

Name: - Suyash Navanath Shinde
Course: - Database Management System Lab
Division: - TY CSF 2

Roll No: - D2223034
Batch: - B

Experiment No. -7

Aim: To implement the concept of cursors and trigger for the Given case studies

PL/SQL: Implicit and Explicit Cursors and Triggers.

- A. Write a Trigger for purchase table where total will be calculated when value is inserted in the table.
- B.
 1. Use the table production.products with Product_id, Product_name, Brand_id, Category_id, model_year, list_price,
 2. Declare two variables to hold product name and list price, and a cursor to hold the result of a query that retrieves product name and list price from the production.products table
 3. fetch each row from the cursor and print out the product name and list price

Software Required -MySQL

Theory :-

Cursor:

Cursors In MySQL, a cursor allows row-by-row processing of the result sets. A cursor is used for the result set and returned from a query. By using a cursor, you can iterate, or by step through the results of a query and perform certain operations on each row. The cursor allows you to iterate through the result set and then perform the additional processing only on the rows that require it.

A cursor contains the data in a loop. Cursors may be different from SQL commands that operate on all the rows returned by a query at one time.

There are some steps we have to follow, given below :

Declare a cursor

Open a cursor statement

Fetch the cursor

Close the cursor

1 . Declaration of Cursor : To declare a cursor you must use the DECLARE statement. With the help of the variables, conditions and handlers we need to declare a cursor before we can use it. First of all we will give the cursor a name, this is how we will refer to it later in the procedure. We can have more than one cursor in a single procedure so it's necessary to give it a name that will in some way tell us what it's doing. We then need to specify the select statement we want to associate with the cursor. The SQL statement can be any valid SQL statement and it is possible to use a dynamic where clause using variables or parameters as we have seen previously.

Syntax : DECLARE cursor_name CURSOR FOR select_statement;

2 . Open a cursor statement : To open a cursor we must use the open statement. If we want to fetch rows from it you must open the cursor.

Syntax : OPEN cursor_name;

3 . Cursor fetch statement : When we have to retrieve the next row from the cursor and move the cursor to the next row then you need to fetch the cursor.

Syntax : FETCH cursor_name INTO var_name;

If any row exists, then the above statement fetches the next row and cursor pointer moves ahead to the next row.

4 . Cursor close statement : By this statement closed the open cursor. Syntax: CLOSE_name;

By this statement we can close the previously opened cursor. If it is not closed explicitly then a cursor is closed at the end of the compound statement in which that was declared.

Example

QUERY: -

```
USE abc;
-- Create a sample table with dummy data
CREATE TABLE orders (
    order_id INT AUTO_INCREMENT PRIMARY KEY,
    order_date DATE,
    product_name VARCHAR(255),
    quantity INT,
    unit_price DECIMAL(10, 2)
);

-- Insert some sample data
INSERT INTO orders (order_date, product_name, quantity, unit_price) VALUES
    ('2023-10-01', 'Product A', 2, 19.99),
    ('2023-10-02', 'Product B', 3, 29.99),
    ('2023-10-03', 'Product C', 1, 39.99),
    ('2023-10-04', 'Product D', 4, 49.99);

select * from orders;
-- Create a procedure to calculate the total price of all orders
```

```
DELIMITER //
```

```
CREATE PROCEDURE calculate_total_orders(OUT total DECIMAL(10, 2))
```

```
BEGIN
```

```
    DECLARE done INT DEFAULT 0;
```

```
    DECLARE order_total DECIMAL(10, 2) DEFAULT 0;
```

```
    DECLARE order_quantity INT;
```

```
    DECLARE order_price DECIMAL(10, 2);
```

```
    -- Declare a cursor
```

```
    DECLARE order_cursor CURSOR FOR
```

```
        SELECT quantity, unit_price
```

```
        FROM orders;
```

```
    -- Declare continue handler to exit loop when no more rows are found
```

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

```
    -- Open the cursor
```

```
    OPEN order_cursor;
```

```
    -- Start fetching rows
```

```
    FETCH order_cursor INTO order_quantity, order_price;
```

```
    -- Loop through the result set
```

```
order_loop: LOOP
```

```
    -- Calculate the order subtotal and add it to the total
```

```
    SET order_total = order_total + (order_quantity * order_price);
```

```
    -- Fetch the next row
```

```
    FETCH order_cursor INTO order_quantity, order_price;
```

```
    -- Exit the loop when no more rows are found
```

```
    IF done = 1 THEN
```

```
        LEAVE order_loop;
```

```
    END IF;
```

```
END LOOP;
```

```
    -- Set the total to the calculated value
```

```
    SET total = order_total;
```

```
    -- Close the cursor
```

```
    CLOSE order_cursor;
```

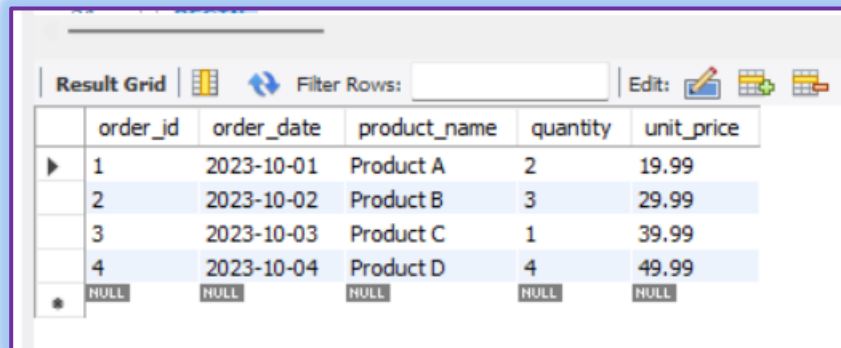
```
END;
```

```
//
```

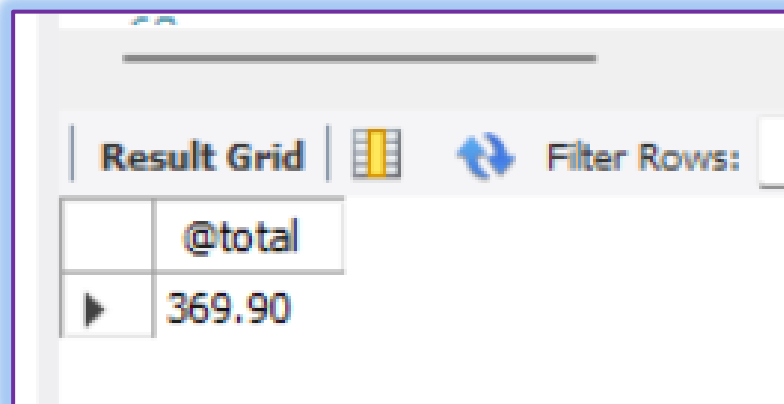
```
DELIMITER ;
```

```
-- Call the procedure to calculate the total price of all orders  
CALL calculate_total_orders(@total);  
SELECT @total;
```

OUTPUT: -



	order_id	order_date	product_name	quantity	unit_price
▶	1	2023-10-01	Product A	2	19.99
	2	2023-10-02	Product B	3	29.99
	3	2023-10-03	Product C	1	39.99
	4	2023-10-04	Product D	4	49.99
*	NULL	NULL	NULL	NULL	NULL



	@total
▶	369.90

Difference between Implicit and Explicit Cursors :

Implicit Cursors	Explicit Cursors
Implicit cursors are automatically created when select statements are executed.	Explicit cursors needs to be defined explicitly by the user by providing a name.
They are capable of fetching a single row at a time.	Explicit cursors can fetch multiple rows.
Closes automatically after execution.	Need to close after execution.
They are more vulnerable to errors such as Data errors, etc.	They are less vulnerable to
Provides less programmatic control to the users	User/Programmer has the entire control.
Implicit cursors are less efficient.	Comparative to Implicit cursors, explicit cursors are more efficient.
Implicit Cursors are defined as: BEGIN SELECT attr_name from table_name where CONDITION; END	Explicit cursors are defined as: DECLARE CURSOR cur_name IS SELECT attr_name from table_name where CONDITION; BEGIN

1. What is a cursor?
2. What are the types of cursor?
3. What is the use of a parameterized cursor?
4. What is the use of the cursor variable?
5. What is a normal cursor?
6. What are Explicit cursor attributes?

Trigger:

A trigger is a named MySQL object that activates when an event occurs in a table. Triggers are a particular type of stored procedure associated with a specific table.

Triggers allow access to values from the table for comparison purposes using NEW and OLD. The availability of the modifiers depends on the trigger event you use:

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

Checking or modifying a value when trying to insert data makes the NEW.<column name> modifier available. This is because a table is updated with new content. In contrast, an OLD.<column name> value does not exist for an insert statement because there is no information exists in its place beforehand.

When updating a table row, both modifiers are available. There is OLD.<column name> data which we want to update to NEW.<column name> data.

Finally, when removing a row of data, the OLD.<column name> modifier accesses the removed value. The NEW.<column name> does not exist because nothing is replacing the old value upon removal.

Create Triggers

Use the CREATE TRIGGER statement syntax to create a new trigger:

```
CREATE TRIGGER <trigger name> <trigger time > <trigger event>
```

```
ON <table name>
```

```
FOR EACH ROW
```

```
<trigger body>;
```

The best practice is to name the trigger with the following information:

```
<trigger time>_<table name>_<trigger event>
```

For example, if a trigger fires before insert on a table named employee, the best convention is to call the trigger:

before_employee_insert

Alternatively, a common practice is to use the following format:

<table name>_<first letter of trigger time><first letter of trigger name>

The before insert trigger name for the table employee looks like this:

employee_bi

The trigger executes at a specific time of an event on a table defined by <table name> for each row affected by the function.

Delete Triggers

To delete a trigger, use the DROP TRIGGER statement:

DROP TRIGGER <trigger name>;

```
mysql> DROP TRIGGER person_bi;
Query OK, 0 rows affected (0.41 sec)
```

Alternatively, use:

DROP TRIGGER IF EXISTS <trigger name>;

```
mysql> DROP TRIGGER IF EXISTS person_bi;
Query OK, 0 rows affected, 1 warning (0.09 sec)
```

```
mysql> DROP TRIGGER person_bi;
ERROR 1360 (HY000): Trigger does not exist
```

The error message does not display because there is no trigger, so no warning prints.

Create Example Database

Create a database for the trigger example codes with the following structure:

1. Create a table called *person* with *name* and *age* for columns.

SQL QUERY: -

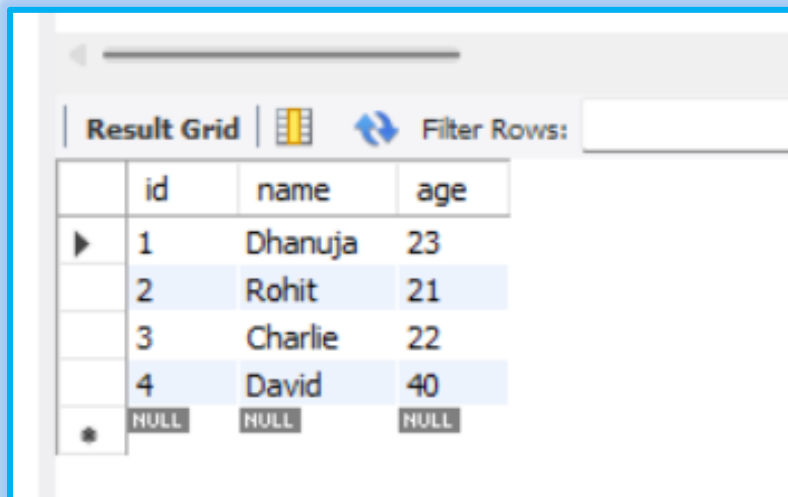
```
-- Create a new database
CREATE DATABASE trigger_example;

-- Switch to the new database
USE trigger_example;

-- Create a table called 'person' with 'name' and 'age' columns
CREATE TABLE person (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    age INT
);
```

```
-- Insert some sample data into the 'person' table
INSERT INTO person (name, age) VALUES
    ('Dhanuja', 23),
    ('Rohit', 21),
    ('Charlie', 22),
    ('David', 40);
```

OUTPUT: -



The screenshot shows a database client interface with a 'Result Grid' tab. It displays the contents of a table with columns 'id', 'name', and 'age'. The data is as follows:

	id	name	age
▶	1	Dhanuja	23
	2	Rohit	21
	3	Charlie	22
	4	David	40
✱	NULL	NULL	NULL

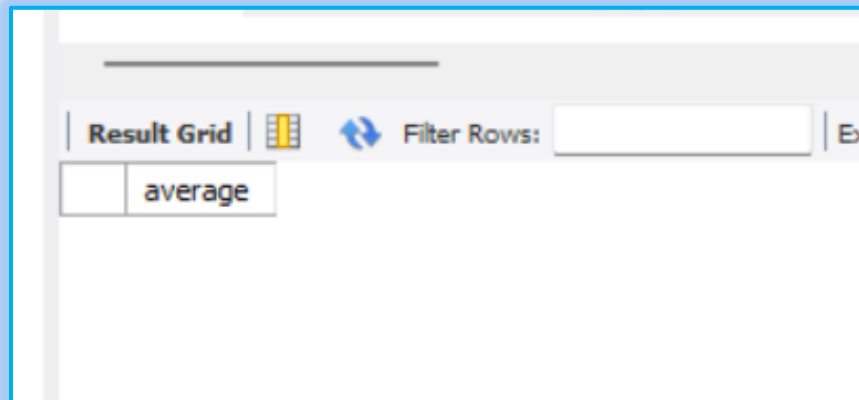
2. Create a table called *average_age* with a column called *average*:

SQL QUERY: -

```
-- Switch to the 'trigger_example' database if not already selected
USE trigger_example;

-- Create the 'average_age' table with a single column 'average'
CREATE TABLE average_age (
    average DECIMAL(10, 2)
);
```


OUTPUT: -



A screenshot of a database application window. At the top, there is a 'Result Grid' tab and a 'Filter Rows' input field. Below the grid, a single row is visible with the text 'average' in the first column.

average

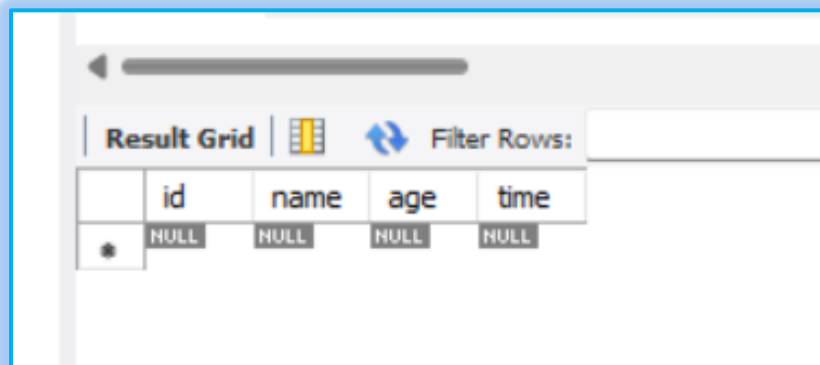
3. Create a table called *person_archive* with *name*, *age*, and *time* columns:

SQL QUERY: -

```
-- Switch to the 'trigger_example' database if not already selected
USE trigger_example;

-- Create the 'person_archive' table with 'name', 'age', and 'time' columns
CREATE TABLE person_archive (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    age INT,
    time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

OUTPUT: -



A screenshot of a database application window showing the 'Result Grid' tab. The grid has four columns: 'id', 'name', 'age', and 'time'. The first row contains NULL values for all four columns. A '*' symbol is visible in the first column of the first row.

	id	name	age	time
*	NULL	NULL	NULL	NULL

Note: The function NOW() records the current time.

Create a BEFORE INSERT Trigger

To create a BEFORE INSERT trigger, use:

```
CREATE TRIGGER <trigger name> BEFORE INSERT
```

```
ON <table name>
```

```
FOR EACH ROW
```

```
<trigger body>;
```

The BEFORE INSERT trigger gives control over data modification before committing into a database table. Capitalizing names for consistency, checking the length of an input, or catching faulty inputs with BEFORE INSERT triggers further provides value limitations before entering new data.

BEFORE INSERT Trigger Example

Create a BEFORE INSERT trigger to check the age value before inserting data into the *person* table:

```
delimiter //
```

```
CREATE TRIGGER person_bi BEFORE INSERT
```

```
ON person
```

```
FOR EACH ROW
```

```
IF NEW.age < 18 THEN
```

```
SIGNAL SQLSTATE '50001' SET MESSAGE_TEXT = 'Person must be older than 18.';
```

```
END IF; //
```

```
delimiter ;
```

```
mysql> delimiter //  
mysql> CREATE TRIGGER person_bi BEFORE INSERT  
-> ON person  
-> FOR EACH ROW  
-> IF NEW.age < 18 THEN  
-> SIGNAL SQLSTATE '50001' SET MESSAGE_TEXT = 'Person must be older than 18.';  
-> END IF; //  
Query OK, 0 rows affected (0.17 sec)  
  
mysql> delimiter ;
```

Inserting data activates the trigger and checks the value of *age* before committing the information:

```
INSERT INTO person VALUES ('John', 14);
```

```
mysql> INSERT INTO person VALUES ('John', 14);  
ERROR 1644 (50001): Person must be older than 18.
```

The console displays the descriptive error message. The data does not insert into the table because of the failed trigger check.

Create an AFTER INSERT Trigger
Create an AFTER INSERT trigger with:

```
CREATE TRIGGER <trigger name> AFTER INSERT  
ON <table name>
```

```
FOR EACH ROW
```

```
<trigger body>;
```

The AFTER INSERT trigger is useful when the entered row generates a value needed to update another table.

AFTER INSERT Trigger Example

Inserting a new row into the *person* table does not automatically update the average in the *average_age* table. Create an AFTER INSERT trigger on the *person* table to update the *average_age* table after insert:

```
delimiter //
```

```
CREATE TRIGGER person_ai AFTER INSERT
```

```
ON person
```

```
FOR EACH ROW
```

```
UPDATE average_age SET average = (SELECT AVG(age) FROM person); //
```

```
delimiter ;
```

```
mysql> delimiter //  
mysql> CREATE TRIGGER person_ai AFTER INSERT  
-> ON person  
-> FOR EACH ROW  
-> UPDATE average_age SET average = (SELECT AVG(age) FROM person); //  
Query OK, 0 rows affected (0.30 sec)  
  
mysql> delimiter ;
```

Inserting a new row into the *person* table activates the trigger:

```
INSERT INTO person VALUES ('John', 19);
```

```
mysql> INSERT INTO person VALUES ('John', 19); ←
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT * FROM person;
```

```
+-----+-----+
| name  | age  |
+-----+-----+
| Matthew | 25 |
| Mark   | 20 |
| John   | 19 |
+-----+-----+
3 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM average_age;
```

```
+-----+
| average |
+-----+
| 21.33333333 |
+-----+
1 row in set (0.00 sec)
```

The data successfully commits to the *person* table and updates the *average_age* table with the correct average value.

Create a BEFORE UPDATE Trigger

Make a BEFORE UPDATE trigger with:

```
CREATE TRIGGER <trigger name> BEFORE UPDATE
```

```
ON <table name>
```

```
FOR EACH ROW
```

```
<trigger body>;
```

The BEFORE UPDATE triggers go together with the BEFORE INSERT triggers. If any restrictions exist before inserting data, the limits should be there before updating as well.

BEFORE UPDATE Trigger Example

If there is an age restriction for the *person* table before inserting data, the age restriction should also exist before updating information. Without the BEFORE UPDATE trigger, the age check trigger is easy to avoid. Nothing restricts editing to a faulty value.

Add a BEFORE UPDATE trigger to the *person* table with the same body as the BEFORE INSERT trigger:

```
delimiter //
```

```
CREATE TRIGGER person_bu BEFORE UPDATE
```

```
ON person
```

```
FOR EACH ROW
```

```
IF NEW.age < 18 THEN
```

```
SIGNAL SQLSTATE '50002' SET MESSAGE_TEXT = 'Person must be older than 18.');
```

```
END IF; //
```

```
delimiter ;
```

```
mysql> delimiter ;
mysql> delimiter //
mysql> CREATE TRIGGER person_bu BEFORE UPDATE
  -> ON person
  -> FOR EACH ROW
  -> IF NEW.age < 18 THEN
  -> SIGNAL SQLSTATE '50002' SET MESSAGE_TEXT = 'Person must be older than 18';
  -> END IF; //
Query OK, 0 rows affected (0.36 sec)

mysql> delimiter ;
```

Updating an existing value activates the trigger check:

```
UPDATE person SET age = 17 WHERE name = 'John';
```

```
mysql> UPDATE person SET age = 17 WHERE name = 'John';
ERROR 1644 (50001): Person must be over the age of 18.
```

Updating the *age* to a value less than 18 displays the error message, and the information does not update.

Create an AFTER UPDATE Trigger

Use the following code block to create an AFTER UPDATE trigger:

```
CREATE TRIGGER <trigger name> AFTER UPDATE
```

```
ON <table name>
```

```
FOR EACH ROW
```

```
<trigger body>;
```

The AFTER UPDATE trigger helps keep track of committed changes to data. Most often, any changes after inserting information also happen after updating data.

AFTER UPDATE Trigger Example

Any successful updates to the *age* data in the table *person* should also update the intermediate average value calculated in the *average_age* table.

Create an AFTER UPDATE trigger to update the *average_age* table after updating a row in the *person* table:

```
delimiter //
```

```
CREATE TRIGGER person_au AFTER UPDATE
```

```
ON person
```

```
FOR EACH ROW
```

```
UPDATE average_age SET average = (SELECT AVG(age) FROM person); //
delimiter ;
```

```
mysql> delimiter //
mysql> CREATE TRIGGER person_au AFTER UPDATE
-> ON person
-> FOR EACH ROW
-> UPDATE average_age SET average = (SELECT AVG(age) FROM person); //
Query OK, 0 rows affected (0.93 sec)

mysql> delimiter ;
```

Updating existing data changes the value in the *person* table:

```
UPDATE person SET age = 21 WHERE name = 'John';
```

```
mysql> UPDATE person SET age = 21 WHERE name = 'John'; ←
Query OK, 1 row affected (0.02 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

```
mysql> SELECT * FROM person;
+-----+-----+
| name  | age |
+-----+-----+
| Matthew | 25 |
| Mark   | 20 |
| John   | 21 |
+-----+-----+
3 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM average_age;
+-----+
| average |
+-----+
| 22      |
+-----+
1 row in set (0.00 sec)
```

Updating the table *person* also updates the average in the *average_age* table.

Create a BEFORE DELETE Trigger

To create a BEFORE DELETE trigger, use:

```
CREATE TRIGGER <trigger name> BEFORE DELETE
```

```
ON <table name>
```

```
FOR EACH ROW
```

```
<trigger body>;
```

The BEFORE DELETE trigger is essential for security reasons. If a parent table has any children attached, the trigger helps block deletion and prevents orphaned tables. The trigger also allows archiving data before deletion.

BEFORE DELETE Trigger Example

Archive deleted data by creating a BEFORE DELETE trigger on the table *person* and insert the values into the *person_archive* table:

```
delimiter //

CREATE TRIGGER person_bd BEFORE DELETE

ON person

FOR EACH ROW

INSERT INTO person_archive (name, age)

VALUES (OLD.name, OLD.age); //

delimiter ;
```

```
mysql> delimiter //
mysql> CREATE TRIGGER person_bd BEFORE DELETE
-> ON person
-> FOR EACH ROW
-> INSERT INTO person_archive (name, age)
-> VALUES (OLD.name, OLD.age); //
Query OK, 0 rows affected (0.33 sec)

mysql> delimiter ;
```

Deleting data from the table *person* archives the data into the *person_archive* table before deleting:

```
DELETE FROM person WHERE name = 'John';
```

```
mysql> DELETE FROM person WHERE name = 'John'; ←
Query OK, 1 row affected (0.03 sec)
```

```
mysql> SELECT * FROM person;
+-----+-----+
| name  | age |
+-----+-----+
| Matthew | 25 |
| Mark   | 20 |
+-----+-----+
2 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM person_archive;
+-----+-----+-----+
| name | age | time                |
+-----+-----+-----+
| John | 21 | 2021-04-09 09:37:57 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

Inserting the value back into the *person* table keeps the log of the deleted data in the *person_archive* table:

```
INSERT INTO person VALUES ('John', 21);
```

```
mysql> INSERT INTO person(name, age) VALUES ('John', 21); ←
```

```
Query OK, 1 row affected (0.14 sec)
```

```
mysql> SELECT * FROM person;
```

```
+-----+-----+
| name  | age  |
+-----+-----+
| Matthew | 25  |
| Mark   | 20  |
| John   | 21  | ←
+-----+-----+
3 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM person_archive;
```

```
+-----+-----+-----+
| name | age | time                |
+-----+-----+-----+
| John | 21  | 2021-04-09 09:37:57 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

The BEFORE DELETE trigger is useful for logging any table change attempts.

Create an AFTER DELETE Trigger

Make an AFTER DELETE trigger with:

```
CREATE TRIGGER <trigger name> AFTER DELETE
```

```
ON <table name>
```

```
FOR EACH ROW
```

```
<trigger body>;
```

The AFTER DELETE triggers maintain information updates that require the data row to disappear before making the updates.

AFTER DELETE Trigger Example

Create an AFTER DELETE trigger on the table *person* to update the *average_age* table with the new information:

```
delimiter //
```

```
CREATE TRIGGER person_ad AFTER DELETE
```

```
ON person
```

```
FOR EACH ROW
```

```
UPDATE average_age SET average = (SELECT AVG(person.age) FROM person); //
```

```
delimiter ;
```



```
mysql> delimiter //
mysql> CREATE TRIGGER person_ad AFTER DELETE
    -> ON person
    -> FOR EACH ROW
    -> UPDATE average_age SET average = (SELECT AVG(age) FROM person); //
Query OK, 0 rows affected (0.25 sec)

mysql> delimiter ;
```

Deleting a record from the table *person* updates the *average_age* table with the new average:

```
mysql> DELETE FROM person WHERE name = 'John'; ←
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM person;
+-----+-----+
| name  | age  |
+-----+-----+
| Matthew | 25 |
| Mark   | 20 |
+-----+-----+
2 rows in set (0.00 sec)

mysql> SELECT * FROM average_age;
+-----+
| average |
+-----+
| 22.5    |
+-----+
1 row in set (0.00 sec)
```

Without the AFTER DELETE trigger, the information does not update automatically.

Case Study:

Consider the Following schema

Emp (eno, ename, designation, salary, dno)

Dept (dno dname,dhod)

1. Increment the salary of all 'comp' dept employees with 10 %
2. Display the ename and designation if salary is above 35000 of dno 101
3. Create the trigger on emp Table: The deleted record from the emp table should be insert in Dummy Table

Consider the Following schema

Boats(Bid, Name, Bcolor)

Sailors(Sid,Sname, Srating)

Reserves (Bid, Sid, Date of Reservation)

1. Create the trigger on Sailors Table: The Rating of the Sailor should get incremented by 1 once the sailor reserves a boat.
2. Create the Cursor which will Insert the Sid, Sname, Bid who reserved red color Boat in Red_Boats Table;

Consider the Following schema

Books (Sid, Bid, BName, BPrice)

Transactions (Sid,Bid, Date_Issue,Date_Return, Status)

Return_books (Sid,Bid, Fine_amout)

1. Create a trigger on Books Table such that insertion of Books details to insert a record in Transaction table (Sid and Bid values should be Same, others values can be Assumable)
2. Display the Book Names Issued to Sid 'XXX' using Cursor
3. Create a trigger on the Books Table so that BName will be stored in uppercase.
4. Update the Date_Return of Sid 'xxx' . Then Create the Trigger to Update the Status of Book to 'Return'
5. Create a cursor which will calculate the Fine_Amount and insert the 'Return' Books in Return_books table.

Conditions:

If No of Days Between Date_Issue and Date_Return > 15 Days, Fine_amount is : 10 Rs Per Day

If No of Days Between Date_Issue and Date_Return > 16 Days and < 30 Days, Fine_amount is : 20 Rs Per Day

If No of Days Between Date_Issue and Date_Return > 30 Days, Fine_amount is : 30 Rs Per Day