What is PL/SQL?

Oracle PL/SQL is an extension of SQL language that combines the data manipulation power of SQL with the processing power of procedural language to create super powerful SQL queries. PL/SQL ensures seamless processing of SQL statements by enhancing the security, portability, and robustness of the Database.

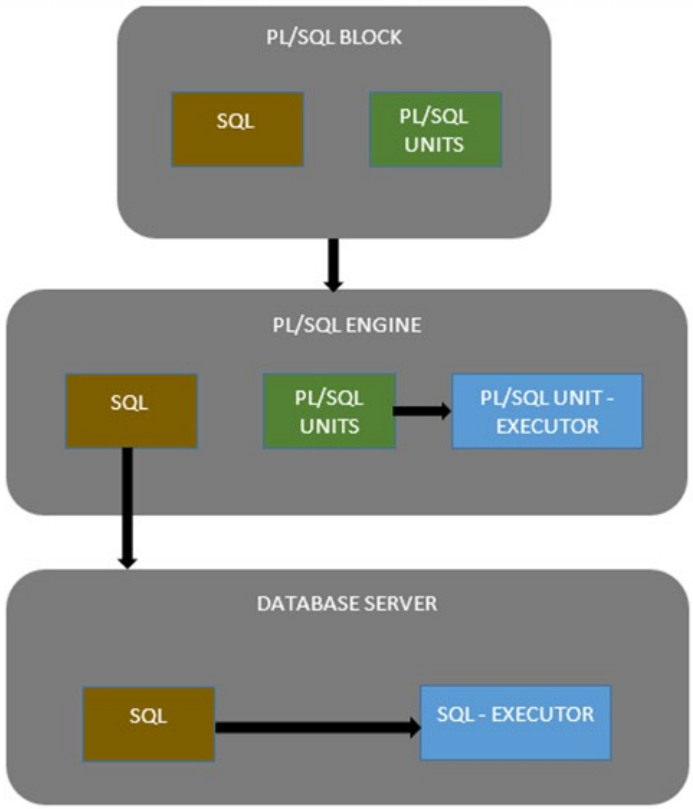
PL/SQL means instructing the compiler ‘what to do’ through SQL and ‘how to do’ through its procedural way. Similar to other database languages, it gives more control to the programmers by the use of loops, conditions and object-oriented concepts. The PL/SQL Full form is “Procedural Language extensions to SQL”.

What is PL/SQL Developer?

PL/SQL Developer is a free Integrated Development Environment provided by Oracle to develop Software in Oracle Database environment and perform various Database tasks with ease. The PL/SQL Developer IDE provides with GUI and Plugins to use in order to help the end users save the time on their Database tasks

Architecture of PL/SQL

The Below PL/SQL Example is a pictorial representation of PL/SQL Architecture.



PL/SQL Architecture Diagram

The PL/SQL architecture mainly consists of following three components:

1. PL/SQL Block
2. PL/SQL Engine
3. Database Server

Basic Syntax Of PL SQL

PL SQL is structured in logical blocks of code. Each block has multiple subsections comprising of the following:

PL/SQL Block consists of three sections:

- The Declaration section (optional).
- The Execution section (mandatory).
- The Exception Handling (or Error) section (optional).

1. **Declaration:** This section begins with the DECLARE keyword. It is not considered as the required one and has variables, subprograms, and so on.
2. **Executable Commands:** This section begins with BEGIN and END keywords respectively. It is considered a required one and contains PL/SQL statements. It consists of at least one executable line of code.
3. **Exception Handling:** This section begins with the keyword EXCEPTION. It comprises the types of exceptions that the code will handle.
4. **Begin:** This is the keyword used for pointing to the execution block. It is required in a PL/SQL code where actual business logic is described.
5. **End:** This is the keyword used to determine the end of the block of code.

Structure of PL/SQL block:

```
DECLARE
    Variable declaration
BEGIN
    Program Execution
EXCEPTION
    Exception handling
END;
```

SQL Command Categories

SQL commands are grouped into four major categories depending on their functionality. They are as follows:

Data Definition Language (DDL)

These SQL commands are used for creating, modifying, and dropping the structure of database objects. The commands are CREATE, ALTER, DROP, RENAME, and TRUNCATE.

Data Manipulation Language (DML)

These SQL commands are used for storing, retrieving, modifying, and deleting data. These commands are SELECT, INSERT, UPDATE, and DELETE.

Transaction Control Language (TCL)

These SQL commands are used for managing changes affecting the data. These commands are COMMIT, ROLLBACK, and SAVEPOINT.

Data Control Language (DCL)

These SQL commands are used for providing security to database objects. These commands are GRANT and REVOKE.

The PL/SQL Engine:

Oracle uses a **PL/SQL** engine to processes the PL/SQL statements. A PL/SQL language code can be stored in the client system (client-side) or in the database (server-side).

Scope of PS/SQL Variables

PL/SQL allows the nesting of Blocks within Blocks i.e, the Execution section of an outer block can contain inner blocks. Therefore, a variable which is accessible to an outer Block is also accessible to all nested inner Blocks. The variables declared in the inner blocks are not accessible to outer blocks. Based on their declaration we can classify variables into two types.

- *Local* variables - These are declared in a inner block and cannot be referenced by outside Blocks.
- *Global* variables - These are declared in a outer block and can be referenced by its itself and by its inner blocks.

For Example: In the below example we are creating two variables in the outer block and assigning thier product to the third variable created in the inner block. The variable 'var_mult' is declared in the inner block, so cannot be accessed in the outer block i.e. it cannot be accessed after line 11. The variables 'var_num1' and 'var_num2' can be accessed anywhere in the block.

```
DECLARE
  a integer := 30;
  b integer := 40;
  c integer;
```

```
  f real;
BEGIN
  c := a + b;
  dbms_output.put_line('Value of c: ' || c);
  f := 100.0/3.0;
  dbms_output.put_line('Value of f: ' || f);
END;
```

create the student table which consist of stud_id,name,mark

insert the values to student table

```
delimiter //
create procedure student_prod()
begin
SELECT mark, name FROM mydb.student where stud_id=2;
end //
```

```
DELIMITER $$
USE `mydb`$$
CREATE DEFINER=`root`@`localhost` PROCEDURE `check_output`()
begin
declare v int default 3;
select v+20;
end$$

DELIMITER ;
;
```

```
DECLARE
   -- Global variables
   num1 number := 95;
   num2 number := 85;
BEGIN
   dbms_output.put_line('Outer Variable num1: ' || num1);
```

```
   dbms_output.put_line('Outer Variable num2: ' || num2);
   DECLARE
      -- Local variables
      num1 number := 195;
      num2 number := 185;
   BEGIN
      dbms_output.put_line('Inner Variable num1: ' || num1);
      dbms_output.put_line('Inner Variable num2: ' || num2);
   END;
END;
```

PL/SQL Constants

As the name implies a *constant* is a value used in a PL/SQL Block that remains unchanged throughout the program. A constant is a user-defined literal value. You can declare a constant and use it instead of actual value.

For example: If you want to write a program which will increase the salary of the employees by 25%, you can declare a constant and use it throughout the program. Next time when you want to increase the salary again you can change the value of the constant which will be easier than changing the actual value throughout the program.

General Syntax to declare a constant is:

```
constant_name CONSTANT datatype := VALUE;
```

- *constant_name* is the name of the constant i.e. similar to a variable name.
- The word *CONSTANT* is a reserved word and ensures that the value does not change.
- *VALUE* - It is a value which must be assigned to a constant when it is declared. You cannot assign a value later.

```
DECLARE
salary_increase CONSTANT number (3) := 10;
```

You *must* assign a value to a constant at the time you declare it. If you do not assign a value to a constant while declaring it and try to assign a value in the execution section, you will get a error. If you execute the below Pl/SQL block you will get error.

```
DECLARE
```

```
 salary_increase CONSTANT number(3);
BEGIN
 salary_increase := 100;
 dbms_output.put_line (salary_increase);
END;
```

PL/SQL Records

What are records?

Records are another type of datatypes which oracle allows to be defined as a placeholder. Records are composite datatypes, which means it is a combination of different scalar datatypes like char, varchar, number etc. Each scalar data types in the record holds a value. A record can be visualized as a row of data. It can contain all the contents of a row.

Declaring a record:

To declare a record, you must first define a composite datatype; then declare a record for that type.

Conditional Statements in PL/SQL

As the name implies, PL/SQL supports programming language features like conditional statements, iterative statements.

```
DECLARE
   a number(2) := 10;
BEGIN
   a:= 10;
  -- check the boolean condition using if statement
   IF( a < 20 ) THEN
      -- if condition is true then print the following
      dbms_output.put_line('a is less than 20 ' );
   END IF;
   dbms_output.put_line('value of a is : ' || a);
END;
/
```

```
create database plsqlstatement;

USE mydb;

use plsqlstatement;

DELIMITER //
CREATE PROCEDURE condition1()
begin
```

```
declare a int default 50;
IF( a < 20 ) THEN
select 'a is less than 20';
end if;
select 'value of a is ',a;
end //

call condition1;
```

```
create the table Employee - id
Insert the value

DELIMITER //
create Procedure Emp_prod()
BEGIN
DECLARE c_id int default 1;
DECLARE c_sal int;
  SELECT  salary  INTO  c_sal FROM customers WHERE id = c_id;
   IF (c_sal <= 10000) THEN
     UPDATE customers SET salary =  salary + 1000 WHERE id =
c_id;
      select 'Salary updated';
   END IF;
END;
/
```

```
delimiter //
create procedure if_else()
begin
declare a int default 100;
if(a<20) then
     select 'a is less than 20';
else
     select 'a is greater than 20';
end if;
select 'value of a is ',a;
end;

Output:
```

```
a is greater than 20
value of a is 100
```

```
delimiter //
create procedure looping()
for_loop: begin
declare c int default 10;
loop
     select c;
    set c:=c+10;
    if c>50 then
         leave for_loop;
     end if;
end loop;
select 'After End c is: ',c;
end;

delimiter //
create procedure strings()
begin
declare name varchar(20);
declare company varchar(30);
declare choice char(1);
set name:='John Smith';
set company:='Infotech';
set choice:='y';
if choice='y' then
     select name;
    select company;
end if;
end;
```

PL/SQL - Procedures

A Procedure in PL/SQL is **a subprogram unit that consists of a group of PL/SQL statements that can be called by name**.

Database Procedures (sometimes referred to as Stored Procedures or Procs) are subroutines that can contain one or more SQL statements that perform a specific task. They can be used **for data validation, access control, or to reduce network traffic between clients and the DBMS servers**.

Advantages of procedures in SQL

The main purpose of stored procedures in SQL is **to hide direct SQL queries from the code and improve the performance of database operations such as select, update, and delete data**.

PL/SQL **enables users to utilize all SQL data manipulation, cursor control, transaction statements and all other SQL functions, operators and pseudo-columns**. Users aren't required to convert between PL/SQL and SQL data types.

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the **calling program**.

A subprogram can be created –

- At the schema level
- Inside a package
- Inside a PL/SQL block

At the schema level, subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter **'PL/SQL - Packages'**.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms –

- **Functions** – These subprograms return a single value; mainly used to compute and return a value.
- **Procedures** – These subprograms do not return a value directly; mainly used to perform an action.

What is a Stored Procedure?
A **stored procedure** or in simple a **proc** is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages.

A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure. The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block.

A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.

Procedures: Passing Parameters

We can pass parameters to procedures in three ways.
1) IN-parameters
2) OUT-parameters
3) IN OUT-parameters

A procedure may or may not return any value.

General Syntax to create a procedure is:

```
CREATE [OR REPLACE] PROCEDURE proc_name [list of parameters]
IS
   Declaration section
BEGIN
   Execution section
EXCEPTION
  Exception section
END;
```

IS - marks the beginning of the body of the procedure and is similar to DECLARE in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.

The syntax within the brackets [ ] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.

Procedures: Example

The below example creates a procedure 'employer_details' which gives the details of the employee.

```
1> CREATE OR REPLACE PROCEDURE employer_details
2> IS
3>  CURSOR emp_cur IS
4>  SELECT first_name, last_name, salary FROM emp_tbl;
5>  emp_rec emp_cur%rowtype;
6> BEGIN
7>  FOR emp_rec in sales_cur
8>  LOOP
9>  dbms_output.put_line(emp_cur.first_name || ' ' ||
emp_cur.last_name
10>    || ' ' ||emp_cur.salary);
```

```
11> END LOOP;
12>END;
13> /
```
How to execute a Stored Procedure?
There are two ways to execute a procedure.

1) From the SQL prompt.

```
EXECUTE [or EXEC] procedure_name;
```
2) Within another procedure – simply use the procedure name.

```
procedure_name;
```
NOTE: In the examples given above, we are using backward slash '/' at the end of the program. This indicates the oracle engine that the PL/SQL program has ended and it can begin processing the statements.

What is a Function in PL/SQL?
A function is a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value.

General Syntax to create a function is

```
CREATE [OR REPLACE] FUNCTION function_name [parameters]
RETURN return_datatype;
IS
Declaration_section
BEGIN
Execution_section
Return return_variable;
EXCEPTION
exception section
Return return_variable;
END;
```

1) **Return Type:** The header section defines the return type of the function. The return datatype can be any of the oracle datatype like varchar, number etc.
   2) The execution and exception section both should return a value which is of the datatype defined in the header section.

For example, let's create a frunction called "employer_details_func' similar to the one created in stored proc

delimiter //

```
1> CREATE OR REPLACE FUNCTION employer_details_func
2>    RETURN VARCHAR(20);
3> IS
5>    emp_name VARCHAR(20);
6> BEGIN
7>   SELECT first_name INTO emp_name
8>   FROM emp_tbl WHERE empID = '100';
9>   RETURN emp_name;
10> END;
11> /
```

MySQL Cursor DECLARE Statement

A cursor in database is a construct which allows you to iterate/traversal the records of a table. In MySQL you can use cursors with in a stored program such as procedures, functions etc.

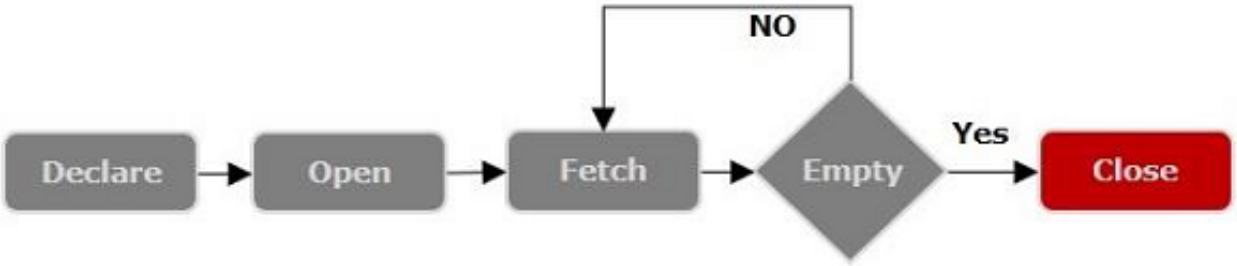
In other words, you can iterate though the records of a table from a MySQL stored program using the cursors. The cursors provided by MySQL are embedded cursors. They are –

- **READ ONLY** – Using these cursors you cannot update any table.
- **Non-Scrollable** – Using these cursors you can retrieve records from a table in one direction i.e., from top to bottom.
- **Asensitive** – These cursors are insensitive to the changes that are made in the table i.e. the modifications done in the table are not reflected in the cursor.
  Which means if we have created a cursor holding all the records in a table and, meanwhile if we add some more records to the table, these recent changes will not be reflected in the cursor we previously obtained.

While Declaring cursors in a stored program you need to make sure these (cursor declarations) always follow the variable and condition declarations.

To use a cursor, you need to follow the steps given below (in the same order)

- Declare the cursor using the *DECLARE* Statement.
- Declare variables and conditions.
- Open the declared cursor using the *OPEN* Statement.
- Retrieve the desired records from a table using the *FETCH* Statement.
- Finally close the cursor using the *CLOSE*statement.




```
CREATE TABLE training (
   ID INT PRIMARY KEY,
   TITLE VARCHAR(100),
   AUTHOR VARCHAR(40),
   DATE VARCHAR(40)
);

insert into training values
(1, 'Java', 'Krishna', '2019-09-01');
insert into training values
(2, 'JFreeCharts', 'Satish', '2019-05-01');
insert into training values
(3, 'JavaSprings', 'Amit', '2019-05-01');
insert into training values
(4, 'Android', 'Ram', '2019-03-01');
insert into training values
(5, 'Cassandra', 'Pruthvi', '2019-04-06');


CREATE TABLE backup (
   ID INT,
   TITLE VARCHAR(100),
   AUTHOR VARCHAR(40),
   DATE VARCHAR(40)
);




use plsqlstatement; - Remove it

DELIMITER //
CREATE PROCEDURE ExampleProc()
   BEGIN
      DECLARE done INT DEFAULT 0;
      DECLARE trainingID INTEGER;
      DECLARE trainingTitle, trainingAuthor, trainingDate
VARCHAR(20);
      DECLARE cur CURSOR FOR SELECT * FROM training;
```

```
      DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
      OPEN cur;
      label: LOOP
      FETCH cur INTO trainingID, trainingTitle,
trainingAuthor, trainingDate;
      INSERT INTO backup VALUES(trainingID, trainingTitle,
trainingAuthor, trainingDate);
      IF done = 1 THEN LEAVE label;
      END IF;
      END LOOP;
      CLOSE cur;
   END//
DELIMITER ;

mysql> CALL ExampleProc;
Query OK, 1 row affected (0.78 sec)

mysql> select * from backup;
+------+-------------+---------+------------+
| ID   | TITLE       | AUTHOR  | DATE       |
+------+-------------+---------+------------+
| 1    | Java        | Krishna | 2019-09-01 |
| 2    | JFreeCharts | Satish  | 2019-05-01 |
| 3    | JavaSprings | Amit    | 2019-05-01 |
| 4    | Android     | Ram     | 2019-03-01 |
| 5    | Cassandra   | Pruthvi | 2019-04-06 |
+------+-------------+---------+------------+
5 rows in set (0.08 sec)
```

MySQL View

A view is a database object that has no values. Its contents are based on the base table. It contains rows and columns similar to the real table. In MySQL, the View is a **virtual table** created by a query by joining one or more tables. It is operated similarly to the base table but does not contain any data of its own. The View and table have one main difference that the views are definitions built on top of other tables (or views).

```
1. CREATE [OR REPLACE] VIEW view_name AS
2. SELECT columns
3. FROM tables
4. [WHERE conditions];
```

Parameters:

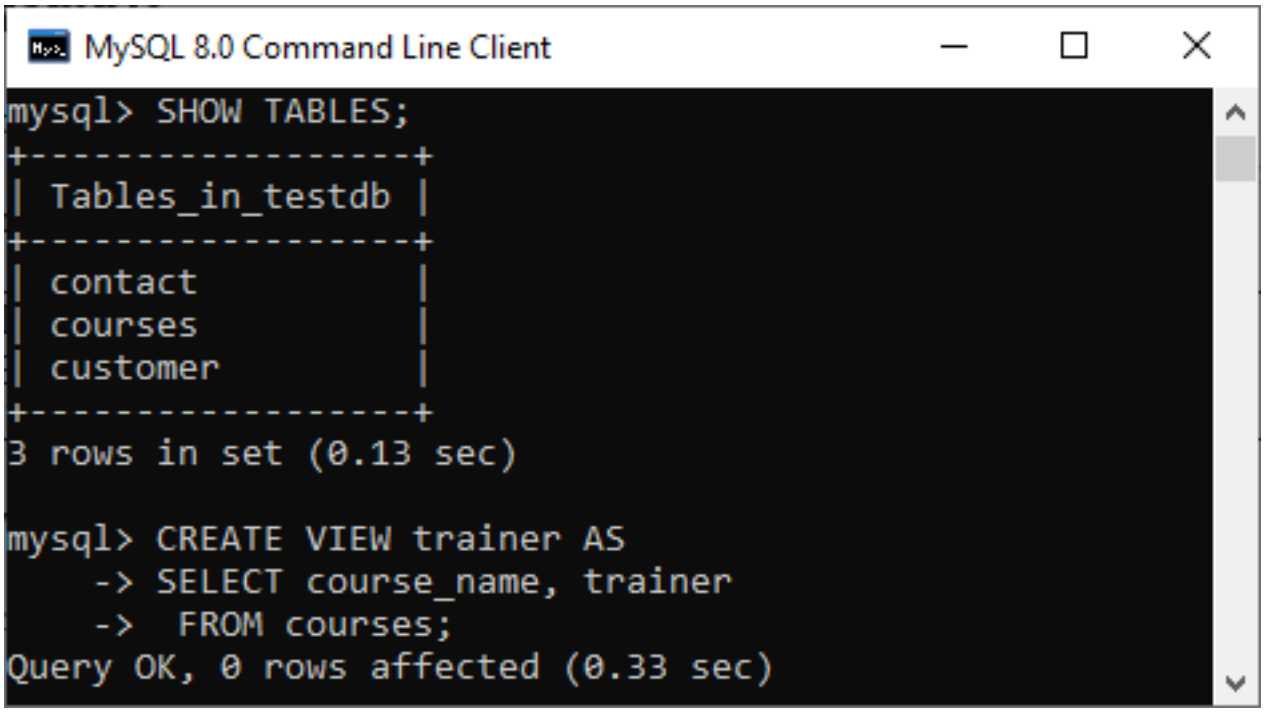
The view syntax contains the following parameters:

OR REPLACE: It is optional. It is used when a VIEW already exists. If you do not specify this clause and the VIEW already exists, the CREATE VIEW statement will return an error.

view_name: It specifies the name of the VIEW that you want to create in MySQL.

WHERE conditions: It is also optional. It specifies the conditions that must be met for the records to be included in the VIEW.

```
1. CREATE VIEW trainer AS
2. SELECT course_name, trainer
3.  FROM courses;
```



We can see the created view by using the following syntax:

```
1. SELECT * FROM view_name;
```

Let's see how it looks the created VIEW:

```
1. SELECT * FROM trainer;
```

MySQL Update VIEW

In MYSQL, the ALTER VIEW statement is used to modify or update the already created VIEW without dropping it.

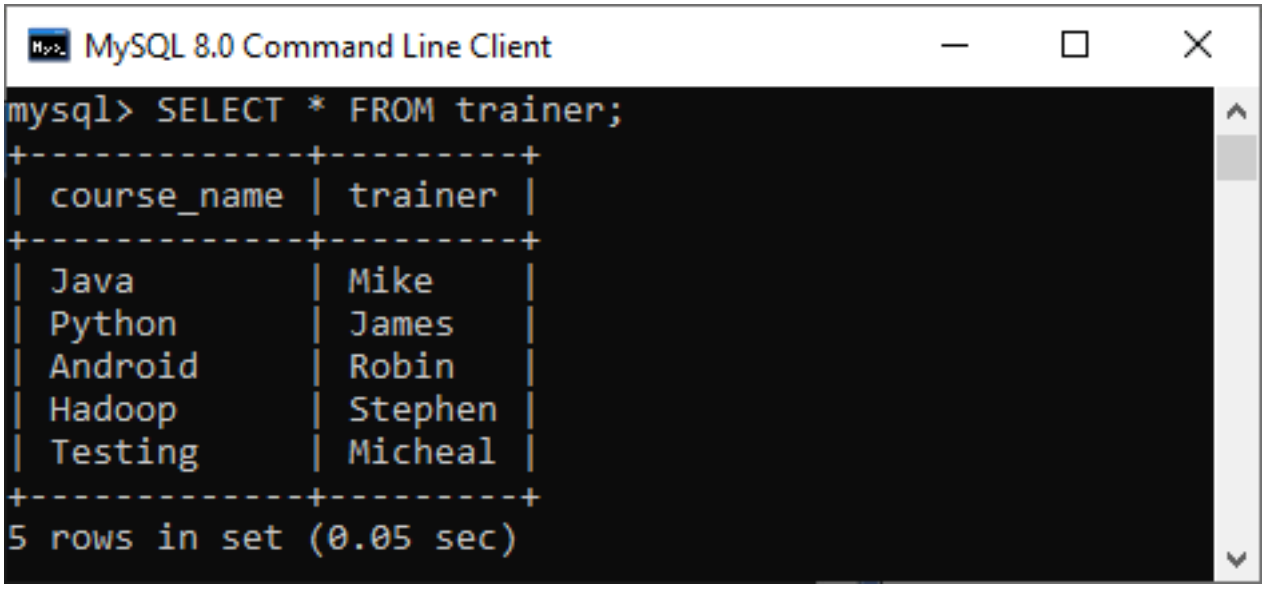
Syntax:

Following is the syntax used to update the existing view in MySQL:

```
1. ALTER VIEW view_name AS
2. SELECT columns
3. FROM table
4. WHERE conditions;
```

Example:

The following example will alter the already created VIEW name "trainer" by adding a new column.




```
1. ALTER VIEW trainer AS
2. SELECT id, course_name, trainer
3. FROM courses;
```

Once the execution of the **ALTER VIEW** statement becomes successful, MySQL will update a view and stores it in the database. We can see the altered view using the SELECT statement, as shown in the output:

MySQL Drop VIEW

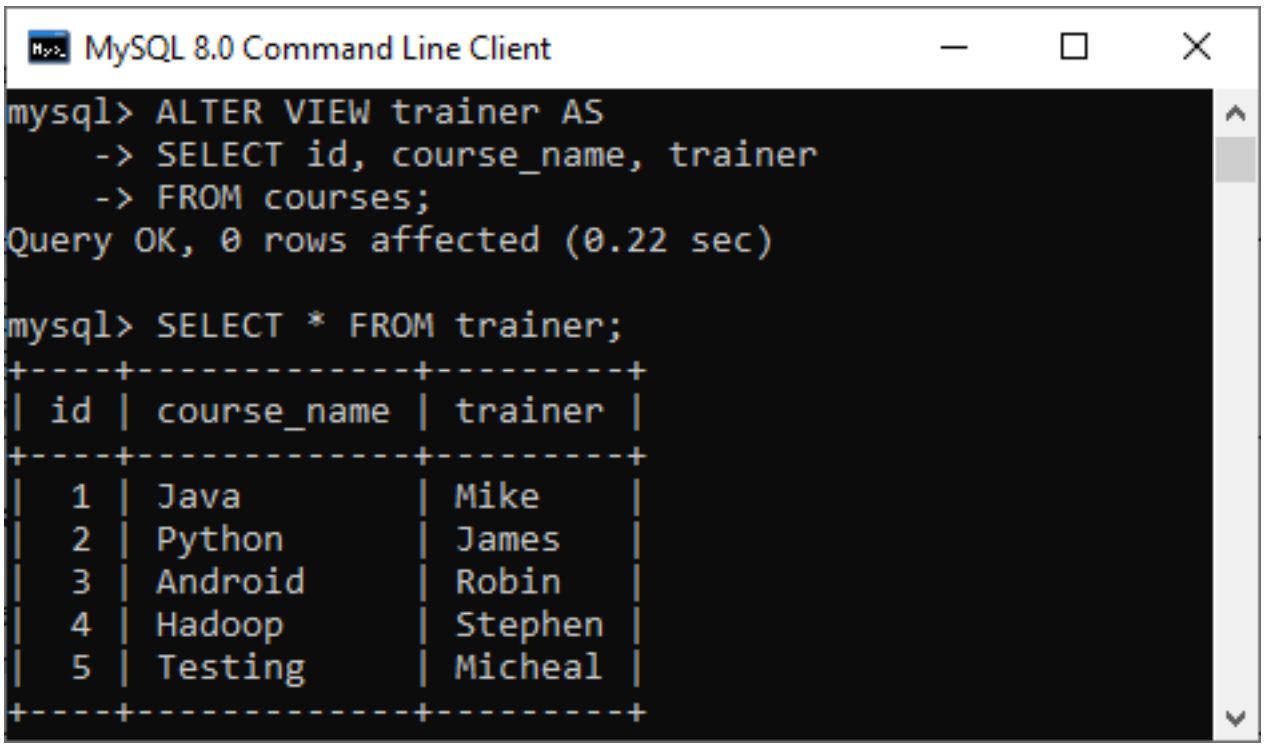
We can drop the existing VIEW by using the **DROP VIEW** statement.

Syntax:

The following is the syntax used to delete the view:

```
1. DROP VIEW [IF EXISTS] view_name;
```

Parameters:



view_name: It specifies the name of the VIEW that we want to drop.

IF EXISTS: It is optional. If we do not specify this clause and the VIEW doesn't exist, the DROP VIEW statement will return an error.

Example:

Suppose we want to delete the view "**trainer**" that we have created above. Execute the below statement:

```
1. DROP VIEW trainer;
```

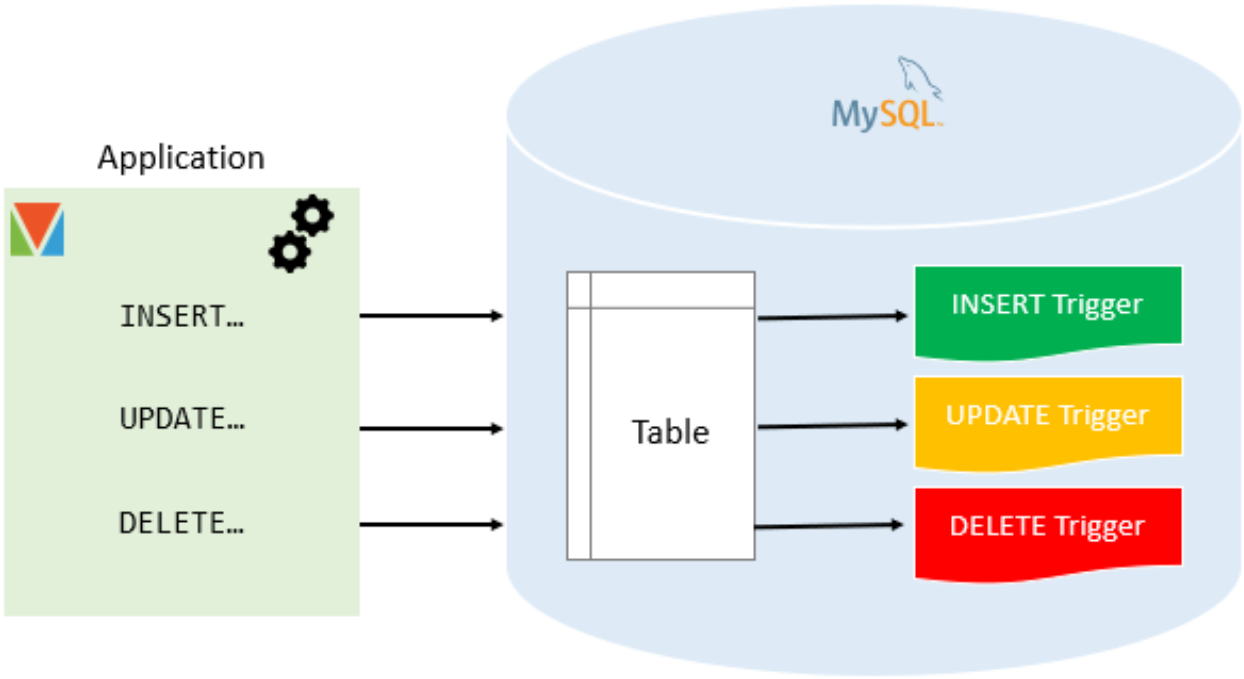
MySQL Triggers

In MySQL, 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.

MySQL supports triggers that are invoked in response to the INSERT, UPDATE or DELETE event.

The SQL standard defines two types of triggers: row-level triggers and statement-level 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.



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.

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 invisible to the client applications.
- Triggers may increase the overhead of the MySQL Server.

Managing MySQL triggers

- Create triggers – describe steps of how to create a trigger in MySQL.
- Drop triggers – show you how to drop a trigger.
- Create a BEFORE INSERT trigger – show you how to create a `BEFORE INSERT` trigger to maintain a summary table from another table.
- Create an AFTER INSERT trigger – describe how to create an `AFTER INSERT` trigger to insert data into a table after inserting data into another table.
- Create a BEFORE UPDATE trigger – learn how to create a `BEFORE UPDATE` trigger that validates data before it is updated to the table.
- Create an AFTER UPDATE trigger – show you how to create an `AFTER UPDATE` trigger to log the changes of data in a table.
- Create a BEFORE DELETE trigger – show how to create a `BEFORE DELETE` trigger.
- Create an AFTER DELETE trigger – describe how to create an `AFTER DELETE` trigger.
- Create multiple triggers for a table that have the same trigger event and time – MySQL 8.0 allows you to define multiple triggers for a table that have the same trigger event and time.
- Show triggers – list triggers in a database, table by specific patterns.

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE| DELETE }
ON table_name FOR EACH ROW
trigger_body;
```

In this syntax:

- First, specify the name of the trigger that you want to create after the `CREATE TRIGGER` keywords. Note that the trigger name must be unique within a database.
- Next, specify the trigger action time which can be either `BEFORE` or `AFTER` which indicates that the trigger is invoked before or after each row is modified.
- Then, specify the operation that activates the trigger, which can be `INSERT`, `UPDATE`, or `DELETE`.
- After that, specify the name of the table to which the trigger belongs after the `ON` keyword.

- Finally, specify the statement to execute when the trigger activates. If you want to execute multiple statements, you use the BEGIN END compound statement.

The trigger body can access the values of the column being affected by the DML statement.

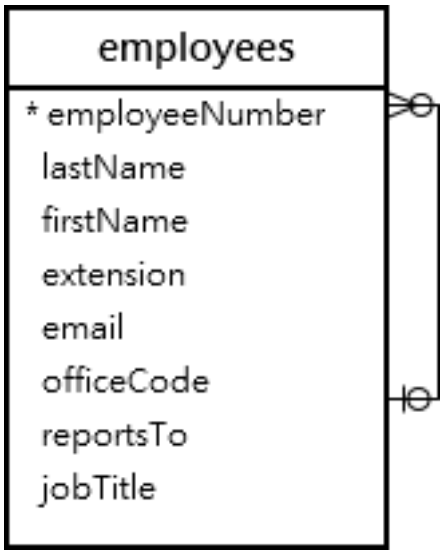
To distinguish between the value of the columns BEFORE and AFTER the DML has fired, you use the NEW and OLD modifiers.

The following table illustrates the availability of the OLD and NEW modifiers:

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

MySQL trigger examples

Let's start creating a trigger in MySQL to log the changes of the employees table.



First, create a new table named employees_audit to keep the changes to the employees table

```
CREATE TABLE employees_audit (

    id INT AUTO_INCREMENT PRIMARY KEY,
```

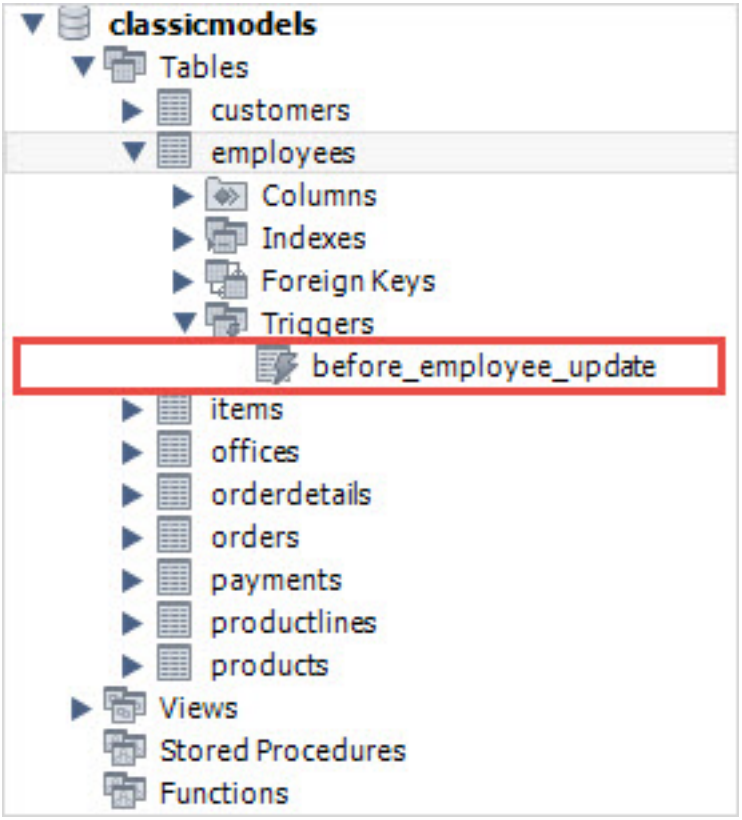
```
    employeeNumber INT NOT NULL,
    lastname VARCHAR(50) NOT NULL,
    changedat DATETIME DEFAULT NULL,
    action VARCHAR(50) DEFAULT NULL
);
CREATE TRIGGER before_employee_update
    BEFORE UPDATE ON employees
    FOR EACH ROW
 INSERT INTO employees_audit
 SET action = 'update',
     employeeNumber = OLD.employeeNumber,
     lastname = OLD.lastname,
     changedat = NOW();
```

Then, show all triggers in the current database by using the `SHOW TRIGGERS` statement:

```
SHOW TRIGGERS;
```

Trigger	Event	Table	Statement	Timing
before_employee_update	UPDATE	employees	INSERT INTO employees_audit SET action = 'update', employeeNumber = OLD.employeeNumber, lastname = OLD.lastname, changedat = NOW()	BEFORE

In addition, if you look at the schema using MySQL Workbench under the **employees > triggers**, you will see the `before_employee_update` trigger as shown in the screenshot below:



After that, update a row in the `employees` table:

```
UPDATE employees

SET

    lastName = 'Phan'

WHERE

    employeeNumber = 1056;
```

Finally, query the `employees_audit` table to check if the trigger was fired by the `UPDATE` statement:

```
SELECT * FROM employees_audit;
```

id	employeeNumber	lastname	changedat	action
1	1056	Patterson	2019-09-06 15:38:30	update

As you see clearly from the output, the trigger was automatically invoked and inserted a new row into the employees_audit table.

Introduction to MySQL DROP TRIGGER statement

The DROP TRIGGER statement deletes a trigger from the database.

Here is the basic syntax of the DROP TRIGGER statement:

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

In this syntax:

- 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 drops the trigger if the trigger exists. The IF EXISTS clause is optional.

If you drop a trigger that does not exist without using the IF EXISTS clause, MySQL issues an error. However, if you use the IF EXISTS clause, MySQL issues a NOTE instead.

The DROP TRIGGER requires the TRIGGER privilege for the table associated with the trigger.

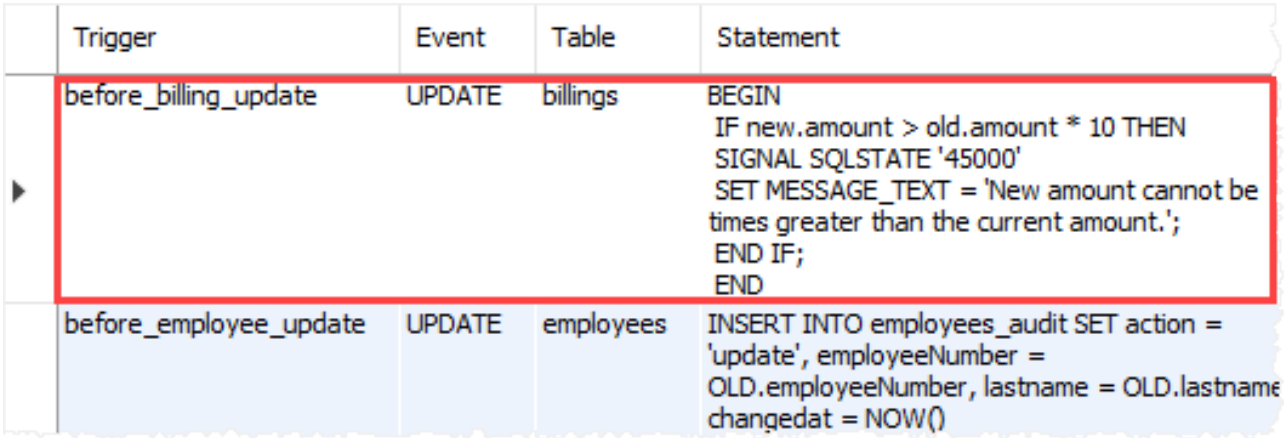
```
CREATE TABLE billings (

    billingNo INT AUTO_INCREMENT,

    customerNo INT,

    billingDate DATE,

    amount DEC(10 , 2 ),

    PRIMARY KEY (billingNo)

);



DELIMITER $$
```

```
CREATE TRIGGER before_billing_update
    BEFORE UPDATE
    ON billings FOR EACH ROW
BEGIN
    IF new.amount > old.amount * 10 THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'New amount cannot be 10 times greater than the current amount.';
    END IF;
END$$
DELIMITER ;
```

The trigger activates before any update. If the new amount is 10 times greater than the current amount, the trigger raises an error.

Third, show the triggers:

```
SHOW TRIGGERS;
```



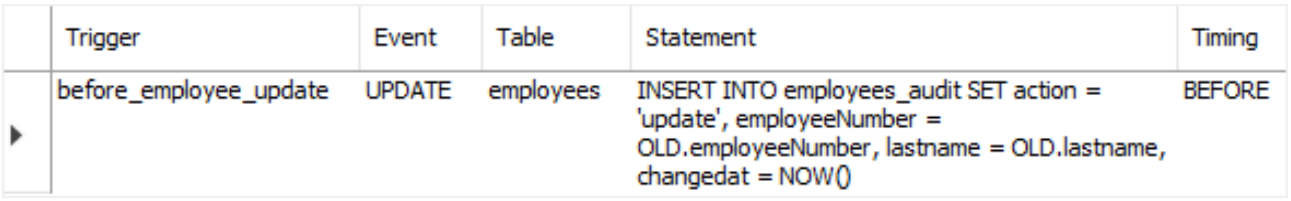
Trigger	Event	Table	Statement
before_billing_update	UPDATE	billings	BEGIN IF new.amount > old.amount * 10 THEN SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'New amount cannot be times greater than the current amount.'; END IF; END
before_employee_update	UPDATE	employees	INSERT INTO employees_audit SET action = 'update', employeeNumber = OLD.employeeNumber, lastname = OLD.lastname changedat = NOW()

Fourth, drop the `before_billing_update` trigger:

```
DROP TRIGGER before_billing_update;
```

Finally, show the triggers again to verify the removal:

SHOW TRIGGERS;



Trigger	Event	Table	Statement	Timing
before_employee_update	UPDATE	employees	INSERT INTO employees_audit SET action = 'update', employeeNumber = OLD.employeeNumber, lastname = OLD.lastname, changedat = NOW()	BEFORE

Introduction to MySQL BEFORE INSERT triggers

MySQL BEFORE INSERT triggers are automatically fired before an insert event occurs on the table.

The following illustrates the basic syntax of creating a MySQL BEFORE INSERT trigger:

```
CREATE TRIGGER trigger_name

    BEFORE INSERT

    ON table_name FOR EACH ROW

trigger_body;
```

In this syntax:

First, specify the name of the trigger that you want to create in the CREATE TRIGGER clause.

Second, use BEFORE INSERT clause to specify the time to invoke the trigger.

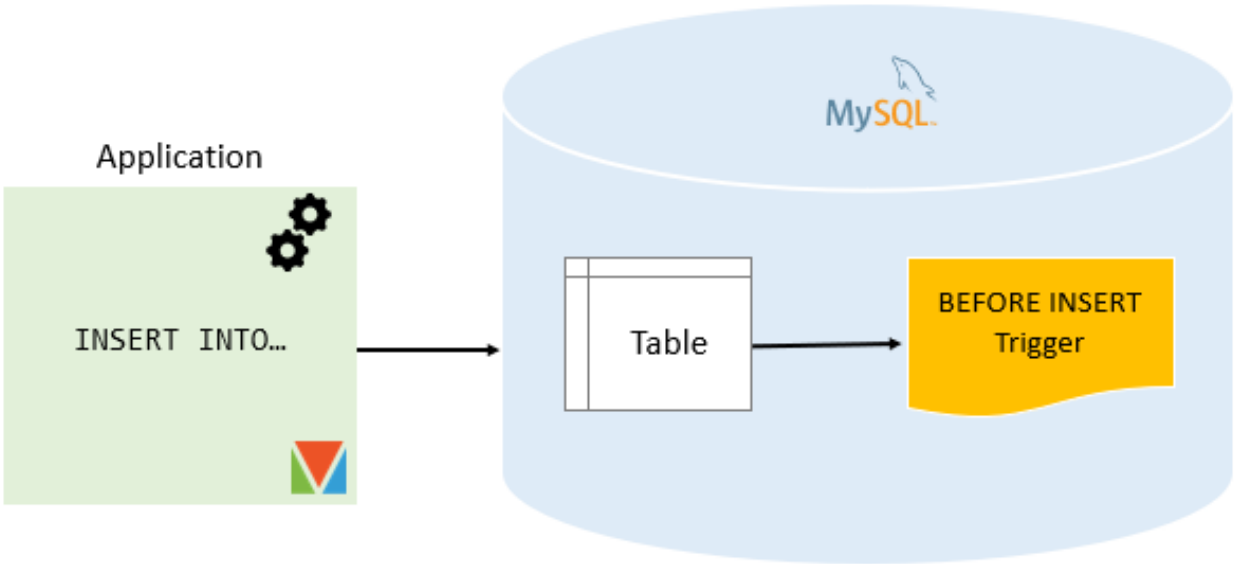
Third, specify the name of the table that the trigger is associated with after the ON keyword.

Finally, specify the trigger body which contains one or more SQL statements that execute when the trigger is invoked.

If you have multiple statements in the trigger_body, you have to use the BEGIN END block and change the default delimiter:

```
DELIMITER $$
```

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

DELIMITER ;
```




```
DROP TABLE IF EXISTS WorkCenters;

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

```
);


DROP TABLE IF EXISTS WorkCenterStats;


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

Creating BEFORE INSERT trigger example

The following trigger updates the total capacity in the `WorkCenterStats` table before a new work center is inserted into the `WorkCenter` table:

```
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 ;
```

In this trigger:

First, the name of the trigger is `before_workcenters_insert` specified in the `CREATE TRIGGER` clause:

```
CREATE TRIGGER before_workcenters_insert
```

Second, the triggering event is:

```
BEFORE INSERT
```

Third, the table that the trigger associated with is `WorkCenters` table:

```
ON WorkCenters FOR EACH ROW
```

Testing the MySQL BEFORE INSERT trigger

First, insert a new row into the WorkCenter table:

```
INSERT INTO WorkCenters(name, capacity)

VALUES('Mold Machine',100);
```

Second, query data from the WorkCenterStats table:

```
SELECT * FROM WorkCenterStats;
```

totalCapacity
100

The trigger has been invoked and inserted a new row into the WorkCenterStats table.

Third, insert a new work center:

```
INSERT INTO WorkCenters(name, capacity)

VALUES('Packing',200);
```

Finally, query data from the WorkCenterStats:

```
SELECT * FROM WorkCenterStats;
```

totalCapacity
300

The trigger has updated the total capacity from 100 to 200 as expected.

Note that to properly maintain the summary table WorkCenterStats, you should also create triggers to handle update and delete events on the WorkCenters table.

Introduction to MySQL AFTER INSERT triggers

MySQL AFTER INSERT triggers are automatically invoked after an insert event occurs on the table.

The following shows the basic syntax of creating a MySQL AFTER INSERT trigger:

```
CREATE TRIGGER trigger_name
```

```
    AFTER INSERT

    ON table_name FOR EACH ROW

        trigger_body
```

In this syntax:

First, specify the name of the trigger that you want to create after the `CREATE TRIGGER` keywords.

Second, use `AFTER INSERT` clause to specify the time to invoke the trigger.

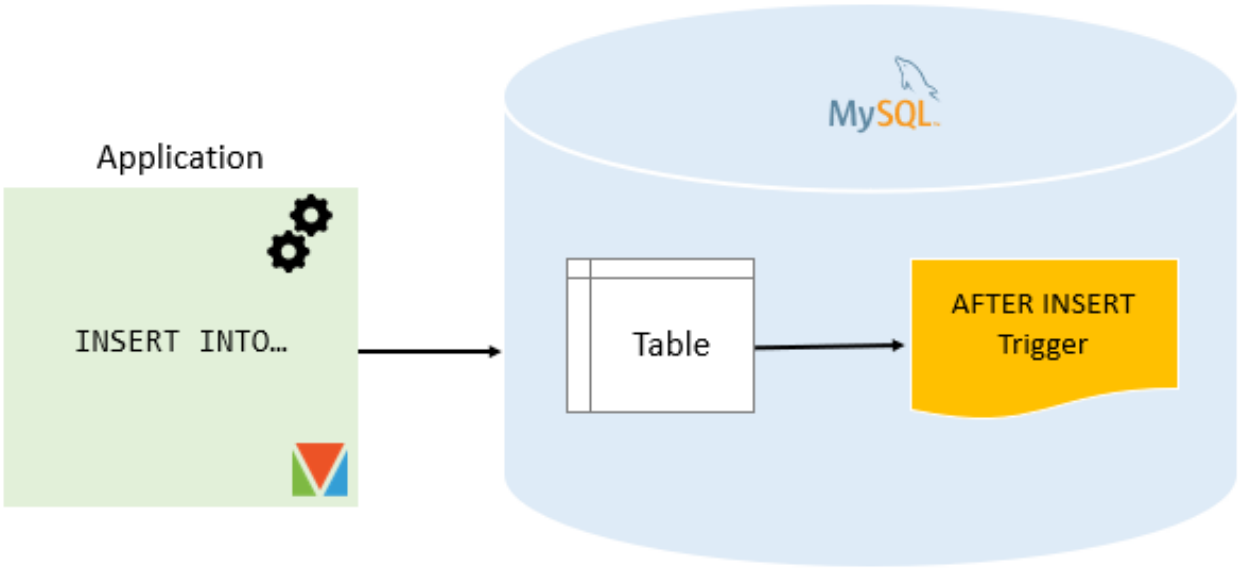
Third, specify the name of the table on which you want to create the trigger after the `ON` keyword.

Finally, specify the trigger body which consists of one or more statements that execute when the trigger is invoked.

In case the trigger body has multiple statements, you need to use the `BEGIN END` block and change the default delimiter:

```
DELIMITER $$


CREATE TRIGGER trigger_name

    AFTER INSERT

    ON table_name FOR EACH ROW

BEGIN

    -- statements

END$$


DELIMITER ;
```

In an AFTER INSERT trigger, you can access the NEW values but you cannot change them. Also, you cannot access the OLD values because there is no OLD on INSERT triggers.



MySQL AFTER INSERT trigger example

Consider the following AFTER INSERT trigger example.

Setting up a sample table

First, create a new table called members:

```
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)
);
```

Second, create another table called reminders that stores reminder messages to members.

```
DROP TABLE IF EXISTS reminders;


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

Creating AFTER INSERT trigger example

The following statement creates an `AFTER INSERT` trigger that inserts a reminder into the `reminders` table if the birth date of the member is `NULL`.

```
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 //
```

In this trigger:

First, the name of the trigger is after_members_insert specified in the CREATE TRIGGER clause:

CREATE TRIGGER after_members_insert

Second, the triggering event is:

AFTER INSERT

Third, the table that the trigger associated with is members table:

ON members FOR EACH ROW

Finally, inside the trigger body, insert a new row into the reminder table if the birth date of the member is NULL.

Testing the MySQL AFTER INSERT trigger

First, insert two rows into the members table:

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

Second, query data from the members table:

```
SELECT * FROM members;
```

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

Third, query data from reminders table:

```
SELECT * FROM reminders;
```

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

We inserted two rows into the members table. However, only the first row that has a birth date value NULL, therefore, the trigger inserted only one row into the reminders table.

Introduction to MySQL BEFORE UPDATE triggers

MySQL BEFORE UPDATE triggers are invoked automatically before an update event occurs on the table associated with the triggers.

Here is the syntax of creating a MySQL BEFORE UPDATE trigger:

```
CREATE TRIGGER trigger_name

BEFORE UPDATE

ON table_name FOR EACH ROW

trigger_body
```

In this syntax:

First, specify the name of the trigger that you want to create after the CREATE TRIGGER keywords.

Second, use BEFORE UPDATE clause to specify the time to invoke the trigger.

Third, specify the name of the table to which the trigger belongs after the ON keyword.

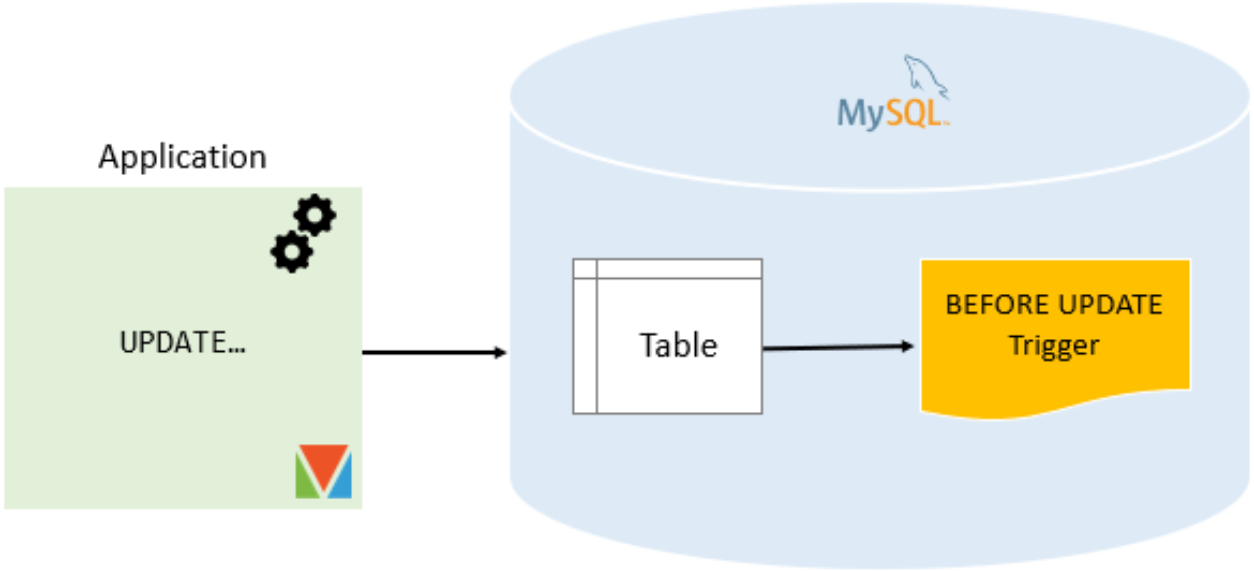
Finally, specify the trigger body which contains one or more statements.

If you have more than one statement in the trigger_body, you need to use the BEGIN END block. In addition, you need to change the default delimiter as follows:

```
DELIMITER $$

CREATE TRIGGER trigger_name
```

```
    BEFORE UPDATE
    ON table_name FOR EACH ROW
BEGIN
    -- statements
END$$

DELIMITER ;
```

In a `BEFORE UPDATE` trigger, you can update the `NEW` values but cannot update



MySQL BEFORE UPDATE trigger example

Let's look at an example of using a `BEFORE UPDATE` trigger.

Setting up a sample table

First, create a new table called `sales` to store sales volumes:

```
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)
);
```

Second, insert some rows into the sales table:

```
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);
```

Third, query data from the sales table to verify the insert:

```
SELECT * FROM sales;
```

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

Creating BEFORE UPDATE trigger example

The following statement creates a BEFORE UPDATE trigger on the sales table

```
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 ;
```

The trigger is automatically fired before an update event occurs for each row in the sales table.

If you update the value in the quantity column to a new value that is 3 times greater than the current value, the trigger raises an error and stops the update.

Let's examine the trigger in details:

First, the name of the trigger is before_sales_update specified in the CREATE TRIGGER clause:

```
CREATE TRIGGER before_sales_update
```

Second, the triggering event is:

```
BEFORE UPDATE
```

Third, the table that the trigger associated with is sales:

```
ON sales FOR EACH ROW
```

Fourth, declare a variable and set its value to an error message. Note that, in the BEFORE TRIGGER, you can access both old and new values of the columns via OLD and NEW modifiers.

```
DECLARE errorMessage VARCHAR(255);

SET errorMessage = CONCAT('The new quantity ',

                    NEW.quantity,

                    ' cannot be 3 times greater than the current quantity ',

                    OLD.quantity);
```

Note that we use the CONCAT() function to form the error message.

Finally, use the IF-THEN statement to check if the new value is 3 times greater than old value, then raise an error by using the SIGNAL statement

```
IF new.quantity > old.quantity * 3 THEN

    SIGNAL SQLSTATE '45000'

        SET MESSAGE_TEXT = errorMessage;

END IF;
```

Testing the MySQL BEFORE UPDATE trigger

First, update the quantity of the row with id 1 to 150:

```
UPDATE sales

SET quantity = 150

WHERE id = 1;
```

It worked because the new quantity does not violate the rule.

Second, query data from the sales table to verify the update:

```
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

Third, update the quantity of the row with id 1 to 500:

```
UPDATE sales

SET quantity = 500

WHERE id = 1;
```

MySQL issued this error:

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

n this case, the trigger found that the new quantity caused a violation and raised an error.

Finally, use the SHOW ERRORS to display the error:

```
SHOW ERRORS;
```

Level	Code	Message
Error	1644	The new quantity 500 cannot be 3 times greater than the current quantity 150