts of different variants of triggers:

1. **What is a trigger?**
   A trigger in MySQL is a set of SQL statements that reside in a system catalog. It is a special type of stored procedure that is invoked automatically in response to an event. Each trigger is associated with a table, which is activated on any DML statement such as INSERT, UPDATE,

or DELETE.

A trigger is called a special procedure because it cannot be called directly like a stored procedure. The main difference between the trigger and procedure is that a trigger is called automatically when a data modification event is made against a table. In contrast, a stored procedure must be called explicitly.

2. **Why do we need triggers in SQL?**
    - Triggers help us to enforce business rules.
    - Triggers help us validate data even before it is inserted or updated.
    - Triggers help us keep a record log like maintaining audit trails in tables.
    - SQL triggers provide an alternative way to check the integrity of data.
    - Triggers provide an alternative way to run the scheduled task.
    - Triggers increase the performance of SQL queries because they do not need to be compiled each time the query is executed.
    - Triggers reduce the client-side code which saves time and effort.
    - Triggers help us to scale our application across different platforms.
    - Triggers are easy to maintain.

3. **Basic Syntax:**

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON table_name
FOR EACH ROW
BEGIN
    -- SQL statements to execute
END;
```

KEY COMPONENTS:

- ***CREATE TRIGGER: The command used to create a new trigger.***
- ***trigger_name: The name you give to the trigger. This should be unique within the database.***
- ***{BEFORE | AFTER}: Specifies whether the trigger should execute before or after the triggering event.***
- ***{INSERT | UPDATE | DELETE}: Specifies the event that activates the trigger.***
- ***ON table_name: Specifies the table the trigger is associated with.***
- ***FOR EACH ROW: Indicates that the trigger activates for each row affected by the triggering event.***
- ***BEGIN ... END: The block of SQL statements that defines the actions of the trigger***

NOTE:

When to use NEW and OLD:

- NEW is used with INSERT and UPDATE operations to access column values that are being applied.
- OLD is used with UPDATE and DELETE operations to access existing column values before they are updated or deleted.

Example:

```
CREATE TRIGGER before_user_insert
BEFORE INSERT ON users
FOR EACH ROW
BEGIN
    SET NEW.email = LOWER(NEW.email);
END;
```

Components of the Trigger

1. *CREATE TRIGGER before_user_insert: CREATE TRIGGER is the SQL statement used to create a new trigger.*
2. *before_user_insert is the name of the trigger. You can name it anything you like, but it's good practice to use a name that describes what the trigger does.*
3. *BEFORE INSERT: This specifies that the trigger will activate before a new row is inserted into the users table.*
4. *ON users: This specifies the table the trigger is associated with, in this case, the users table.*
5. *FOR EACH ROW: This indicates that the trigger will execute its actions for each row that is inserted into the users table.*
6. *BEGIN ... END: This block contains the SQL statements that define what the trigger will do when it is activated.*
7. *NEW: In the context of a BEFORE INSERT trigger, NEW is a keyword that refers to the row that is about to be inserted into the table.*
8. *NEW.email refers to the email field of the new row.*
9. *LOWER(): LOWER() is a built-in SQL function that converts a string to lowercase.*
10. *SET NEW.email = LOWER(NEW.email): This statement sets the email field of the new row to its lowercase equivalent. Essentially, it modifies the value of NEW.email before the row is actually inserted into the table.*

4. **Types of triggers in SQL:**
   Now we have understood the syntax of the triggers, we can move on to different types of triggers that can be used in SQL using an analogy and examples.

   - **Before Insert:** It is activated before the insertion of data into the table.
   - **After Insert:** It is activated after the insertion of data into the table.
   - **Before Update:** It is activated before the update of data in the table.
   - **After Update:** It is activated after the update of the data in the table.
   - **Before Delete:** It is activated before the data is removed from the table.
   - **After Delete:** It is activated after the deletion of data from the table.

Analogy:
In this section, we will understand the different triggers using an analogy and also look into the ways how we can create a trigger using different examples.

Imagine you own a restaurant with a well-organized kitchen. Your staff follows certain procedures whenever customers place orders, update their orders, or cancel them. Think of these procedures as triggers in SQL that automatically execute in response to specific events.

Specific Examples:

1. **BEFORE INSERT Trigger:**
   Analogy: When a new order comes in, the head chef checks the availability of all ingredients before starting to prepare the meal. If any ingredient is missing or not fresh, the order is adjusted or rejected.
   In SQL: A BEFORE INSERT trigger checks the data being inserted into the table to ensure it meets certain criteria. For example, verifying that the ordered quantity is available in stock before adding a new order to the system.

Example:

```
CREATE TRIGGER before_order_insert
BEFORE INSERT ON orders
FOR EACH ROW
BEGIN
    IF (SELECT quantity FROM inventory WHERE item_id = NEW.item_id) <
NEW.order_quantity THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Not enough stock for the order';
    END IF;
END;
```

Explanation:

The before_order_insert trigger is designed to ensure sufficient stock availability before a new order is inserted into the orders table. This trigger activates before the insertion of each new row, indicated by BEFORE INSERT ON orders FOR EACH ROW. It begins by checking if the quantity available in the inventory for the specific item being ordered (NEW.item_id) is less than the quantity requested in the new order (NEW.order_quantity). This is achieved through a subquery that selects the available stock from the inventory table. If the available stock is insufficient, the trigger raises an error using SIGNAL SQLSTATE '45000', with a custom message 'Not enough stock for the order'. This error prevents the insertion of the new order, ensuring that the database maintains inventory integrity and avoids overselling items.

NOTE: *SIGNAL SQLSTATE is used within a trigger, stored procedure, or function to raise a custom error condition. It allows you to specify an error code and an optional error message, which can help manage error handling and ensure data integrity.*

2. **BEFORE UPDATE Trigger**
   Analogy: Before a waiter updates an order (e.g., changing a dish), the kitchen staff checks if the new dish can be prepared within the given time frame. If it can't be, they notify the waiter.
   In SQL: A BEFORE UPDATE trigger ensures that data modifications meet certain conditions. For example, validating that a table reservation can be updated without conflicting with other reservations.

Example:

```
CREATE TRIGGER before_reservation_update
```

```
BEFORE UPDATE ON reservations
FOR EACH ROW
BEGIN
    IF EXISTS (SELECT 1 FROM reservations WHERE table_id = NEW.table_id
AND reservation_time = NEW.reservation_time AND reservation_id !=
OLD.reservation_id) THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'This table is already reserved for the
requested time';
    END IF;
END;
```

Explanation:
The before_reservation_update trigger is designed to prevent conflicts when updating reservations in a restaurant's booking system. This trigger is activated before any update operation on the reservations table. When a reservation is being updated, the trigger checks if there is already an existing reservation for the same table (table_id) at the same time (reservation_time) but with a different reservation ID (reservation_id). If such a conflicting reservation exists, the trigger raises an error by using the SIGNAL SQLSTATE '45000' command, which interrupts the update operation and returns an error message: 'This table is already reserved for the requested time'. This ensures that double bookings for the same table at the same time are prevented, maintaining the integrity of the reservation system.

3. **BEFORE DELETE Trigger:**
   Analogy: Before canceling a large party's reservation, the manager checks if any special arrangements (e.g., a custom cake or decorations) have been made. If so, the manager must ensure these arrangements are canceled too.
   In SQL: A BEFORE DELETE trigger ensures that certain conditions are met before a record is deleted. For example, preventing the deletion of a menu item that is part of an ongoing promotion.

Example:

```
CREATE TRIGGER before_menu_item_delete
BEFORE DELETE ON menu_items
FOR EACH ROW
BEGIN
    IF EXISTS (SELECT 1 FROM promotions WHERE item_id = OLD.item_id AND
end_date > NOW()) THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Cannot delete a menu item that is currently
part of a promotion';
    END IF;
END;
```

Explanation:

The before_menu_item_delete trigger is designed to ensure that menu items currently part of an active

promotion cannot be deleted from the menu_items table. When an attempt is made to delete a menu item, this trigger is activated before the deletion operation proceeds. The trigger checks if there are any entries in the promotions table that reference the menu item (item_id from the menu_items table) and have an end date later than the current date and time (NOW()). If such a promotion exists, the trigger raises an error using the SIGNAL SQLSTATE '45000' command, which stops the deletion process and returns an error message: 'Cannot delete a menu item that is currently part of a promotion'. This mechanism ensures that active promotions remain intact and prevents the deletion of items that are still being promoted, thus preserving the integrity of the promotion system.

4. **AFTER INSERT Trigger**
   Analogy: After a new order is placed, the kitchen system logs the order details and notifies the kitchen staff to start preparing the meal.
   In SQL: An AFTER INSERT trigger logs new records added to the table. For example, recording a new order in an order history table and notifying the kitchen staff.

Example:

```
CREATE TRIGGER after_order_insert
AFTER INSERT ON orders
FOR EACH ROW
BEGIN
    INSERT INTO order_history (order_id, order_time, status)
    VALUES (NEW.order_id, NOW(), 'Order Placed');
    CALL notify_kitchen(NEW.order_id);
END;
```

Explanation:

The SQL trigger after_order_insert is designed to automatically execute a set of actions after a new record is inserted into the orders table. This trigger operates as follows:

Upon the insertion of a new order, the trigger captures the new order's ID using the NEW.order_id reference. It then performs two primary actions. First, it inserts a new record into the order_history table with the captured order ID, the current timestamp (retrieved using the NOW() function), and a status message indicating that the order has been placed. This ensures that there is a historical record of the order being placed. Second, the trigger calls a stored procedure named notify_kitchen, passing the new order ID as an argument. This procedure likely contains the logic to notify the kitchen staff about the new order, ensuring they are promptly informed and can begin order preparation. This trigger effectively integrates order logging and kitchen notification into the order insertion process, streamlining operations and ensuring accurate record-keeping and timely communication.

5. **AFTER UPDATE Trigger:**
   Analogy: After a customer updates their order, the system updates the kitchen display and sends a notification to the kitchen staff about the changes.
   In SQL: An AFTER UPDATE trigger logs changes to records after they are updated. For example, recording the changes in an order history table.

Example:

```
CREATE TRIGGER after_order_update
AFTER UPDATE ON orders
FOR EACH ROW
BEGIN
    INSERT INTO order_history (order_id, order_time, status)
    VALUES (NEW.order_id, NOW(), 'Order Updated');
    CALL update_kitchen_display(NEW.order_id);
END;
```

6. **AFTER DELETE Trigger:**
   Analogy: After a reservation is canceled, the system logs the cancellation and notifies the staff to free up the table.
   In SQL: An AFTER DELETE trigger logs records that are deleted from a table. For example, recording the cancellation of a reservation in a history table.

Example:

```
CREATE TRIGGER after_reservation_delete
AFTER DELETE ON reservations
FOR EACH ROW
BEGIN
    INSERT INTO reservation_history (reservation_id, cancel_time)
    VALUES (OLD.reservation_id, NOW());
    CALL notify_staff_free_table(OLD.table_id);
END;
```

We can understand the availability of OLD and NEW modifiers with the below table:

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

5. **EXAMPLE:**

   Let's look into an example that will help you understand the concept of Triggers practically. For this example, we will create a hypothetical table named employees and perform the triggers in this example which will give a clear picture as the dataset we will be creating is going to be small.

Here is a query to CREATE the table.

```
CREATE TABLE employee(
```

```
    name varchar(45) NOT NULL,
    occupation varchar(35) NOT NULL,
    working_date date,
    working_hours varchar(10)
);
```

Now, let's insert some of the data in the table that we have just created.

```
INSERT INTO employee VALUES
('Robin', 'Scientist', '2020-10-04', 12),
('Warner', 'Engineer', '2020-10-04', 10),
('Peter', 'Actor', '2020-10-04', 13),
('Marco', 'Doctor', '2020-10-04', 14),
('Brayden', 'Teacher', '2020-10-04', 12),
('Antonio', 'Business', '2020-10-04', 11);
```

Next, we will create a BEFORE INSERT trigger. This trigger is invoked automatically to insert the working_hours = 0 if someone tries to insert working_hours < 0.

```
DELIMITER //
Create Trigger before_insert_empworkinghours
BEFORE INSERT ON employee FOR EACH ROW
BEGIN
IF NEW.working_hours < 0 THEN SET NEW.working_hours = 0;
END IF;
END //
```

Now, we can use the following statements to invoke this trigger:

```
INSERT INTO employee VALUES
('Markus', 'Former', '2020-10-08', 14);

INSERT INTO employee VALUES
('Alexander', 'Actor', '2020-10-012', -13);
```

Result Grid | Filter Rows: | Export:

name	occupation	working_date	working_hours
Robin	Scientist	2020-10-04	12
Warner	Engineer	2020-10-04	10
Peter	Actor	2020-10-04	13
Marco	Doctor	2020-10-04	14
Brayden	Teacher	2020-10-04	12
Antonio	Business	2020-10-04	11
Markus	Former	2020-10-08	14
Alexander	Actor	2020-10-12	0

employee 7

In this output, we can see that inserting the negative values into the working_hours column of the table will automatically fill the zero value by a trigger.

6. **How to show/list all the triggers:**

   The show or list trigger is much needed when we have many databases that contain various tables. Sometimes we have the same trigger names in many databases; this query plays an important role in that case. We can get the trigger information from the database server using the statement below.

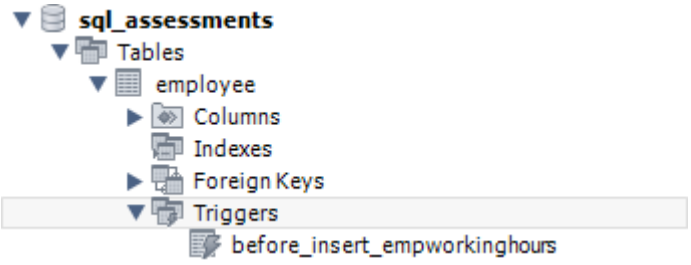
```
USE database_name;
SHOW TRIGGERS;
```

Output:

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Trigger	Event	Table	Statement	Timing	Created
before_insert_empworkinghours	INSERT	employee	BEGIN IF NEW.working_hours < 0 THEN SET N...	BEFORE	2024-06-21 16:33:29.07

Alternative:



The above image shows how we can check if the particular table has any triggers or not.

7. **How do you drop triggers?**

   We can drop/delete/remove a trigger in MySQL using the DROP TRIGGER statement. You must be very careful while removing a trigger from the table. Because once we have deleted

the trigger, it cannot be recovered. If a trigger is not found, the DROP TRIGGER statement throws an error.

Syntax:

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

NOTE:

- **Trigger_name:** This is the name of a trigger that we want to remove from the database server. It is a required parameter.
- **Schema_name:** It is the database name to which the trigger belongs. If we skip this parameter, the statement will remove the trigger from the current database.
- **IF_EXISTS:** It is an optional parameter that conditionally removes triggers only if they exist on the database server.

If we remove the trigger that does not exist, we will get an error. However, if we have specified the IF EXISTS clause, MySQL gives a NOTE instead of an error.

```
DROP TRIGGER IF EXISTS sql_assessments.before_insert_empworkinghours;
```

This will help you drop that specific trigger that has been mentioned above.