1. An overview of relational database design using MySQL/ MariaDB/ PostgreSQL etc. (Apply the following basic queries on an Employee/ Student database etc.)
   1. DDL Commands
   2. DML Commands
   3. Imposing restrictions on database (DCL & TCL Commands)
   4. Accessing database (SELECT, Filtering using WHERE, HAVING, GROUP BY, ORDER BY Clauses, Subquery and View)
   5. Optimizing databases (Join, Aggregate & Set operations, Other operators like arithmetic, logical, special etc.)

LABExercize

OBJECTIVES

To understand DDL commands.

1.Consider the database for an organisation**.** Write the queries for the following ( 6th to 10th march)

create the database

select the current database (iii)Create the following tables.

**employee** (emp\_no,emp\_name,DOB, address, doj, mobile\_no, dept\_no, salary).

**department** (dept\_no, dept\_name, location).

1. Include necessary constraints.
2. List all the tables in the current database
3. Display the structure of the employee table
4. Add a new column Designation to the employee table
5. Drop the column location from Dept table
6. Drop the tables
7. Delete the database

**DATA MANIPULATION LANGUAGE(DML)**

2.Consider the database for an organisation. Write the queries for the following

1. Add 5 rows in the employee and dept tables
2. Display all the records from the above tables
3. Display the empno and name of all the employees from department no2
4. Display empno,name,designation,dept no and salary in the descending order of salary
5. Display the empno and name of all employees whose salary is between 2000 and 5000
6. Display all designations without duplicate values.
7. Display the dept name and total salary of employees of each department.
8. Change the salary of employees to 25000 whose designation is ‘Typist’
9. Change the memployeobile no of e named ‘john’
10. Delete all employees whose salaries are equal to Rs.7000
11. Select the department that has total salary paid for its employees more than 25000

**SUB QUERIES AND JOIN 14th march**

Consider the database for the organization and Write the queries for the following

1. display the empno, name, and salaries for employees whose average salary is higher than the average salary of the organization
2. Display the details of employees whose salary is equal to the minimum salary of organisation.
3. Display all the employees whose designation is same as that of ‘Arun’
4. display the empno and name of employees who earn more than any Employee in dept 1.
5. Display the empno,name , departments that the departments are same in both the emp and dept
6. Display the employee details by implementing left inner join
7. Display employee details by implementing a right outer join

SELF LEARNING EXERCIZE EXAMPLE 16TH TO 21ST MARCH

1. PL/SQL Programs (Trigger, Cursor, Stored Procedures and Functions)

23Rrd MARCH to 28th march

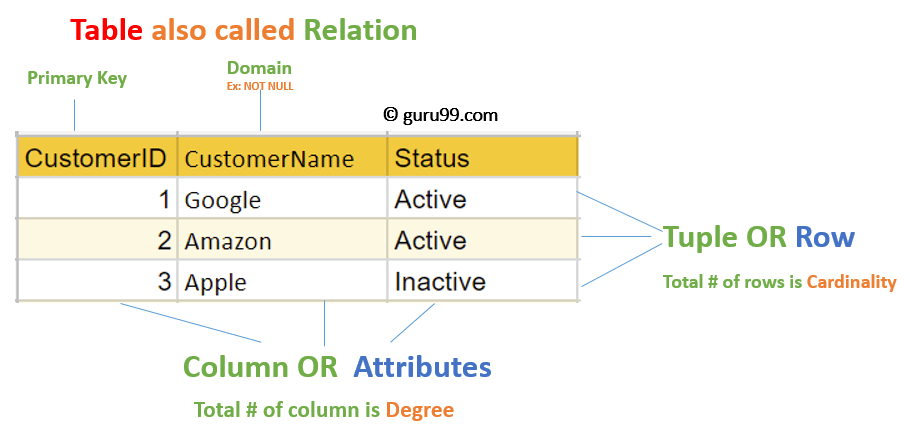
1. Introduction to NoSQL Databases.
   1. Installation and configuration of any one of the NoSQL databases - MongoDB/ Cassandra/ HBase/ CouchDB/ Amazon DynamoDB/ Redis/ Neo4j etc.
2. Designing Databases using NoSQL
3. Query Processing
   1. Performing CRUD operations
   2. Retrieving Data from a NoSQL database
   3. Usage of aggregate functions, regular expressions etc.
4. NoSQL Administration
   1. Security, Monitoring & Backup
   2. Create Users and Roles
5. NoSQL shell commands
   1. Perform Sharding, Replication (Master-Slave/ Master-Less/ Peer-to-Peer Architectures), Clustering, Partitioning, Indexing (Corresponding to the selected NoSQL Database)
6. Deployment
   1. Local Deployment
      1. NoSQL and Front-End: PHP/Java/Python (MongoDB/ Cassandra etc.)
   2. Cloud Deployment
      1. NoSQL and Cloud: Amazon DynamoDB/ Google Bigtable/ Azure Cosmos DB
      2. Familiarization of Atlas/ DataStax corresponding to the selected NoSQL Database
7. ***Micro project:*** *Students can be given a group micro project, so that they learn to work in a team environment.*

**1)EXPERIMENT No.1**

* **Relational Model (RM)** represents the database as a collection of relations. A relation is nothing but a table of values. Every row in the table represents a collection of related data values. These rows in the table denote a real-world entity or relationship.

**Relational Model Terms**

* **Attribute:** Each column in a Table. Attributes are the properties which define a relation. e.g., Student\_Rollno, NAME,etc.
* **Tables** – In the Relational model the, relations are saved in the table format. It is stored along with its entities. A table has two properties rows and columns. Rows represent records and columns represent attributes.
* **Tuple** – It is nothing but a single row of a table, which contains a single record.
* **Relation Schema:** A relation schema represents the name of the relation with its attributes.
* **Degree:** The total number of attributes which in the relation is called the degree of the relation.
* **Cardinality:**Total number of rows present in the Table.
* The table name and column names are helpful to interpret the meaning of values in each row. The data are represented as a set of relations. In the relational model, data are stored as tables. However, the physical storage of the data is independent of the way the data are logically organized.
* STRUCTURED QUERY LANGUAGE/SQL is the standard language for database management.
* **SQL** is a database language designed for the retrieval and management of data in a relational database.
* All the RDBMS systems like MySQL, MS Access, Oracle, Sybase, Postgres, and SQL Server use SQL as their standard database language. SQL programming language uses various commands for different operations. We will learn about the like DCL, TCL, DQL, DDL and DML commands in SQL with examples.



MySQL Relational Database is an assemblage of relational data that is structured or organized in the form of tables, columns, and rows, where tables represent the objects, columns represent the fields and rows represent the records.

**Relational Integrity Constraints**

Relational Integrity constraints in DBMS are referred to conditions which must be present for a valid relation. These Relational constraints in DBMS are derived from the rules in the mini-world that the database represents.

There are many types of Integrity Constraints in DBMS. Constraints on the Relational database management system is mostly divided into three main categories are:

1. Domain Constraints
2. Key Constraints
3. Referential Integrity Constraints

**Domain Constraints**

Domain constraints can be violated if an attribute value is not appearing in the corresponding domain or it is not of the appropriate data type.

Domain constraints specify that within each tuple, and the value of each attribute must be unique. This is specified as data types which include standard data types integers, real numbers, characters, Booleans, variable length strings, etc.

**Example:**

Create DOMAIN CustomerName

CHECK (value not NULL)

The example shown demonstrates creating a domain constraint such that CustomerName is not NULL

**Key Constraints**

An attribute that can uniquely identify a tuple in a relation is called the key of the table. The value of the attribute for different tuples in the relation has to be unique.

**Example:**

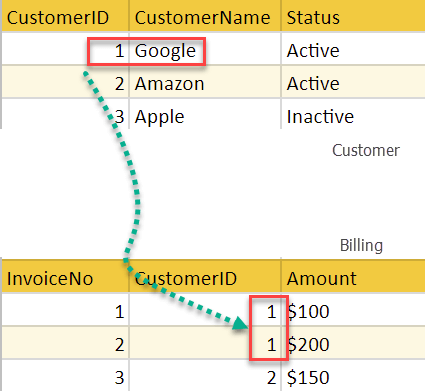
In the given table, CustomerID is a key attribute of Customer Table. It is most likely to have a single key for one customer, CustomerID =1 is only for the CustomerName =" Google".

| CustomerID | CustomerName | Status |
| --- | --- | --- |
| 1 | Google | Active |
| 2 | Amazon | Active |
| 3 | Apple | Inactive |
|  |  |  |

**Referential Integrity Constraints**

Referential Integrity constraints in DBMS are based on the concept of Foreign Keys. A foreign key is an important attribute of a relation which should be referred to in other relationships. Referential integrity constraint state happens where relation refers to a key attribute of a different or same relation. However, that key element must exist in the table.

**Example:**



In the above example, we have 2 relations, Customer and Billing.

Tuple for CustomerID =1 is referenced twice in the relation Billing. So we know CustomerName=Google has billing amount $300

**Operations in Relational Model**

Four basic update operations performed on relational database model are

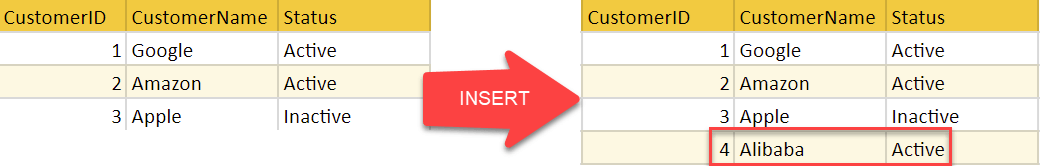
Insert, update, delete and select.

* Insert is used to insert data into the relation
* Delete is used to delete tuples from the table.
* Modify allows you to change the values of some attributes in existing tuples.
* Select allows you to choose a specific range of data.

Whenever one of these operations are applied, integrity constraints specified on the relational database schema must never be violated.

**Insert Operation**

The insert operation gives values of the attribute for a new tuple which should be inserted into a relation.



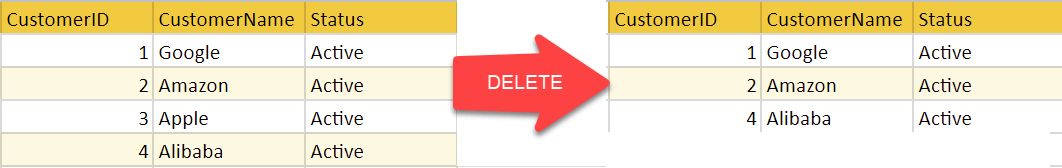
**Update Operation**

You can see that in the below-given relation table CustomerName= 'Apple' is updated from Inactive to Active.



**Delete Operation**

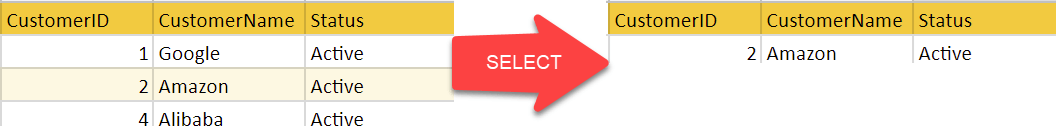
To specify deletion, a condition on the attributes of the relation selects the tuple to be deleted.



In the above-given example, CustomerName= "Apple" is deleted from the table.

The Delete operation could violate referential integrity if the tuple which is deleted is referenced by foreign keys from other tuples in the same database.

**Select Operation**

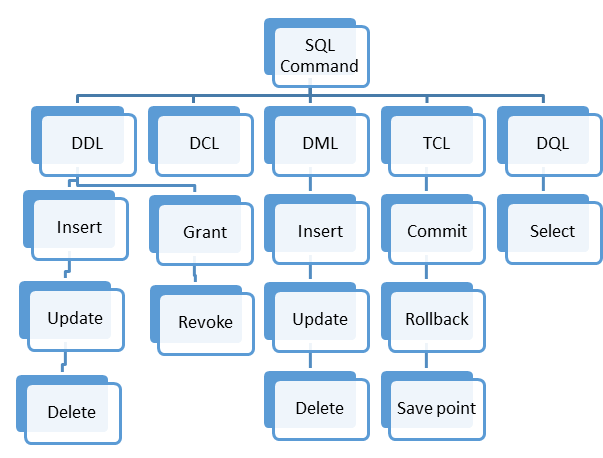


In the above-given example, CustomerName="Amazon" is selected

**Types of SQL**

Here are five types of widely used SQL queries.

* Data Definition Language (DDL)
* Data Manipulation Language (DML)
* Data Control Language(DCL)
* Transaction Control Language(TCL)
* Data Query Language (DQL)



Types of SQL

EXPERIMENT NO:1

#### DDL (Data Definition Language) :

Data Definition Language is used to define the database structure or schema. DDL is also used to specify additional properties of the data. The storage structure and access methods used by the database system by a set of statements in a special type of DDL called a data storage and definition language. These statements define the implementation details of the database schema, which are usually hidden from the users. The data values stored in the database must satisfy certain consistency constraints.  
For example, suppose the university requires that the account balance of a department must never be negative. The DDL provides facilities to specify such constraints. The database system checks these constraints every time the database is updated. In general, a constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to the test. Thus, the database system implements integrity constraints that can be tested with minimal overhead.

1. **Domain Constraints :** A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as the constraints on the values that it can take.
2. **Referential Integrity :** There are cases where we wish to ensure that a value appears in one relation for a given set of attributes also appear in a certain set of attributes in another relation i.e. Referential Integrity. For example, the department listed for each course must be one that actually exists.
3. **Assertions :** An assertion is any condition that the database must always satisfy. Domain constraints and Integrity constraints are special form of assertions.
4. **Authorization :** We may want to differentiate among the users as far as the type of access they are permitted on various data values in database. These differentiation are expressed in terms of Authorization. The most common being :  
   *read authorization* – which allows reading but not modification of data ;  
   *insert authorization* – which allow insertion of new data but not modification of existing data  
   *update authorization* – which allows modification, but not deletion.

**Some Commands:**

CREATE : to create objects in database

ALTER : alters the structure of database

DROP : delete objects from database

RENAME : rename an objects

Following SQL DDL-statement defines the department table :

create table department

(dept\_name char(20),

building char(15),

budget numeric(12,2));

Execution of the above DDL statement creates the department table with three columns – dept\_name, building, and budget; each of which has a specific datatype associated with it.

**Data Definition Language helps you to define the database structure or schema. Let's learn about DDL commands with syntax.**

Five types of DDL commands in SQL are:

**CREATE**

CREATE statements is used to define the database structure schema:

**Syntax:**

CREATE TABLE TABLE\_NAME (COLUMN\_NAME DATATYPES[,....]);

**For example**:

Create database university;

Create table students;

Create view for\_students;

**DROP**

Drops commands remove tables and databases from RDBMS.

Syntax

DROP TABLE ;

**For example:**

Drop object\_type object\_name;

Drop database university;

Drop table student;

**ALTER**

Alters command allows you to alter the structure of the database.

**Syntax:**

To add a new column in the table

ALTER TABLE table\_name ADD column\_name COLUMN-definition;

To modify an existing column in the table:

ALTER TABLE MODIFY(COLUMN DEFINITION....);

**For example:**

Alter table guru99 add subject varchar;

**TRUNCATE:**

This command used to delete all the rows from the table and free the space containing the table.

**Syntax:**

TRUNCATE TABLE table\_name;

**Example:**

TRUNCATE table students;

**EXPERIMENT NO:1.b**

#### DML (Data Manipulation Language) :

DML statements are used for managing data with in schema objects.  
DML are of two types –

1. **Procedural DMLs** : require a user to specify what data are needed and how to get those data.
2. **Declerative DMLs** (also referred as **Non-procedural DMLs**) : require a user to specify what data are needed without specifying how to get those data.

Declarative DMLs are usually easier to learn and use than procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data.

**Some Commands :**

SELECT: retrieve data from the database

INSERT: insert data into a table

UPDATE: update existing data within a table

DELETE: deletes all records from a table, space for the records remain

Example of SQL query that finds the names of all instructors in the History department :

select instructor.name

from instructor

where instructor.dept\_name = 'History';

The query specifies that those rows from the table instructor where the dept\_name is History must be retrieved and the name attributes of these rows must be displayed.

#### TCL (Transaction Control Language) :

Transaction Control Language commands are used to manage transactions in the database. These are used to manage the changes made by DML-statements. It also allows statements to be grouped together into logical transactions.

Examples of TCL commands –

COMMIT: Commit command is used to permanently save any transaction

into the database.

ROLLBACK: This command restores the database to last committed state.

It is also used with savepoint command to jump to a savepoint

in a transaction.

SAVEPOINT: Savepoint command is used to temporarily save a transaction so

that you can rollback to that point whenever necessary.

#### DCL (Data Control Language) :

A Data Control Language is a syntax similar to a computer programming language used to control access to data stored in a database (Authorization). In particular, it is a component of Structured Query Language (SQL).

Examples of DCL commands :

GRANT: allow specified users to perform specified tasks.

REVOKE: cancel previously granted or denied permissions.

The operations for which privileges may be granted to or revoked from a user or role apply to both the Data definition language (DDL) and the Data manipulation language (DML), and may include CONNECT, SELECT, INSERT, UPDATE, DELETE, EXECUTE and USAGE.

There are three basic constructs which allow database program and user to enter data and information are:

Here are some important DML commands in SQL:

* INSERT
* UPDATE
* DELETE

**INSERT:**

This is a statement is a SQL query. This command is used to insert data into the row of a table.

**Syntax:**

INSERT INTO TABLE\_NAME (col1, col2, col3,.... col N)

VALUES (value1, value2, value3, .... valueN);

Or

INSERT INTO TABLE\_NAME

VALUES (value1, value2, value3, .... valueN);

**For example:**

INSERT INTO students (RollNo, FIrstName, LastName) VALUES ('60', 'Tom', Erichsen');

**UPDATE:**

This command is used to update or modify the value of a column in the table.

**Syntax:**

UPDATE table\_name SET [column\_name1= value1,...column\_nameN = valueN] [WHERE CONDITION]

**For example:**

UPDATE students

SET FirstName = 'Jhon', LastName= 'Wick'

WHERE StudID = 3;

**DELETE:**

This command is used to remove one or more rows from a table.

**Syntax:**

DELETE FROM table\_name [WHERE condition];

**For example:**

DELETE FROM students

WHERE FirstName = 'Jhon';

**What is DCL?**

DCL (Data Control Language) includes commands like GRANT and REVOKE, which are useful to give "rights & permissions." Other permission controls parameters of the database system.

**Examples of DCL commands:**

Commands that come under DCL:

* Grant
* Revoke

**Grant:**

This command is use to give user access privileges to a database.

**Syntax:**

GRANT SELECT, UPDATE ON MY\_TABLE TO SOME\_USER, ANOTHER\_USER;

**For example:**

GRANT SELECT ON Users TO'Tom'@'localhost;

**Revoke:**

It is useful to back permissions from the user.

**Syntax:**

REVOKE privilege\_nameON object\_nameFROM {user\_name |PUBLIC |role\_name}

**For example:**

REVOKE SELECT, UPDATE ON student FROM BCA, MCA;

**Experiment no:1.c**

Transactions group a set of tasks into a single execution unit.

Each transaction begins with a specific task and ends when all the tasks in the group successfully complete. If any of the tasks fail, the transaction fails. Therefore, a transaction has only two results: **success** or **failure**.

Incomplete steps result in the failure of the transaction. A database transaction, by definition, must be atomic, consistent, isolated and durable. These are popularly known as   
[ACID](https://en.wikipedia.org/wiki/ACID) properties. 

**How to implement Transactions using SQL?**

Following commands are used to control transactions. It is important to note that these statements cannot be used while creating tables and are only used with the DML Commands such as – INSERT, UPDATE and DELETE. 

**1. BEGIN TRANSACTION:**It indicates the start point of an explicit or local transaction.

**Syntax:**

BEGIN TRANSACTION transaction\_name ;

**2. SET TRANSACTION:** Places a name on a transaction.

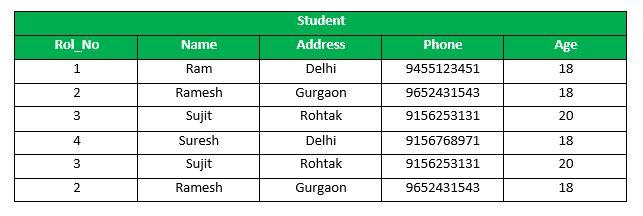
**Syntax:**

SET TRANSACTION [ READ WRITE | READ ONLY ];

**3. COMMIT:** If everything is in order with all statements within a single transaction, all changes are recorded together in the database is called **committed**. The COMMIT command saves all the transactions to the database since the last COMMIT or ROLLBACK command.   
 **Syntax:** 

COMMIT;

**Example: Sample table 1**



Following is an example which would delete those records from the table which have age = 20 and then COMMIT the changes in the database.   
**Queries:** 

DELETE FROM Student WHERE AGE = 20;

COMMIT;

**Output:**   
Thus, two rows from the table would be deleted and the SELECT statement would look like,



**4. ROLLBACK:**If any error occurs with any of the SQL grouped statements, all changes need to be aborted. The process of reversing changes is called **rollback**. This command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.   
**Syntax:** 

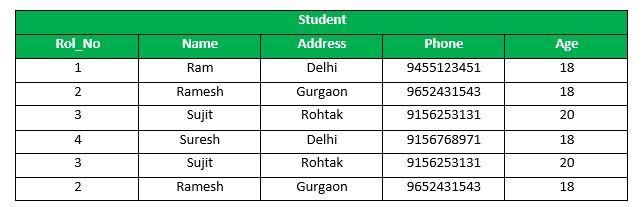
ROLLBACK;

**Example:**   
From the above example **Sample table1**,   
Delete those records from the table which have age = 20 and then ROLLBACK the changes in the database.   
**Queries:** 

DELETE FROM Student WHERE AGE = 20;

ROLLBACK;

**Output:**



**5. SAVEPOINT:** creates points within the groups of transactions in which to ROLLBACK.   
A SAVEPOINT is a point in a transaction in which you can roll the transaction back to a certain point without rolling back the entire transaction. 

**Syntax for Savepoint command:** 

SAVEPOINT SAVEPOINT\_NAME;

This command is used only in the creation of SAVEPOINT among all the transactions.   
In general ROLLBACK is used to undo a group of transactions.   
**Syntax for rolling back to Savepoint command:** 

ROLLBACK TO SAVEPOINT\_NAME;

you can ROLLBACK to any SAVEPOINT at any time to return the appropriate data to its original state.   
**Example:**   
From the above example **Sample table1**,   
Delete those records from the table which have age = 20 and then ROLLBACK the changes in the database by keeping Savepoints.   
**Queries:** 

SAVEPOINT SP1;

//Savepoint created.

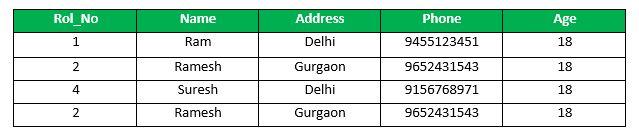
DELETE FROM Student WHERE AGE = 20;

//deleted

SAVEPOINT SP2;

//Savepoint created.

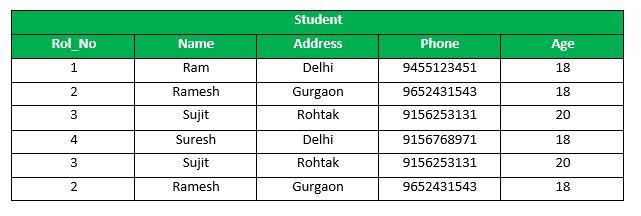
Here SP1 is first SAVEPOINT created before deletion.In this example one deletion have taken place.   
After deletion again SAVEPOINT SP2 is created.   
**Output:**



Deletion have been taken place, let us assume that you have changed your mind and decided to ROLLBACK to the SAVEPOINT that you identified as SP1 which is before deletion.   
deletion is undone by this statement , 

ROLLBACK TO SP1;

//Rollback completed.



**6. RELEASE SAVEPOINT:-** This command is used to remove a SAVEPOINT that you have created.   
**Syntax:** 

RELEASE SAVEPOINT SAVEPOINT\_NAME

Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the last SAVEPOINT.

It is used to initiate a database transaction and used to specify characteristics of the transaction that follows.

## Optimizing databases (Join, Aggregate & Set operations, Other operators like arithmetic, logical, special etc.)

### What are JOINS?

### Joins help retrieving data from two or more database tables.

### The tables are mutually related using primary and foreign keys.

### For sake of simplicity and ease of understanding , we will be using a new Database to practice sample.  As shown below

| id | first\_name | last\_name | movie\_id |
| --- | --- | --- | --- |
| 1 | Adam | Smith | 1 |
| 2 | Ravi | Kumar | 2 |
| 3 | Susan | Davidson | 5 |
| 4 | Jenny | Adrianna | 8 |
| 6 | Lee | Pong | 10 |
| id | Title | | Category |
| 1 | ASSASSIN'S CREED: EMBERS | | Animations |
| 2 | Real Steel(2012) | | Animations |
| 3 | Alvin and the Chipmunks | | Animations |
| 4 | The Adventures of Tin Tin | | Animations |
| 5 | Safe (2012) | | Action |
| 6 | Safe House(2012) | | Action |
| 7 | GIA | | 18+ |
| 8 | Deadline 2009 | | 18+ |
| 9 | The Dirty Picture | | 18+ |
| 10 |  | |  |
| id | Title |  | id | first\_name | last\_name | movie\_id |
| 1 | ASSASSIN'S CREED: EMBERS | Animations | 1 | Adam | Smith | 1 |
| 1 | ASSASSIN'S CREED: EMBERS | Animations | 2 | Ravi | Kumar | 2 |
| 1 | ASSASSIN'S CREED: EMBERS | Animations | 3 | Susan | Davidson | 5 |
| 1 | ASSASSIN'S CREED: EMBERS | Animations | 4 | Jenny | Adrianna | 8 |
| 1 | ASSASSIN'S CREED: EMBERS | Animations | 6 | Lee | Pong | 10 |
| 2 | Real Steel(2012) | Animations | 1 | Adam | Smith | 1 |
| 2 | Real Steel(2012) | Animations | 2 | Ravi | Kumar | 2 |
| 2 | Real Steel(2012) | Animations | 3 | Susan | Davidson | 5 |
| 2 | Real Steel(2012) | Animations | 4 | Jenny | Adrianna | 8 |
| 2 | Real Steel(2012) | Animations | 6 | Lee | Pong | 10 |
| 3 | Alvin and the Chipmunks | Animations | 1 | Adam | Smith | 1 |
| 3 | Alvin and the Chipmunks | Animations | 2 | Ravi | Kumar | 2 |
| 3 | Alvin and the Chipmunks | Animations | 3 | Susan | Davidson | 5 |
| 3 | Alvin and the Chipmunks | Animations | 4 | Jenny | Adrianna | 8 |
| 3 | Alvin and the Chipmunks | Animations | 6 | Lee | Pong | 10 |
| 4 | The Adventures of Tin Tin | Animations | 1 | Adam | Smith | 1 |
| 4 | The Adventures of Tin Tin | Animations | 2 | Ravi | Kumar | 2 |
| 4 | The Adventures of Tin Tin | Animations | 3 | Susan | Davidson | 5 |
| 4 | The Adventures of Tin Tin | Animations | 4 | Jenny | Adrianna | 8 |
| 4 | The Adventures of Tin Tin | Animations | 6 | Lee | Pong | 10 |
| 5 | Safe (2012) | Action | 1 | Adam | Smith | 1 |
| 5 | Safe (2012) | Action | 2 | Ravi | Kumar | 2 |
| 5 | Safe (2012) | Action | 3 | Susan | Davidson | 5 |
| 5 | Safe (2012) | Action | 4 | Jenny | Adrianna | 8 |
| 5 | Safe (2012) | Action | 6 | Lee | Pong | 10 |
| 6 | Safe House(2012) | Action | 1 | Adam | Smith | 1 |
| 6 | Safe House(2012) | Action | 2 | Ravi | Kumar | 2 |
| 6 | Safe House(2012) | Action | 3 | Susan | Davidson | 5 |
| 6 | Safe House(2012) | Action | 4 | Jenny | Adrianna | 8 |
| 6 | Safe House(2012) | Action | 6 | Lee | Pong | 10 |
| 7 | GIA | 18+ | 1 | Adam | Smith | 1 |
| 7 | GIA | 18+ | 2 | Ravi | Kumar | 2 |
| 7 | GIA | 18+ | 3 | Susan | Davidson | 5 |
| 7 | GIA | 18+ | 4 | Jenny | Adrianna | 8 |
| 7 | GIA | 18+ | 6 | Lee | Pong | 10 |
| 8 | Deadline(2009) | 18+ | 1 | Adam | Smith | 1 |
| 8 | Deadline(2009) | 18+ | 2 | Ravi | Kumar | 2 |
| 8 | Deadline(2009) | 18+ | 3 | Susan | Davidson | 5 |
| 8 | Deadline(2009) | 18+ | 4 | Jenny | Adrianna | 8 |
| 8 | Deadline(2009) | 18+ | 6 | Lee | Pong | 10 |
| 9 | The Dirty Picture | 18+ | 1 | Adam | Smith | 1 |
| 9 | The Dirty Picture | 18+ | 2 | Ravi | Kumar | 2 |
| 9 | The Dirty Picture | 18+ | 3 | Susan | Davidson | 5 |
| 9 | The Dirty Picture | 18+ | 4 | Jenny | Adrianna | 8 |
| 9 | The Dirty Picture | 18+ | 6 | Lee | Pong | 10 |
| 10 | Marley and me | Romance | 1 | Adam | Smith | 1 |
| 10 | Marley and me | Romance | 2 | Ravi | Kumar | 2 |
| 10 | Marley and me | Romance | 3 | Susan | Davidson | 5 |
| 10 | Marley and me | Romance | 4 | Jenny | Adrianna | 8 |
| 10 | Marley and me | Romance | 6 | Lee | Pong | 10 |

### 

### Types of joins

### Cross JOIN

### Cross JOIN is a simplest form of JOINs which matches each row from one database table to all rows of another.

### In other words it gives us combinations of each row of first table with all records in second table.

### Suppose we want to get all member records against all the movie records, we can use the script shown below to get our desired results.

### MySQL JOINS Tutorial: INNER, OUTER, LEFT, RIGHT, CROSS

### SELECT \* FROM `movies` CROSS JOIN `members`

### Executing the above script in MySQL workbench gives us the following results.

### INNER JOIN

### The inner JOIN is used to return rows from both tables that satisfy the given condition.

### Suppose , you want to get list of members who have rented movies together with titles of movies rented by them. You can simply use an INNER JOIN for that, which returns rows from both tables that satisfy with given conditions.

### MySQL JOINS Tutorial: INNER, OUTER, LEFT, RIGHT, CROSS

### 

### SELECT members.`first\_name` , members.`last\_name` , movies.`title`

### FROM members ,movies

### WHERE movies.`id` = members.`movie\_id`

### Executing the above script give

| first\_name | last\_name | Title |
| --- | --- | --- |
| Adam | Smith | ASSASSIN'S CREED: EMBERS |
| Ravi | Kumar | Real Steel(2012) |
| Susan | Davidson | Safe (2012) |
| Jenny | Adrianna | Deadline(2009) |
| Lee | Pong | Marley and me |

### Note the above results script can also be written as follows to achieve the same results.

### SELECT A.`first\_name` , A.`last\_name` , B.`title`

### FROM `members`AS A

### INNER JOIN `movies` AS B

### ON B.`id` = A.`movie\_id`

### Outer JOINs

### MySQL Outer JOINs return all records matching from both tables .

### It can detect records having no match in joined table. It returns NULL values for records of joined table if no match is found.

### Sounds Confusing ? Let's look into an example -

### LEFT JOIN

### Assume now you want to get titles of all movies together with names of members who have rented them. It is clear that some movies have not being rented by any one. We can simply use LEFT JOIN for the purpose.

### MySQL JOINS Tutorial: INNER, OUTER, LEFT, RIGHT, CROSS

### 

### The LEFT JOIN returns all the rows from the table on the left even if no matching rows have been found in the table on the right. Where no matches have been found in the table on the right, NULL is returned.

### 

### SELECT A.`title` , B.`first\_name` , B.`last\_name`

### FROM `movies` AS A

### LEFT JOIN `members` AS B

### ON B.`movie\_id` = A.`id`

### Executing the above script in MySQL workbench gives.You can see that in the returned result which is listed below that for movies which are not rented, member name fields are having NULL values. That means no matching member found members table for that particular movie.

| Title | first\_name | last\_name |
| --- | --- | --- |
| ASSASSIN'S CREED: EMBERS | Adam | Smith |
| Real Steel(2012) | Ravi | Kumar |
| Safe (2012) | Susan | Davidson |
| Deadline(2009) | Jenny | Adrianna |
| Marley and me | Lee | Pong |
| Alvin and the Chipmunks | NULL | NULL |
| The Adventures of Tin Tin | NULL | NULL |
| Safe House(2012) | NULL | NULL |
| GIA | NULL | NULL |
| The Dirty Picture | NULL | NULL |

### Note: Null is returned for non-matching rows on right

### 

### RIGHT JOIN

### RIGHT JOIN is obviously the opposite of LEFT JOIN. The RIGHT JOIN returns all the columns from the table on the right even if no matching rows have been found in the table on the left. Where no matches have been found in the table on the left, NULL is returned.

### In our example,  let's assume that you need to get names of members and movies rented by them. Now we have a new member who has not rented any movie yet

### MySQL JOINS Tutorial: INNER, OUTER, LEFT, RIGHT, CROSS

### SELECT A.`first\_name` , A.`last\_name`, B.`title`

### FROM `members` AS A

### RIGHT JOIN `movies` AS B

### ON B.`id` = A.`movie\_id`

### 

### Executing the above script in MySQL workbench gives the following results.

| first\_name | last\_name | Title |
| --- | --- | --- |
| Adam | Smith | ASSASSIN'S CREED: EMBERS |
| Ravi | Kumar | Real Steel(2012) |
| Susan | Davidson | Safe (2012) |
| Jenny | Adrianna | Deadline(2009) |
| Lee | Pong | Marley and me |
| NULL | NULL | Alvin and the Chipmunks |
| NULL | NULL | The Adventures of Tin Tin |
| NULL | NULL | Safe House(2012) |
| NULL | NULL | GIA |
| NULL | NULL | The Dirty Picture |

### Note: Null is returned for non-matching rows on left

### "ON" and "USING" clauses

### In above JOIN query examples, we have used ON clause to match the records between table.

### USING clause can also be used for the same purpose. The difference with USING is it needs to have identical names for matched columns in both tables.

### 

### In "movies" table so far we used its primary key with the name "id". We referred to same in  "members" table with the name "movie\_id".

### 

### Let's rename "movies" tables "id" field to have the name "movie\_id". We do this in order to have identical matched field names.

### ALTER TABLE `movies` CHANGE `id` `movie\_id` INT( 11 ) NOT NULL AUTO\_INCREMENT;

### Next let's use USING with above LEFT JOIN example.

### SELECT A.`title` , B.`first\_name` , B.`last\_name`

### FROM `movies` AS A

### LEFT JOIN `members` AS B

### USING ( `movie\_id` )

### Apart from using ON and USING with JOINs you can use many other MySQL clauses like GROUP BY, WHERE and even functions like SUM, AVG, etc.

### 

### Aggregate functions are a bunch of methods that operate on a set of values. They can do calculations for us and then returns one final value.

### For example, you may like to compute the sum of the data values in a given field. The following are aggregate functions that we are covering in this tutorial.

### 1. [COUNT function](https://www.techbeamers.com/mysql-aggregate-functions/#count) 2. [MIN function](https://www.techbeamers.com/mysql-aggregate-functions/#min) 3. [MAX function](https://www.techbeamers.com/mysql-aggregate-functions/#max) 4. [SUM function](https://www.techbeamers.com/mysql-aggregate-functions/#sum) 5. [AVG function](https://www.techbeamers.com/mysql-aggregate-functions/#avg)

### MySQL Aggregate Functions with Examples

### Before we go through each of the function one by one. Let’s first have a sample data table we’ll use to demonstrate the usage.

### CREATE TABLE empl (month INT, emp VARCHAR(15), dept VARCHAR(15), salary INT);

### INSERT INTO empl VALUES

### (1, "Oliver", "HR", 9000),

### (1, "George", "IT", 8000),

### (3, "Harry", "HR", 20000),

### (6, "Jack", "IT", 110123),

### (6, "Jacob", "SALES", 3000),

### (12, "Noah", "SALES", 101000),

### (12, "Charlie", "IT", 123456);

### As stated above, you can apply MySQL aggregate functions on a set of data values and do some calculations. These methods would discard NULL values unless you specified.

### MySQL Aggregate Functions

### COUNT

### If you want to count total records matching a condition, then call the COUNT function to get the number. However, it returns zero when no matching rows exist in the table.

### Syntax:

### SELECT COUNT([DISTINCT] field\_name)

### FROM target\_table

### [WHERE test\_expr];

### Example

### Let’s learn how to use COUNT in different situations.

### -- Count total no. of employess

### SELECT COUNT(\*) FROM empl;

### -- Count total no. of employees in HR dept

### SELECT COUNT(\*) FROM empl WHERE dept = "HR";

### -- Count total no. of employees in each dept

### SELECT COUNT(\*), dept FROM empl GROUP BY dept;

### -- Count total no. of employees earning 9000 or below in each dept

### SELECT COUNT(\*), dept FROM empl WHERE salary <= 9000 GROUP BY dept;

### -- Count distinct joining months in the empl table

### SELECT COUNT(DISTINCT month) FROM empl;

### After running the above MySQL commands, the output is:

### | 7

### +-------------

### | 2

### +-------------

### | 2 HR

### | 3 IT

### | 2 SALES

### +-------------

### | 1 HR

### | 1 IT

### | 1 SALES

### +-------------

### | 4

### +-------------

### MIN

### If you want to find the minimum from a set of values, then call the MIN function to get the number. However, it returns zero when no matching rows exist in the table.

### Syntax:

### SELECT MIN(field\_name)

### FROM target\_table

### [WHERE test\_expr];

### Example

### Let’s learn how to use MIN in different situations.

### -- Find the employee with the lowest salary

### SELECT MIN(salary) FROM empl;

### -- Find the lowest salaries in each dept

### SELECT dept, MIN(salary) FROM empl GROUP BY dept;

### -- Find month-wise minimum salaries

### SELECT month, MIN(salary) FROM empl GROUP BY month;

### -- Find full employee detail having the lowest salary

### SELECT \* FROM empl

### WHERE salary = (SELECT MIN(salary) FROM empl);

### After running the above MySQL commands, the output is:

### | 3000

### +--------------

### | HR 9000

### | IT 8000

### | SALES 3000

### +--------------

### | 1 8000

### | 3 20000

### | 6 3000

### +--------------

### | 12 101000

### +----------------------------

### | 6 Jacob SALES 3000

### +----------------------------

### MAX

### If you want to find the maximum from a set of values, then call the MAX function to get the number. However, it returns zero when no matching rows exist in the table.

### Syntax:

### SELECT MAX(field\_name)

### FROM target\_table

### [WHERE test\_expr];

### Example

### Let’s learn how to use MAX in different situations.

### -- Find the employee with the highest salary

### SELECT MAX(salary) FROM empl;

### -- Find the highest salaries in each dept

### SELECT dept, MAX(salary) FROM empl GROUP BY dept;

### -- Find month-wise maximum salaries

### SELECT month, MAX(salary) FROM empl GROUP BY month;

### -- Find full employee detail having the highest salary

### SELECT \* FROM empl

### WHERE salary = (SELECT MAX(salary) FROM empl);

### After running the above MySQL commands, the output is:

### | 123456

### +--------------

### | HR 20000

### | IT 123456A

### | SALES 101000

### +--------------

### | 1 9000

### | 3 20000

### | 6 110123

### +--------------

### | 12 123456

### +------------------------------

### | 12 Charlie IT 123456

### +------------------------------

### SUM

### If you want to the total of a set of values, then call the SUM function to get the result. However, it returns NULL when no matching rows exist in the table.

### Syntax:

### SELECT SUM(field\_name)

### FROM target\_table

### [WHERE test\_expr];

### Example

### Let’s learn how to use the SUM in different situations.

### -- Find the sum all employee salaries

### SELECT SUM(salary) FROM empl;

### -- Find the sum of salaries in each dept

### SELECT dept, SUM(salary) FROM empl GROUP BY dept;

### -- Find month-wise sum of salaries

### SELECT month, SUM(salary) FROM empl GROUP BY month;

### After running the above MySQL commands, the output is:

### | 374579

### +--------------

### | HR 29000

### | IT 241579

### | SALES 104000

### +--------------

### | 1 17000

### | 3 20000

### | 6 113123

### | 12 224456

### +--------------

### AVG

### If you want to find the average of a set of values, then call the AVG function to get the result. However, it returns zero when no matching rows exist in the table.

### Syntax:

### SELECT AVG(field\_name)

### FROM target\_table

### [WHERE test\_expr];

### Example

### Let’s learn how to use AVG in different situations.

### -- Find the AVG of all employee salaries

### SELECT AVG(salary) FROM empl;

### -- Find the AVG of salaries in each dept

### SELECT dept, AVG(salary) FROM empl GROUP BY dept;

### -- Find month-wise AVG of salaries

### SELECT month, AVG(salary) FROM empl GROUP BY month;

### After running the above MySQL commands, the output is:

### | 53511.2857

### +-------------------

### | HR 14500.0000

### | IT 80526.3333

### | SALES 52000.0000

### +-------------------

### | 1 8500.0000

### | 3 20000.0000

### | 6 56561.5000

**Experiment No.2**

* 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.

T**riggers are of two types** according to the [SQL](https://www.javatpoint.com/sql-tutorial) standard: row-level triggers and statement-level triggers.

**Row-Level Trigger:** It is a trigger, which is activated for each row by a triggering statement such as insert, update, or delete. For example, if a table has inserted, updated, or deleted multiple rows, the row trigger is fired automatically for each row affected by the [insert](https://www.javatpoint.com/mysql-insert), [update](https://www.javatpoint.com/mysql-update), or [delete statement](https://www.javatpoint.com/mysql-delete).

**Statement-Level Trigger:** It is a trigger, which is fired once for each event that occurs on a table regardless of how many rows are inserted, updated, or deleted.

NOTE: MySQL doesn't support statement-level triggers. It provides supports for row-level triggers only.

### Why we need/use triggers in MySQL?

We need/use triggers in MySQL due to the following features:

* Triggers help us to enforce business rules.
* Triggers help us to validate data even before they are inserted or updated.
* Triggers help us to keep a log of records 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.

### Limitations of Using Triggers in MySQL

* MySQL triggers do not allow to use of all validations; they only provide extended validations. **For example**, we can use the NOT NULL, UNIQUE, CHECK and FOREIGN KEY constraints for simple validations.
* Triggers are invoked and executed invisibly from the client application. Therefore, it isn't easy to troubleshoot what happens in the database layer.
* Triggers may increase the overhead of the database server.

### Types of Triggers in MySQL?

We can define the maximum six types of actions or events in the form of triggers:

1. [Before Insert](https://www.javatpoint.com/mysql-before-insert-trigger)**:** It is activated before the insertion of data into the table.
2. [After Insert](https://www.javatpoint.com/mysql-after-insert-trigger)**:** It is activated after the insertion of data into the table.
3. [Before Update](https://www.javatpoint.com/mysql-before-update-trigger)**:** It is activated before the update of data in the table.
4. [After Update](https://www.javatpoint.com/mysql-after-update-trigger)**:** It is activated after the update of the data in the table.
5. [Before Delete](https://www.javatpoint.com/mysql-before-delete-trigger)**:** It is activated before the data is removed from the table.
6. [After Delete](https://www.javatpoint.com/mysql-after-delete-trigger)**:** It is activated after the deletion of data from the table.

When we use a statement that does not use INSERT, UPDATE or DELETE query to change the data in a table, the triggers associated with the trigger will not be invoked.

### Naming Conventions

Naming conventions are the set of rules that we follow to give appropriate unique names. It saves our time to keep the work organize and understandable.

Therefore, **we must use a unique name for each trigger associated with a table**. However, it is a good practice to have the same trigger name defined for different tables.

The following naming convention should be used to name the trigger in [MySQL](https://www.javatpoint.com/mysql-tutorial):

1. (BEFOR | **AFTER**) table\_name (**INSERT** | **UPDATE** | **DELETE**)

Thus,

### **Trigger Activation Time:** BEFORE | AFTER

### How to create triggers in MySQL?

1. **CREATE** **TRIGGER** trigger\_name
2. (**AFTER** | BEFORE) (**INSERT** | **UPDATE** | **DELETE**)
3. **ON** table\_name **FOR** EACH ROW
4. **BEGIN**
5. --variable declarations
6. --trigger code
7. **END**;

Now we are going to learn how to create the first trigger in MySQL. We can create a new trigger in MySQL by using the CREATE TRIGGER statement. It is to ensure that we have trigger privileges while using the CREATE TRIGGER command. The following is the basic syntax to create a trigger:

1. **CREATE** **TRIGGER** trigger\_name  trigger\_time trigger\_event
2. **ON** table\_name **FOR** EACH ROW
3. **BEGIN**
4. --variable declarations
5. --trigger code
6. **END**;

### Parameter ExplanationThe create trigger syntax contains the following parameters:

**trigger\_name:** It is the name of the trigger that we want to create. It must be written after the CREATE [TRIGGER statement](https://www.javatpoint.com/mysql-trigger). It is to make sure that the trigger name should be unique within the schema.

**trigger\_time:** It is the trigger action time, which should be either BEFORE or AFTER. It is the required parameter while defining a trigger. It indicates that the trigger will be invoked before or after each row modification occurs on the table.

**trigger\_event:** It is the type of operation name that activates the trigger. It can be either [INSERT](https://www.javatpoint.com/mysql-insert), [UPDATE](https://www.javatpoint.com/mysql-update), or [DELETE](https://www.javatpoint.com/mysql-delete) operation. The trigger can invoke only one event at one time. If we want to define a trigger which is invoked by multiple events, it is required to define multiple triggers, and one for each event.

**table\_name:** It is the name of the table to which the trigger is associated. It must be written after the ON keyword. If we did not specify the table name, a trigger would not exist.

**BEGIN END Block:** Finally, we will specify the statement for execution when the trigger is activated. If we want to execute multiple statements, we will use the BEGIN END block that contains a set of queries to define the logic for the trigger.

The trigger body can access the column's values, which are affected by the DML statement. The **NEW** and **OLD** modifiers are used to distinguish the column values **BEFORE** and **AFTER** the execution of the DML statement. We can use the column name with NEW and OLD modifiers as **OLD.col\_name** and **NEW.col\_name**. The OLD.column\_name indicates the column of an existing row before the updation or deletion occurs. NEW.col\_name indicates the column of a new row that will be inserted or an existing row after it is updated.

**For example**, suppose we want to update the column name **message\_info** using the trigger. In the trigger body, we can access the column value before the update as **OLD.message\_info** and the new value **NEW.message\_info**.

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

| **Trigger Event** | **OLD** | **NEW** |
| --- | --- | --- |
| INSERT | No | Yes |
| UPDATE | Yes | Yes |
| ELETE | Yes | No |

### MySQL Trigger Example

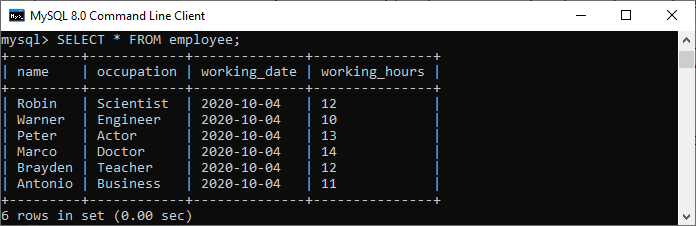
Let us start creating a trigger in [MySQL](https://www.javatpoint.com/mysql-tutorial) that makes modifications in the employee table. First, we will create a new table named **employee** by executing the below statement:

1. **CREATE** **TABLE** employee(
2. **name** **varchar**(45) NOT NULL,
3. occupation **varchar**(35) NOT NULL,
4. working\_date **date**,
5. working\_hours **varchar**(10)
6. );

Next, execute the below statement to **fill the records** into the employee table:

1. **INSERT** **INTO** employee **VALUES**
2. ('Robin', 'Scientist', '2020-10-04', 12),
3. ('Warner', 'Engineer', '2020-10-04', 10),
4. ('Peter', 'Actor', '2020-10-04', 13),
5. ('Marco', 'Doctor', '2020-10-04', 14),
6. ('Brayden', 'Teacher', '2020-10-04', 12),
7. ('Antonio', 'Business', '2020-10-04', 11);

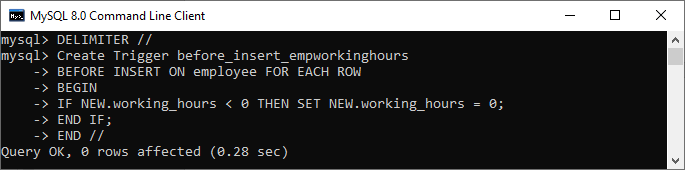
Next, execute the [**SELECT statement**](https://www.javatpoint.com/mysql-select) to verify the inserted record:



Next, we will create a [**BEFORE INSERT trigger**](https://www.javatpoint.com/mysql-before-insert-trigger). This trigger is invoked automatically insert the **working\_hours = 0** if someone tries to insert **working\_hours < 0**.

1. mysql> DELIMITER //
2. mysql> **Create** **Trigger** before\_insert\_empworkinghours
3. BEFORE **INSERT** **ON** employee **FOR** EACH ROW
4. **BEGIN**
5. IF NEW.working\_hours < 0 **THEN** **SET** NEW.working\_hours = 0;
6. **END** IF;
7. **END** //

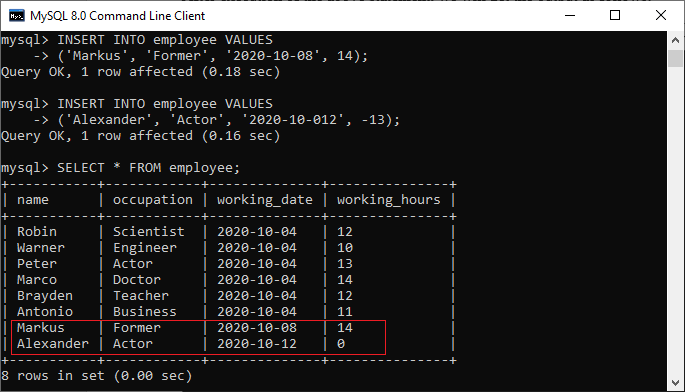
If the trigger is created successfully, we will get the output as follows:



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

1. mysql> **INSERT** **INTO** employee **VALUES**
2. ('Markus', 'Former', '2020-10-08', 14);
4. mysql> **INSERT** **INTO** employee **VALUES**
5. ('Alexander', 'Actor', '2020-10-012', -13);

After execution of the above statement, we will get the output as follows:



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

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

# **MySQL Functions**

## Creating a function

In MySQL, Function can also be created. A function always returns a value using the return statement. The function can be used in SQL queries.

## Syntax

1. **CREATE** **FUNCTION** function\_name [ (parameter datatype [, parameter datatype]) ]
2. **RETURNS** return\_datatype
3. **BEGIN**
4. Declaration\_section
5. Executable\_section
6. **END**;

## Parameter:

**Function\_name:** name of the function

**Parameter:** number of parameter. It can be one or more than one.

**return\_datatype:** return value datatype of the function

**declaration\_section:** all variables are declared.

**executable\_section:** code for the function is written here.

## Example 1

**Step 1:** Create database and table.

**Database:** employee

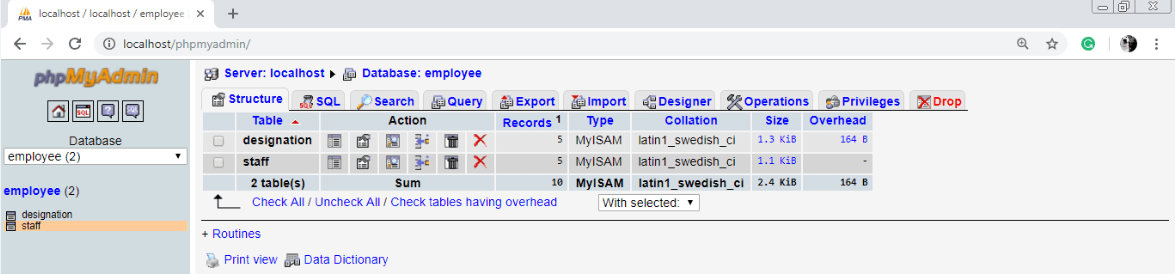


Table 1 : designation

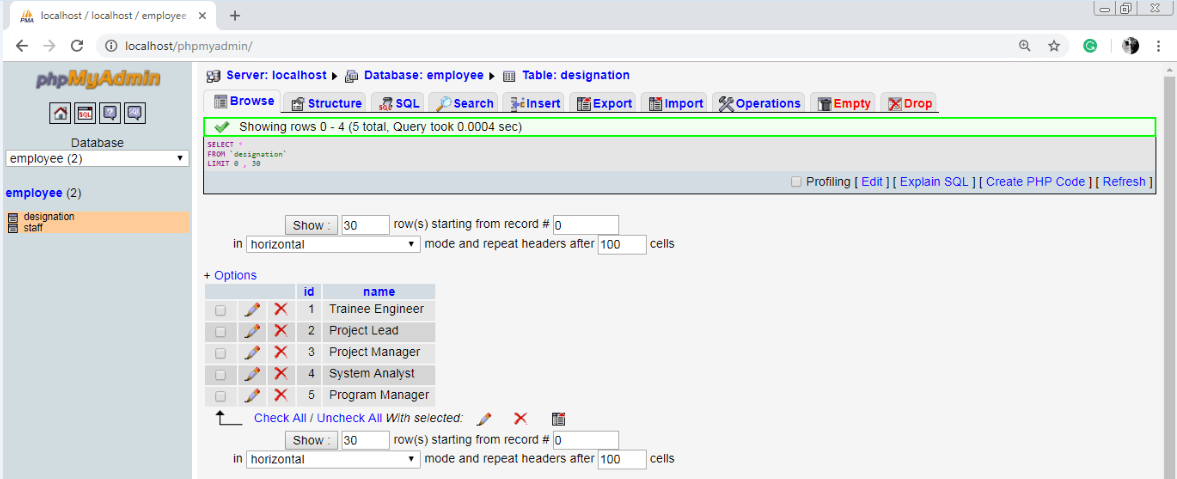
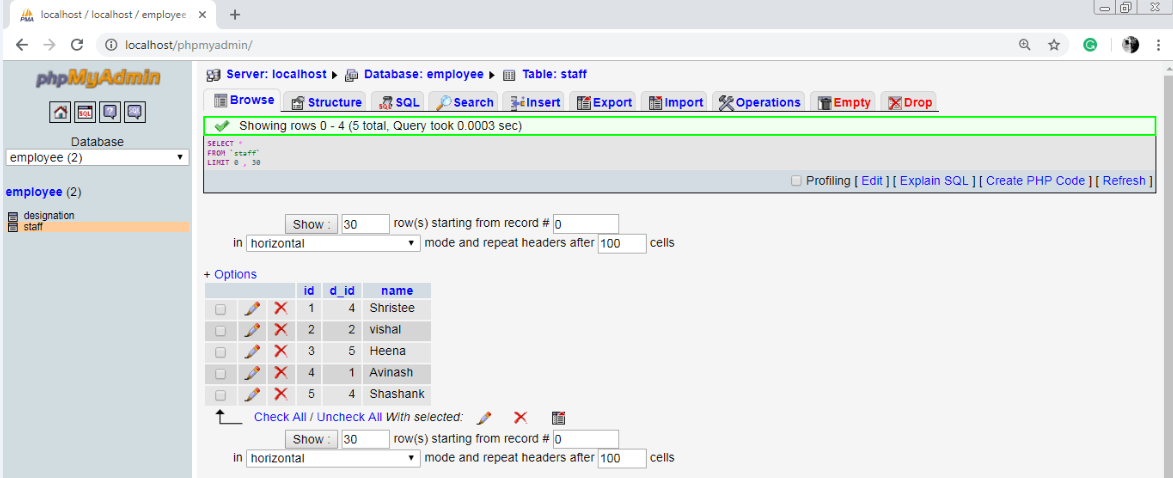


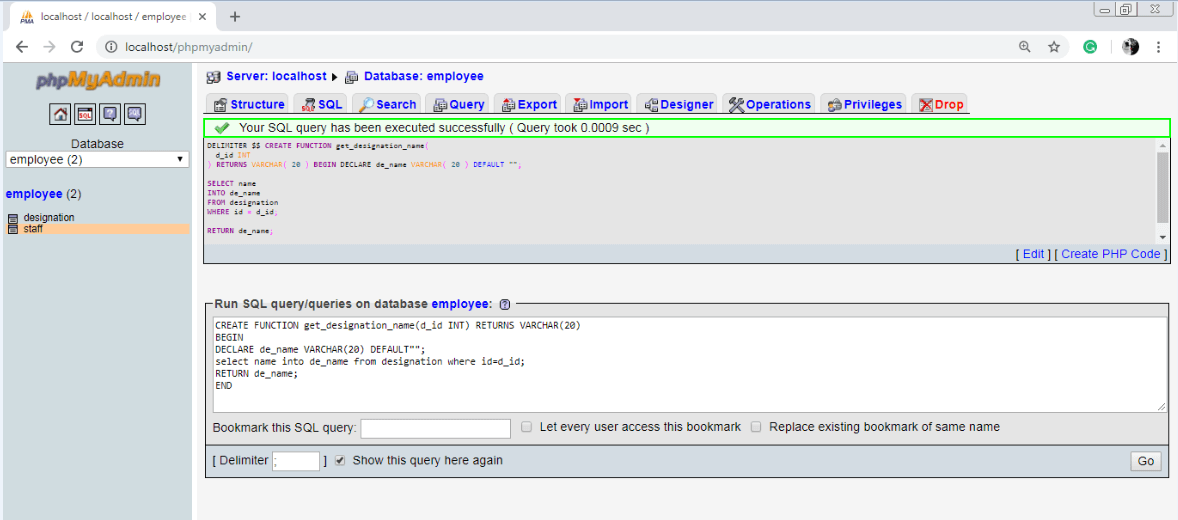
Table 2 : staff



**Step 2:** Create a function

**Function query:**

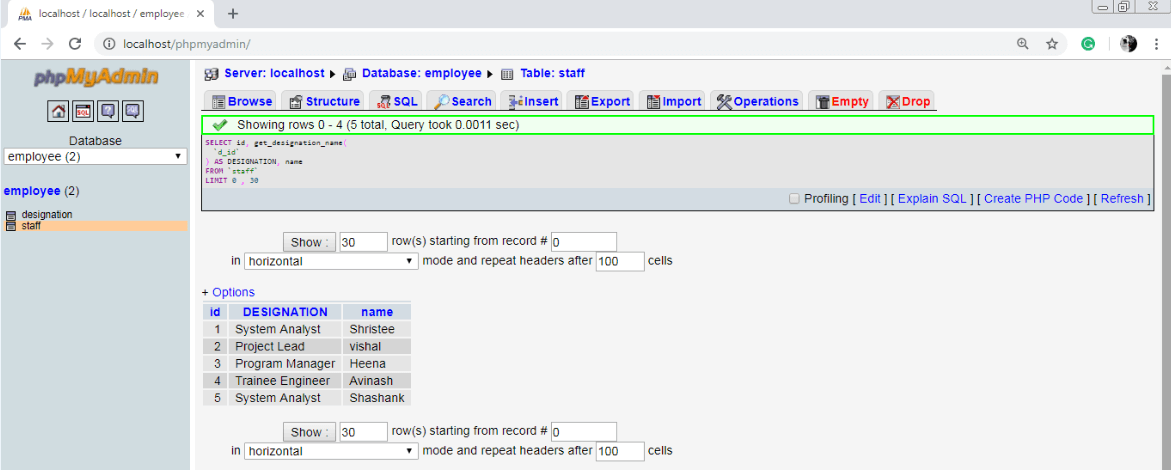
1. DELIMITER $$
2. **CREATE** **FUNCTION** get\_designation\_name(d\_id **INT**) **RETURNS** **VARCHAR**( 20 )
3. **BEGIN**
4. **DECLARE** de\_name **VARCHAR**( 20 ) **DEFAULT** "";
5. **SELECT** **name** **INTO** de\_name **FROM** designation **WHERE** id = d\_id;
6. **RETURN** de\_name;
7. **END** $$



**Step 3:** Execute the function

**Query :**

SELECT id, get\_designation1(`d\_id`) as DESIGNATION, name FROM 'staff'



## Drop a function

In MySQL Function can also be dropped. When A function id dropped, it is removed from the database.

## Syntax:

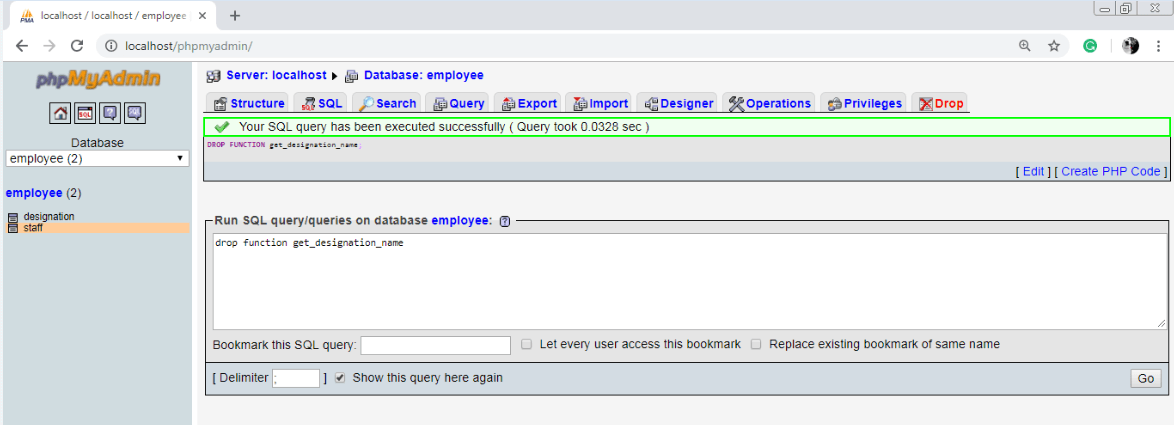
1. **Drop** **function** [ IF EXISTS ] function\_name;

## Parameter

**function\_name:** name of the function to be dropped.

## Example 1:

drop function get\_designation\_name;



Now we will learn how to create stored functions using the CREATE FUNCTION statement.

A stored function is a special kind stored program that returns a single value. Typically, you use stored functions to encapsulate common formulas or business rules that are reusable among SQL statements or stored programs.

Different from a [stored procedure](https://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx), you can use a stored function in SQL statements wherever an expression is used. This helps improve the readability and maintainability of the procedural code.

To create a stored function, you use the CREATE FUNCTION statement.

MySQL CREATE FUNCTION syntax

The following illustrates the basic syntax for creating a new stored function:

DELIMITER $$

CREATE FUNCTION function\_name(

param1,

param2,…

)

RETURNS datatype

[NOT] DETERMINISTIC

BEGIN

-- statements

END $$

DELIMITER ;

Code language: SQL (Structured Query Language) (sql)

In this syntax:

First, specify the name of the stored function that you want to create after CREATE FUNCTION  keywords.

Second, list all [parameters](https://www.mysqltutorial.org/stored-procedures-parameters.aspx) of the stored function inside the parentheses followed by the function name. By default, all parameters are the IN parameters. You cannot specify IN , OUT or INOUT modifiers to parameters

Third, specify the data type of the return value in the RETURNS statement, which can be any valid [MySQL data types](https://www.mysqltutorial.org/mysql-data-types.aspx).

Fourth, specify if a function is deterministic or not using the DETERMINISTIC keyword.

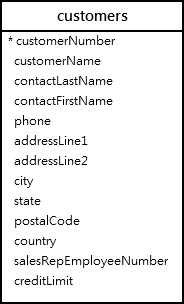
A deterministic function always returns the same result for the same input parameters whereas a non-deterministic function returns different results for the same input parameters.

If you don’t use DETERMINISTIC or NOT DETERMINISTIC, MySQL uses the NOT DETERMINISTIC option by default.

Fifth, write the code in the body of the stored function in the BEGIN END block. Inside the body section, you need to specify at least one RETURN statement. The RETURN statement returns a value to the calling programs. Whenever the RETURN statement is reached, the execution of the stored function is terminated immediately.

MySQL CREATE FUNCTION example

Let’s take the example of creating a stored function. We will use the customers table in the [sample database](https://www.mysqltutorial.org/mysql-sample-database.aspx) for the demonstration.



#### The following CREATE FUNCTION statement creates a function that returns the customer level based on credit:

#### DELIMITER $$

#### CREATE FUNCTION CustomerLevel(

#### credit DECIMAL(10,2)

#### )

#### RETURNS VARCHAR(20)

#### DETERMINISTIC

#### BEGIN

#### DECLARE customerLevel VARCHAR(20);

#### IF credit > 50000 THEN

#### SET customerLevel = 'PLATINUM';

#### ELSEIF (credit >= 50000 AND

#### credit <= 10000) THEN

#### SET customerLevel = 'GOLD';

#### ELSEIF credit < 10000 THEN

#### SET customerLevel = 'SILVER';

#### END IF;

#### -- return the customer level

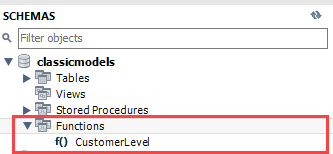
#### RETURN (customerLevel);

#### END$$

#### DELIMITER ;

Code language: SQL (Structured Query Language) (sql)

Once the function is created, you can view it in MySQL Workbench under the **Functions** section:



Or you can view all stored functions in the current classicmodels database by using the SHOW FUNCTION STATUS as follows:

### SHOW FUNCTION STATUS

### WHERE db = 'classicmodels';

Code language: SQL (Structured Query Language) (sql)

mysql stored function - show function status

# Calling a stored function in an SQL statement

# The following statement uses the CustomerLevel stored function:

# SELECT

# customerName,

# CustomerLevel(creditLimit)

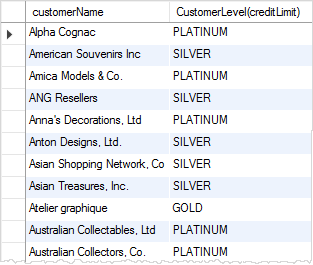
# FROM

# customers

# ORDER BY

customerName;

Code language: SQL (Structured Query Language) (sql)



Calling a stored function in a stored procedure

# A view is a “virtual” table that is derived from other tables

* Allows for limited update operations
  + Since the table may not physically be stored

# Allows full query operations



* Create a view for Department Managers:

## CREATE VIEW MANAGER AS

**SELECT FNAME, LNAME, DName, Dnumber, SALARY FROM EMPLOYEE, DEPARTMENT**

**WHERE SSN=MGRSSN AND DNO=DNUMBER;**

* Find employees who earn more than their managers

## SELECT E.FNAME, E.LNAME FROM EMPLOYEE E, MANAGER M

**WHERE E.DNO=M.DNUMBER AND E.SALARY > M.SALARY;**

* When no longer needed, a view can be dropped:

## DROP VIEW MANAGER;



* There are two ways to implement a view:
* Approach 1: Query modification
  + Modify the view query into a query on the underlying base tables
  + Example:

SELECT \* FROM Manager WHERE Salary > 100000 becomes

SELECT Fname, Lname, Dname, Dnumber, Salary FROM EMPLOYEE, DEPARTMENT

WHERE SSN=MgrSSN AND Salary > 100000

* + Disadvantage:
    - Inefficient for views defined via complex queries



# Approach 2: View materialization

* + Involves physically creating and keeping a temporary table
  + Concerns:
    - Maintaining correspondence between the base table and the view when the base table is updated

# ORACLE

CREATE MATERIALIZED VIEW or CREATE SNAPSHOT



* Update on a view can be implemented by mapping it to an update on the underlying base table

UPDATE MANAGER

SET Salary = 1.1\*Salary WHERE Dname = ‘Research’;

* + Becomes:

UPDATE EMPLOYEE

SET Salary = 1.1\*Salary

WHERE SSN in (SELECT MgrSSN

FROM DEPARTMENT

WHERE DName = ‘Research’);

* Updating views involving joins are not always possible
  + Views defined using groups and aggregate functions are not updateable
* For mySQL**,** the keyword **“WITH CHECK OPTION”** must be added to the view definition if the view is to be updated



* A stored procedure contains a sequence of SQL commands stored in the database catalog so that it can be invoked later by a program
* Stored procedures are declared using the following syntax:

Create Procedure <proc-name>

(param\_spec1, param\_spec2, …, param\_specn )

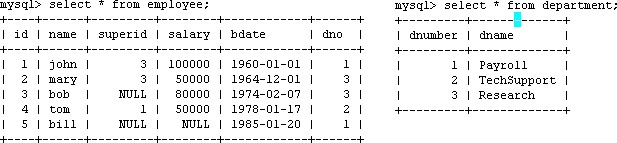
begin

-- execution code end;

where each param\_spec is of the form:

[in | out | inout] <param\_name> <param\_type>

* in mode: allows you to pass values into the procedure,
* out mode: allows you to pass value back from procedure to the calling program



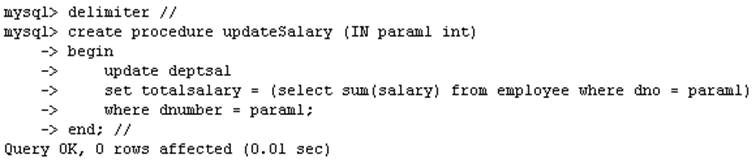
* Suppose we want to keep track of the total salaries of employees working for each department





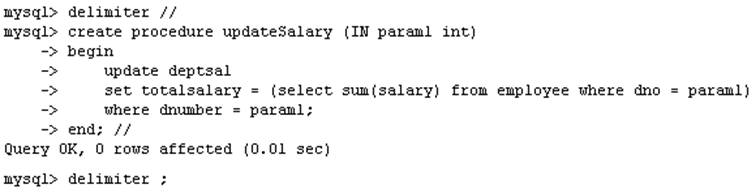
Step 1: Change the delimiter (i.e., terminating character) of SQL statement from semicolon (;) to something else (e.g., //)

So that you can distinguish between the semicolon of the SQL statements in the procedure and the terminating character of the procedure definition

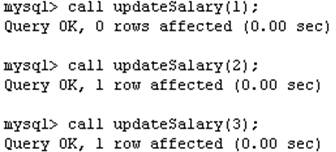


Step 2:

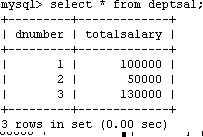
1. Define a procedure called updateSalary which takes as input a department number.
2. The body of the procedure is an SQL command to update the totalsalary column of the deptsal table.
3. Terminate the procedure definition using the delimiter you had defined in step 1 (//)



Step 3: Change the delimiter back to semicolon (;)



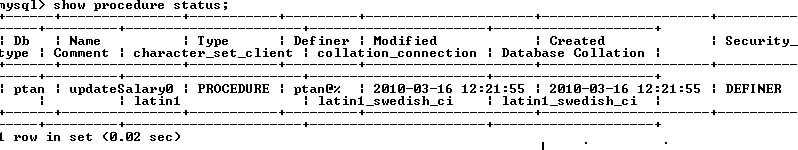
Step 4: Call the procedure to update the totalsalary for each department



Step 5: Show the updated total salary in the deptsal table



# Use show procedure status to display the list of stored procedures you have created



* Use drop procedure to remove a stored procedure





* You can declare variables in stored procedures
* You can use flow control statements (conditional IF- THEN-ELSE or loops such as WHILE and REPEAT)
* MySQL also supports cursors in stored procedures.
  + A cursor is used to iterate through a set of rows returned by a query so that we can process each individual row.

# To learn more about stored procedures, go to:

<http://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx>



**PL SQL Cursor**

Oracle has dedicated memory locations for executing SQL statements and then it holds that processed information, **for example**, the total number of rows updated.

A cursor in PL/SQL gives a name and acts as a pointer to the area of work called a context area and then uses its information. It keeps the number of rows processed by the SQL statement. These rows are called as an active set. The size of the active set is equal to the count of the rows that meet the condition.

**There are two types of cursors which are listed below:**

1. Implicit Cursor
2. Explicit cursor

Implicit Cursors

The implicit cursors are allocated by Oracle by default while executing SQL statements. It holds the affected rows by the DML operations like UPDATE, DELETE and INSERT. Thus implicit cursors are used when we don’t have an explicit cursor in place.

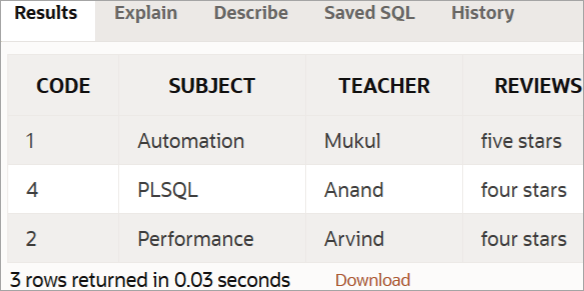
While we are inserting a row, the cursor keeps that particular data. Similarly, for deletion and updating operations, the affected rows are stored by the cursors. The implicit cursors are not given any names and hence cannot be manipulated by the developers and the data contained on it cannot be used anywhere.

The latest updated cursors can be used with the help of cursor attributes. These attributes are the properties that help to refer to the data type and structure of items without repeating their definitions. All the columns and tables in a database have a common attribute (represented by % sign) characteristics and they can be used as **sql%attribute\_name.**

| **Sl No.** | **Name** | **Purposes** |
| --- | --- | --- |
| **1** | **%FOUND** | Gives the result in boolean. Returns true if DELETE, INSERT, UPDATE or SELECT statements affect single or multiple rows. Or else false is returned. |
| **2** | **%NOTFOUND** | Gives the result in boolean and has reverse functionality of %FOUND. Returns true if DELETE, INSERT, UPDATE or SELECT statements affect no rows. Or else false is returned. |
| **3** | **%ISOPEN** | Gives the result in boolean. Returns true if the cursor is currently open. Or else false is returned. |
| **4** | **%ROWCOUNT** | Gives the count of the number of rows fetched from DELETE, INSERT, UPDATE or SELECT statements. |
| **5** | **%TYPE** | Gives the datatype of the column or variable of the database. |
| **6** | **%ROWTYPE** | Gives the record type that is equivalent to a database row. |

**Let us consider a table which is named TUTOR.**

SELECT \* FROM TUTOR;



**We have created a table with the SQL statement given below:**

| CREATE TABLE TUTOR(  CODE INT NOT NULL,  SUBJECT VARCHAR(15) NOT NULL,  TEACHER VARCHAR(15),  REVIEWS VARCHAR (10) NOT NULL,PRIMARY KEY (CODE)); |
| --- |

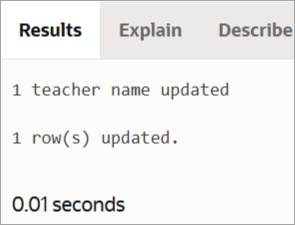
**Inserted values to this table with SQL statements given below:**

| INSERT INTO TUTOR (CODE,SUBJECT,TEACHER,REVIEWS)  VALUES (1, 'Automation', 'Mukul', 'five stars');  INSERT INTO TUTOR (CODE,SUBJECT,TEACHER,REVIEWS)  VALUES (4, 'PLSQL', 'Anand', 'four stars');  INSERT INTO TUTOR (CODE,SUBJECT,TEACHER,REVIEWS)  VALUES (2, 'Performance', 'Arvind', 'four stars'); |
| --- |

**Code implementation with the implicit cursor:**

| **DECLARE**  **total\_count number(30);**  **BEGIN**    **--updating a row**  **UPDATE TUTOR**  **SET TEACHER = 'Zen' where CODE = 1;**    **-- result in boolean, true returned if no rows affected**  **IF sql%notfound THEN**  **dbms\_output.put\_line('no subjects fetched');**      **-- result in boolean, true returned if any rows affected**  **ELSIF sql%found THEN**    **-- count the number of rows affected rows affected**  **total\_count := sql%rowcount;**  **dbms\_output.put\_line( total\_count ||'teacher name updated ');**  **END IF;**  **END;**  **/** |
| --- |

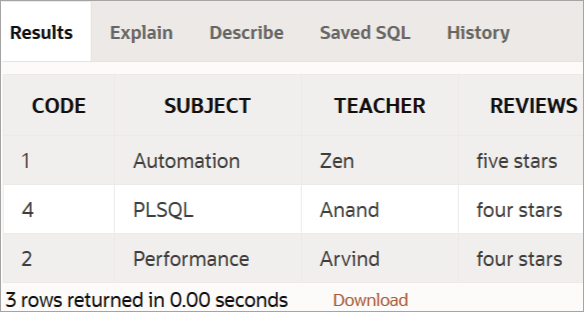
**The output of the above code should be:**

****

**Let us now verify the changes reflected in the table named TUTOR.**

**We are using a SQL statement to get the columns of the table:**

**SELECT \* FROM TUTOR;**



Thus we see that as pointed by the cursor, the name of the teacher with code = 1, gets updated to Zen.

Explicit Cursors

The developers can have their own user-defined context area to run DML operations. Thus they can exercise more power over it. The declaration section of the PL/SQL block of code contains explicit cursors. It is normally built on SELECT operations that fetch multiple rows.

**Syntax of explicit cursor:**

DECLARE

CURSOR <<cursor name>> IS <<select statement>>

<<Cursor variable>>

BEGIN

OPEN <<cursor name>>;

FETCH <<cursor name>> INTO <Cursor variable>;

.

.

CLOSE <cursor name>;

END;

**Explicit Cursor works on the processes listed below:**

**#1)** **Cursor declaration for memory initialization.** Here, a named context area is created which serves as a cursor name.

**Syntax:**

CURSOR tutorial\_s IS

SELECT code FROM TUTORIAL;

**#2)** **Cursor opening for memory allocation**. A cursor is now available for fetching the updated rows from the database.

**Syntax:**

OPEN tutorial\_s;

**#3) Cursor is fetched for getting the data.** After the SELECT operation is done, the rows obtained are put in the memory allocated and these are now considered as active sets. The cursor can access one row at a time.

**Syntax:**

FETCH tutorial\_s INTO c\_code;

**#4) Cursor is finally closed to free the allocated memory.** As all the records are obtained one by one, the cursor is closed to release context area memory.

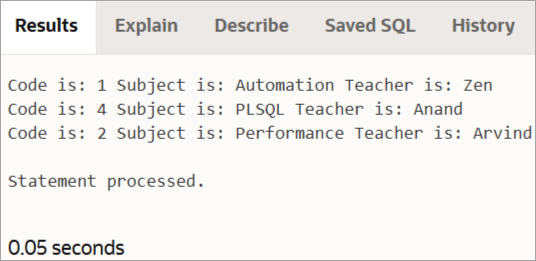
**Syntax:**

CLOSE tutorial\_s;

**Code implementation with explicit cursor:**

| DECLARE     -- cursor declaration  CURSOR t\_tutorials is  SELECT code, subject, teacher FROM Tutor;  t\_code Tutor.code%type;  t\_subject Tutor.subject%type;  t\_teacher Tutor.teacher%type;  BEGIN       -- opening a cursor     OPEN t\_tutorials;  LOOP        -- fetching values from cursor      FETCH t\_tutorials into t\_code, t\_subject, t\_teacher;      EXIT WHEN t\_tutorials%notfound;        -- printing in console      dbms\_output.put\_line('Code is: ' || t\_code || ' ' || 'Subject is: ' || t\_subject || ' Teacher is: ' || t\_teacher);  END LOOP;  CLOSE t\_tutorials;  END;  / |
| --- |

**The output of the above code should be:**



Cursor For Loop

While working with explicit cursors, we can use FOR loop instead of using statements like FETCH, OPEN, and CLOSE. Cursor FOR Loop has the loop index as a record which points to the row obtained from the database. Next after opening the cursor, it fetches the multiple rows of data repeatedly from the result set into the record fields.

Finally, the cursor is closed after all the rows are obtained. We use a dot (.) sign to refer to each field in the record. (.) dot sign is actually used for selecting a component.

**The syntax for Cursor For loop:**

| DECLARE  CURSOR c IS  SELECT code, subject, price FROM Tutorial;  ...  BEGIN  FOR Tutorial\_rec IN c LOOP  ...  price\_sum:= price\_sum + Tutorial\_rec.price;  END LOOP; |
| --- |

Here, the Cursor FOR loop declares ‘**Tutorial\_rec’** as a record.

Cursor Variables

A cursor variable is used to refer to the present row in the result set that has more than one row. It can be used for any type of query. It is similar to a variable of PL/SQL, where we can assign values and can be passed via a subprogram in the database. Thus cursor variables provide a lot of flexibility and data can be obtained in a centralized process.

# The previous procedure updates one row in deptsal table based on input parameter

* Suppose we want to update all the rows in deptsal simultaneously
  + First, let’s reset the totalsalary in deptsal to zero





Drop the old procedure

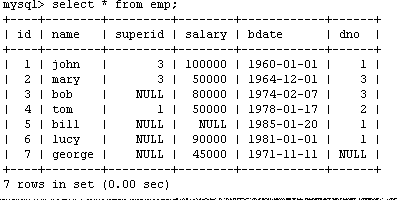
Use cursor to iterate the rows

‹#›

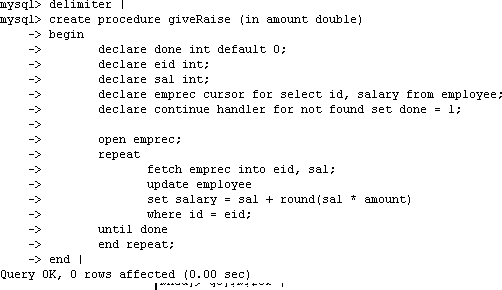


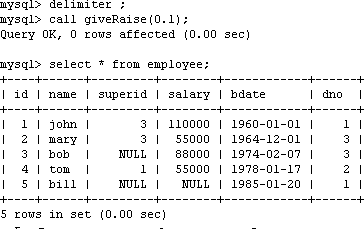
# Call procedure





* Create a procedure to give a raise to all employees









* To monitor a database and take a corrective action when a condition occurs
  + Examples:
    - Charge $10 overdraft fee if the balance of an account after a withdrawal transaction is less than $500
    - Limit the salary increase of an employee to no more than 5% raise

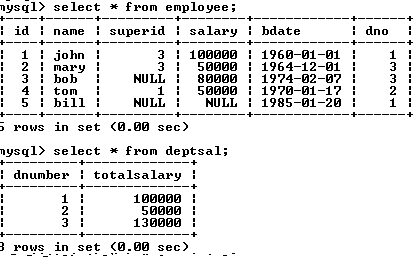
**CREATE TRIGGER** trigger-name **trigger-time trigger-event ON** table-name

## FOR EACH ROW

**trigger-action**;

* + trigger-time ∈ {BEFORE, AFTER}
  + trigger-event ∈ {INSERT,DELETE,UPDATE}

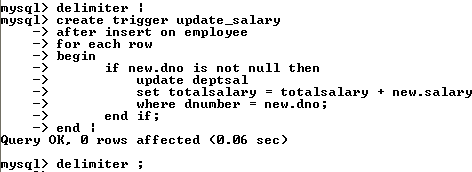




# We want to create a trigger to update the total salary of a department when a new employee is hired



* Create a trigger to update the total salary of a department when a new employee is hired:



* The keyword “new” refers to the new row inserted





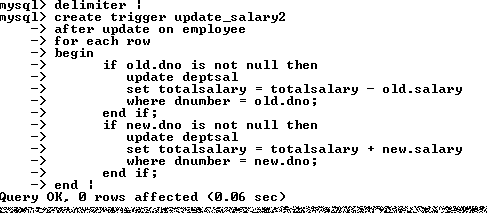
totalsalary increases by 90K

totalsalary did not change

‹#›



* A trigger to update the total salary of a department when an employee tuple is modified:



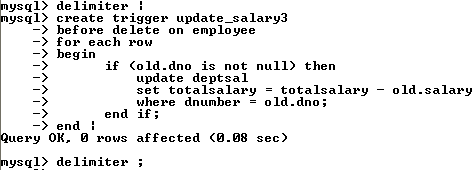




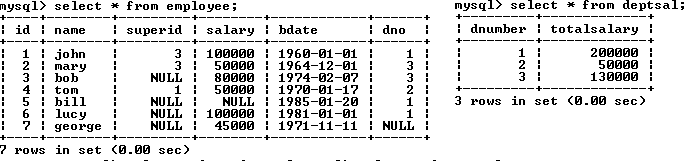
‹#›

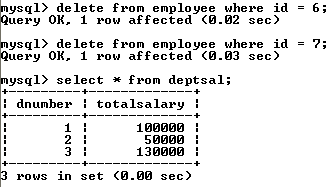


* A trigger to update the total salary of a department when an employee tuple is deleted:











# To list all the triggers you have created:

mysql> show triggers;