Experiment no – 1

Aim: **Draw E-R diagram and convert entities and relationships to relation table for a given scenario**

a. Bank

b. College

E-R diagram:

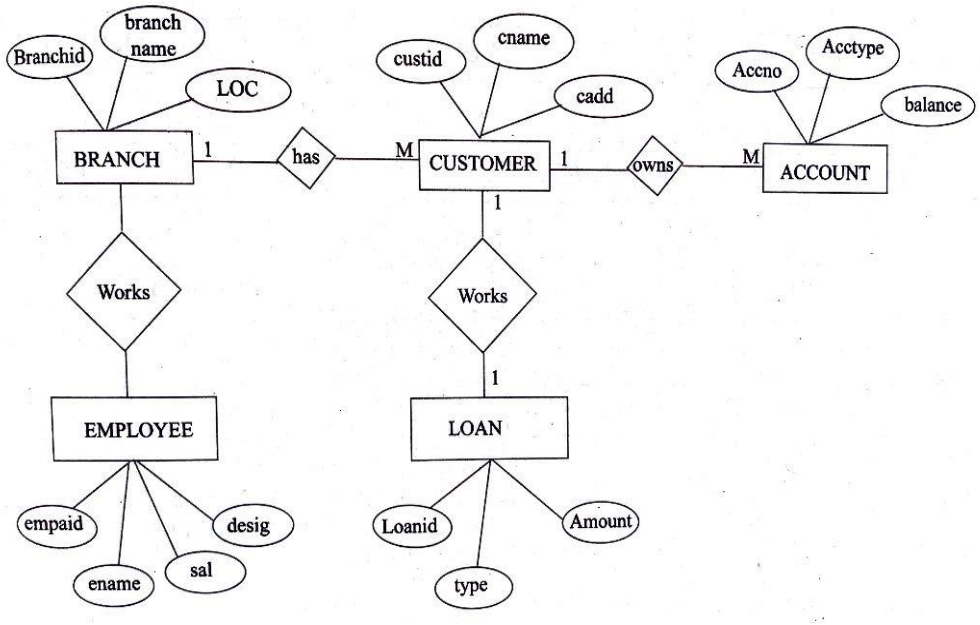

Figure-1: E-R diagram for Bank.

Theory:

Step 1- Identify the entity sets

a) Branch

b) Customer

c) Account

d) Loan

e) Employee

Step 2 - Identify the attributes for the given entities

a) Branch - Branchid, Branchname, Loc

b) Customer - Custid, Cname, Cadd

c) Account - Accno, Acctype, Balance

d) Loan - Loanid, Type, Amount

e) Employee - Empid, Ename, Sal, Desig Step 3 - Identify the Key attributes

a) Branch - Branchid, Branchname, Loc

b) Customer - Custid, Cname, Cadd

c) Account - Accno, Acctype, Balance

d) Loan - Loanid, Type, Amount

e) Employee - Empid, Ename, Sal, Desig

Step 4- Relations

Table: BRANCH

BRANCHID	BRANCH NAME	LOC

Table: CUSTOMER

CUSTID	CNAME	CADD

Table: ACCOUNT

ACCNO	ACCTYPE	BALANCE

Table: LOAN

LOANID	TYPE	AMOUNT

Table: EMPLOYEE

EMPID	ENAME	SAL	DESIG

Table: WORKS

BRANCHID	EMPNO

Table: BORROWS

CUSTID	LOANID

Table: OWNS

ACCNO	CUSTID

Table: HAS

BRANCHID	CUSTID

Conclusion: Successfully created E-R diagram and convert entities and relationships to relation table for a bank.

Experiment no – 2

Aim: **Write relational algebra queries for a given set of relations.**

Consider the given relation Emp(eid,ename,sal,job, deptid, dloc) and answer the following using Relation Alegbra:

a) Retrieve the details when sal is greater than 2000.

σ sal > 2000 (Emp)

b) Display the names and job of the employees.

π enam, job (Emp)

c) Display the name of the employees who are working as manager in London.

7 ename(σ job = "MANAGER" and dioc = "London" (Emp)

a) Rename the table emp as empdetails

σ (empdetails, emp)

e) Display the employee name when deptid is 10.

π ename(σ deptid = 10) (Emp)

Conclusion: Successfully created relational algebra queries for a given set of relations.

Experiment no – 3

Aim: **Defining data**

a. Using CREATE statement

b. Using ALTER statement

c. Using DROP statement

d. Using TRUNCATE statement

e. Using RENAME statement

Using CREATE statement:

Syntax

CREATE TABLE *table_name (*
column1 datatype,
column2 datatype,
column3 datatype,
....
);

Example:

CREATE TABLE Persons (
PersonID int,
LastName varchar(255),
FirstName varchar(255),
Address varchar(255),
City varchar(255)
);

```
mysql> create table student01
    -> (
    -> SID int,
    -> SNAME varchar(20),
    -> COURSE varchar(10),
    -> GRADE varchar(10),
    -> REMARK varchar(10)
    ->  );
Query OK, 0 rows affected (0.02 sec)

mysql> desc student01;
+--------+-------------+------+-----+---------+-------+
| Field  | Type        | Null | Key | Default | Extra |
+--------+-------------+------+-----+---------+-------+
| SID    | int         | YES  |     | NULL    |       |
| SNAME  | varchar(20) | YES  |     | NULL    |       |
| COURSE | varchar(10) | YES  |     | NULL    |       |
| GRADE  | varchar(10) | YES  |     | NULL    |       |
| REMARK | varchar(10) | YES  |     | NULL    |       |
+--------+-------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

Using ALTER statement:

Syntax:

ALTER TABLE table_name
ADD column_name datatype;

Example:

ALTER TABLE Customers
ADD Email varchar(255);

```
mysql> alter table student01 add City varchar(10);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> Select * from student01;
+------+--------+--------+-------+--------+------+
| SID  | SNAME  | COURSE | GRADE | REMARK | City |
+------+--------+--------+-------+--------+------+
|    1 | ALSTON | BSC-IT | A     | REMARK | NULL |
+------+--------+--------+-------+--------+------+
1 row in set (0.00 sec)
```

Using DROP statement:

Syntax:

DROP object object_name

Examples:

DROP TABLE table_name;

table_name: Name of the table to be deleted.

DROP DATABASE database_name;

database_name: Name of the database to be deleted.

Using TRUNCATE statement:

Syntax:

TRUNCATE TABLE

{ database_name.schema_name.table_name | schema_name.table_name | table_name }

[WITH (PARTITIONS ({ <partition_number_expression> | <range> }

[, ...n]))]

[;]

<range> ::=

<partition_number_expression> TO <partition_number_expression>

```
mysql> alter table student01 drop column GRADE;
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> desc student01;
+--------+-------------+------+-----+---------+-------+
| Field  | Type        | Null | Key | Default | Extra |
+--------+-------------+------+-----+---------+-------+
| SID    | int         | YES  |     | NULL    |       |
| SNAME  | varchar(20) | YES  |     | NULL    |       |
| COURSE | varchar(10) | YES  |     | NULL    |       |
| REMARK | varchar(10) | YES  |     | NULL    |       |
| age    | int         | YES  |     | NULL    |       |
+--------+-------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

Using RENAME statement:

There may be occasions in which you want to change some of the variable names in your SAS data set. To do so, you'll want to use the RENAME= option. As its name suggests, the RENAME= option allows you to change the variable names within a SAS data set.

Syntax:

RENAME = (old1=new1 old2=new2 .... oldk=newk);

where old1, old2, ... oldk are the variable names as they appear in the data set that precedes the RENAME= option, and new1, new2, ..., newk are the corresponding new variable names.

```
mysql> RENAME TABLE student01 TO student;
Query OK, 0 rows affected (0.01 sec)
```

Experiment no – 4

Aim: Manipulating data

a. Using INSERT statement

b. Using UPDATE statement

c. Using DELETE statement

d. Using SELECT statement

Using INSERT statement:

The INSERT command is used to store data in tables. The INSERT command is often used in higher-level programming languages such as Visual Basic.NET or C++ as an embedded SQL command; however,this command can also be executed at the SQL*PLUS prompt in command mode.There are two different forms of the INSERT command. The first form is used if a new row will have a value inserted into each column of the row. The second form of the INSERT command is used to insert rows where some of the column data is unknown or defaulted from another business logic.This form of the INSERT command requires that you specify column names for which data are being stored.

Syntax:

```
INSERT INTO table (column1 name, column2 name, . . .)
VALUES (column1 value, column2 value, . . .);
```

Using UPDATE statement:

The *UPDATE statement* changes data in existing rows either by adding new data or modifying existing data.

This example uses the UPDATE statement to standardize the country field to be Canada for all records in the Publishers table.

```
UPDATE Publishers
SET country = 'Canada'
```

This example increases the royalty amount by 10% for those royalty amounts between 10 and 20.

```
UPDATE roysched
SET royalty = royalty + (royalty * .10)
WHERE royalty BETWEEN 10 and 20
```

Including subqueries in an UPDATE statement

The employees from the Employees table who were hired by the publisher in 2010 are given a promotion to the highest job level for their job type. This is what the UPDATE statement would look like.

UPDATE Employees
SET job_lvl =
(SELECT max_lvl FROM jobs
WHERE employee.job_id = jobs.job_id)
WHERE DATEPART(year, employee.hire_date) = 2010

```
mysql> update student01 set SNAME ='David' where SID=1;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> Select * from student01;
+------+-------+--------+-------+--------+
| SID  | SNAME | COURSE | GRADE | REMARK |
+------+-------+--------+-------+--------+
|    1 | David | BSC-IT | A     | REMARK |
+------+-------+--------+-------+--------+
1 row in set (0.00 sec)
```

Using DELETE statement:

The *DELETE statement* removes rows from a record set. DELETE names the table or view that holds the rows that will be deleted and only one table or row may be listed at a time. WHERE is a standard WHERE clause that limits the deletion to select records.

The DELETE syntax looks like this.

DELETE [FROM] {table_name | view_name }
[WHERE clause]

The rules for the DELETE statement are:

1. If you omit a WHERE clause, all rows in the table are removed (except for indexes, the table, constraints).
2. DELETE cannot be used with a view that has a FROM clause naming more than one table. (Delete can affect only one base table at a time.)

What follows are three different DELETE statements that can be used.

1. Deleting all rows from a table.

DELETE
FROM Discounts

2. Deleting selected rows:

DELETE
FROM Sales
WHERE stor_id = ‘6380’

3. Deleting rows based on a value in a subquery:

DELETE FROM Sales
WHERE title_id IN
(SELECT title_id FROM Books WHERE type = ‘mod_cook’)

```
mysql> delete from student01;
Query OK, 2 rows affected (0.01 sec)

mysql> SELECT * FROM student01;
Empty set (0.00 sec)

mysql>
```

Using SELECT statement:

The SELECT statement, or command, allows the user to extract data from tables, based on specific criteria. It is processed according to the following sequence:

SELECT DISTINCT item(s)
FROM table(s)
WHERE predicate
GROUP BY field(s)
ORDER BY fields

We can use the SELECT statement to generate an employee phone list from the table as follows:

```
mysql> INSERT INTO student01 VALUES(1,'ALSTON','BSC-IT','A','REMARK');
Query OK, 1 row affected (0.00 sec)

mysql> Select * from student01;
+------+--------+--------+-------+--------+
| SID  | SNAME  | COURSE | GRADE | REMARK |
+------+--------+--------+-------+--------+
|    1 | ALSTON | BSC-IT | A     | REMARK |
+------+--------+--------+-------+--------+
1 row in set (0.00 sec)
```

Experiment no – 5

Aim: Creating and managing the tables

a. Creating table with constraints: NOTNULL, UNIQUE, PRIMARY KEY, FOREIGN KEY

NOT NULL Constraint:

By default, a column can hold NULL values.

The NOT NULL constraint enforces a column to NOT accept NULL values.

This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

SQL NOT NULL on CREATE TABLE

The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values when the "Persons" table is created:

Example

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255) NOT NULL,
   Age int
);
```

SQL NOT NULL on ALTER TABLE

To create a NOT NULL constraint on the "Age" column when the "Persons" table is already created, use the following SQL:

Syntax:

```
ALTER TABLE Persons
ALTER COLUMN Age int NOT NULL;
```

```
mysql> CREATE TABLE Student
    -> (        s_id int NOT NULL,
    ->          name varchar(60),
    ->          age  int
    -> );
Query OK, 0 rows affected (0.01 sec)

mysql> desc Student;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| s_id  | int         | NO   |     | NULL    |       |
| name  | varchar(60) | YES  |     | NULL    |       |
| age   | int         | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

UNIQUE Constraint:

The following SQL creates a UNIQUE constraint on the "ID" column when the "Persons" table is created:

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons (
   ID int NOT NULL UNIQUE,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int
);
```

MySQL:

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int,
   UNIQUE (ID)
);
```

```
mysql> ALTER TABLE Student
    -> MODIFY age INT NOT NULL UNIQUE;
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> desc Student;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| s_id  | int         | NO   |     | NULL    |       |
| name  | varchar(60) | YES  |     | NULL    |       |
| age   | int         | NO   | PRI | NULL    |       |
+-------+-------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

PRIMARY KEY:

SQL PRIMARY KEY on CREATE TABLE:

The following SQL creates a PRIMARY KEY on the "ID" column when the "Persons" table is created:

MySQL:

```
CREATE TABLE Persons (
   ID int NOT NULL,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int,
   PRIMARY KEY (ID)
);
```

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons (
   ID int NOT NULL PRIMARY KEY,
   LastName varchar(255) NOT NULL,
   FirstName varchar(255),
   Age int
);
```

```
mysql> ALTER table Student
    -> ADD PRIMARY KEY (s_id);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> desc Student;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| s_id  | int         | NO   | PRI | NULL    |       |
| name  | varchar(60) | YES  |     | NULL    |       |
| age   | int         | NO   | UNI | NULL    |       |
+-------+-------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

FOREIGN KEY:

The following SQL creates a FOREIGN KEY on the "PersonID" column when the "Orders" table is created:

MySQL:

```
CREATE TABLE Orders (
  OrderID int NOT NULL,
  OrderNumber int NOT NULL,
  PersonID int,
  PRIMARY KEY (OrderID),
  FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

SQL Server / Oracle / MS Access:

```
CREATE TABLE Orders (
  OrderID int NOT NULL PRIMARY KEY,
  OrderNumber int NOT NULL,
  PersonID int FOREIGN KEY REFERENCES Persons(PersonID)
);
```

```
mysql> CREATE TABLE categories(
    ->     categoryId INT AUTO_INCREMENT PRIMARY KEY,
    ->     categoryName VARCHAR(100) NOT NULL
    -> ) ENGINE=INNODB;
Query OK, 0 rows affected (0.01 sec)

mysql>
mysql> CREATE TABLE products(
    ->     productId INT AUTO_INCREMENT PRIMARY KEY,
    ->     productName varchar(100) not null,
    ->     categoryId INT,
    ->     CONSTRAINT fk_category
    ->     FOREIGN KEY (categoryId)
    ->         REFERENCES categories(categoryId)
    -> ) ENGINE=INNODB;
Query OK, 0 rows affected (0.01 sec)
```

Experiment no – 6

Aim: Restricting and sorting data

a. Using DISTINCT,IN, AS, SORT,LIKE,ISNULL, OR

b. Using Group By, Having clause, Order By clause

The IN Operator:

The IN operator is used to test a column value in a given set of value. If the column can be equated to any of the values from the given set, the condition is validated. The condition defined using the IN operator is also known as the membership condition.

For example, the WHERE condition SALARY IN (1500, 3000, 2500) in a SELECT query will restrict the rows where salary is either of 1500, 3000 or 2500**.**

```
mysql> Select * from student01
    -> where SID IN (1);
+------+--------+--------+-------+--------+
| SID  | SNAME  | COURSE | GRADE | REMARK |
+------+--------+--------+-------+--------+
|    1 | Alston | BSCIT  | A     | BEST   |
+------+--------+--------+-------+--------+
1 row in set (0.00 sec)
```

The LIKE Operator:

The LIKE operator is used for pattern matching and wildcard searches in a SELECT query. If a portion of the column value is unknown, wildcard can be used to substitute the unknown part. It uses wildcard operators to build up the search string, thus search is known as Wildcard search. These two operators are Percentile ('%') and Underscore ('_'). Underscore ('_') substitutes a single character while percentile ('%') replaces more than one characters. They can be used in combination as well.

For example, the below SELECT query lists the first names of those employees whose last name starts with 'SA'.

SELECT first_name

FROM employees

WHERE last_name LIKE 'SA%';

```
mysql> select * from student01 where SNAME LIKE '%AL%';
+------+--------+--------+-------+--------+
| SID  | SNAME  | COURSE | GRADE | REMARK |
+------+--------+--------+-------+--------+
|    1 | Alston | BSCIT  | A     | BEST   |
|    2 | Kunal  | BCOM   | A     | GOOD   |
+------+--------+--------+-------+--------+
2 rows in set (0.00 sec)
```

OR:

The OR operator links more than one condition in a WHERE clause and returns TRUE if either of the condition returns true. Suppose that your organizational manager's requirements change a bit. Another employee listing is needed, but in this listing the employees should: (1) be female or, (2) have a last name that begins with the letter "T" or a letter that comes later in the alphabet. The result table should be sorted by employee last name. In this situation either of the two conditions can be met in order to satisfy the query. Female employees should be listed along with employees having a name that satisfies the second condition.

```
mysql> select * from student01 WHERE SID=2 OR SID=3;
+------+-----------+--------+-------+--------+
| SID  | SNAME     | COURSE | GRADE | REMARK |
+------+-----------+--------+-------+--------+
|    2 | Kunal     | BCOM   | A     | GOOD   |
|    3 | Sarabjeet | BSCIT  | B     | GOOD   |
+------+-----------+--------+-------+--------+
2 rows in set (0.00 sec)
```

DISTINCT:

The SELECT DISTINCT statement is used to return only distinct (different) values.

Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.

SELECT DISTINCT Syntax

SELECT DISTINCT *column1, column2, ...*
FROM *table_name*;

```
mysql> select Distinct Remark from student01;
+--------+
| Remark |
+--------+
| BEST   |
| GOOD   |
+--------+
2 rows in set (0.00 sec)
```

ORDER BY clause:

You use the ORDER BY clause to sort the records in the resulting list. Use ASC to sort the results in ascending order and DESC to sort the results in descending order.

For example, with ASC:

SELECT *

FROM Employees

ORDER BY HireDate ASC

And with DESC:

SELECT *

FROM Books

ORDER BY type, price DESC

```
mysql> Select * from student01 ORDER by SNAME;
+------+-----------+--------+-------+--------+
| SID  | SNAME     | COURSE | GRADE | REMARK |
+------+-----------+--------+-------+--------+
|    1 | Alston    | BSCIT  | A     | BEST   |
|    2 | Kunal     | BCOM   | A     | GOOD   |
|    3 | Sarabjeet | BSCIT  | B     | GOOD   |
+------+-----------+--------+-------+--------+
3 rows in set (0.00 sec)
```

HAVING:

The HAVING clause can be used to restrict rows. It is similar to the WHERE condition except HAVING can include the aggregate function; the WHERE cannot do this.

The HAVING clause behaves like the WHERE clause, but is applicable to groups. In this example, we use the HAVING clause to exclude the groups with the province ‘BC’.

SELECT au_fname AS ‘Author”s First Name’, province as ‘Province’

FROM Authors

GROUP BY au_fname, province

HAVING province <> ‘BC’

```
mysql> select SNAME,SID from student01 SNAME HAVING SID>1;
+-----------+------+
| SNAME     | SID  |
+-----------+------+
| Kunal     |    2 |
| Sarabjeet |    3 |
+-----------+------+
2 rows in set (0.00 sec)
```

Experiment no – 7

Aim: Aggregate and Mathematical functions:

a. **AVG,MIN,MAX,SUM,COUNT**

b. **ABS,SQRT,ROUND,TRUNCATE,SIGN,POWER,MOD,FLOOR,CEIL**

Theory:

Aggregate functions in SQL

As the Basic SQL Tutorial points out, SQL is excellent at aggregating data the way you might in a pivot table in Excel. You will use aggregate functions all the time, so it's important to get comfortable with them. The functions themselves are the same ones you will find in Excel or any other analytics program. We'll cover them individually in the next few lessons. Here's a quick preview:

- COUNT counts how many rows are in a particular column.
- SUM adds together all the values in a particular column.
- MIN and MAX return the lowest and highest values in a particular column, respectively.
- AVG calculates the average of a group of selected values.

The Basic SQL Tutorial also pointed out that arithmetic operators only perform operations across rows. Aggregate functions are used to perform operations across entire columns (which could include millions of rows of data or more).

ABS:

```
mysql> Select ABS(-25000)
    -> ;
+-------------+
| ABS(-25000) |
+-------------+
|       25000 |
+-------------+
1 row in set (0.00 sec)
```

SUM:

```
mysql> select SUM(FEES) from student01;
+-----------+
| SUM(FEES) |
+-----------+
|     73000 |
+-----------+
1 row in set (0.00 sec)
```

SQRT:

```
mysql> SElect sqrt(16) from dual;
+----------+
| sqrt(16) |
+----------+
|        4 |
+----------+
1 row in set (0.00 sec)
```

SIGN:

```
mysql> Select sign(4.56) from dual;
+------------+
| sign(4.56) |
+------------+
|          1 |
+------------+
1 row in set (0.01 sec)
```

ROUND:

```
mysql> Select Round(14.565) from dual;
+---------------+
| Round(14.565) |
+---------------+
|            15 |
+---------------+
1 row in set (0.00 sec)
```

POWER:

```
mysql> select power(2,4) from Dual;
+------------+
| power(2,4) |
+------------+
|         16 |
+------------+
1 row in set (0.00 sec)
```

MOD:

```
mysql> select MOD(4,2) from Dual;
+----------+
| MOD(4,2) |
+----------+
|        0 |
+----------+
1 row in set (0.00 sec)
```

MIN and MAX:

```
mysql> select MIN(FEES) from student01;
+-----------+
| MIN(FEES) |
+-----------+
|     20000 |
+-----------+
1 row in set (0.00 sec)

mysql> select MAX(FEES) from student01;
+-----------+
| MAX(FEES) |
+-----------+
|     28000 |
+-----------+
1 row in set (0.00 sec)
```

Floor:

```
mysql> select Floor(12.12) from Dual;
+--------------+
| Floor(12.12) |
+--------------+
|           12 |
+--------------+
1 row in set (0.00 sec)
```

Count:

```
mysql> select count(*) from student01;
+----------+
| count(*) |
+----------+
|        3 |
+----------+
1 row in set (0.00 sec)
```

CEIL:

```
mysql> select CEIL(5.78) from Dual;
+------------+
| CEIL(5.78) |
+------------+
|          6 |
+------------+
1 row in set (0.00 sec)
```

AVG:

```
mysql> select avg(FEES) from student01;
+------------+
| avg(FEES)  |
+------------+
| 24333.3333 |
+------------+
1 row in set (0.01 sec)
```

Experiment no-8

Aim: Views and Joins: For a given set of relation tables perform the following

a. Creating view

b. Dropping view

c. Selecting from a view

Theory:

Following are the different view options in MySQL:

1. DROP: A view/virtual table can be deleted using the DROP VIEW command. If we want to delete the customer_archive table.

Syntax:

DROP VIEW customer_archive;

```
mysql> select * from student;
+------+-----------+--------+-------+--------+-------+
| SID  | SNAME     | COURSE | GRADE | REMARK | FEES  |
+------+-----------+--------+-------+--------+-------+
|    3 | Sarabjeet | BCOM   | A     | GOOD   | 28000 |
|    4 | Harsh     | BCOM   | A     | GOOD   | 27000 |
+------+-----------+--------+-------+--------+-------+
2 rows in set (0.00 sec)

mysql> drop view student;
Query OK, 0 rows affected (0.01 sec)
```

2. CREATE OR REPLACE: With CREATE OR REPLACE VIEW command we can update a view / virtual table.

Syntax:

CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2,….
FROM table;

Syntax:

CREATE VIEW view-name AS
SELECT column1, column2, column3, …
FROM table_name1 INNER JOIN table_name2
ON table_name1.column = table_name2.column;

```
mysql> create view student as select * from student01 where COURSE='BCOM';
Query OK, 0 rows affected (0.01 sec)

mysql> select * from student;
+------+-----------+--------+-------+--------+-------+
| SID  | SNAME     | COURSE | GRADE | REMARK | FEES  |
+------+-----------+--------+-------+--------+-------+
|    3 | Sarabjeet | BCOM   | A     | GOOD   | 28000 |
|    4 | Harsh     | BCOM   | A     | GOOD   | 27000 |
+------+-----------+--------+-------+--------+-------+
2 rows in set (0.00 sec)
```

Experiment no-9

Aim: Database trigger

a. **Using CREATE OR REPLACE TRIGGER**

Theory:

Triggers in MySQL are stored programs similar to procedures. These can be created on a table, schema, view and database that are associated with an event and whenever an event occurs the respective trigger is invoked.

Triggers are, in fact, written to be executed in response to any of the following events-

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE)
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

You can create a trigger using the CREATE TRIGGER Statement.

Experiment no-10

Aim: Index

a. Create index

b. Drop index

Theory:

A database index is a data structure that improves the speed of operations in a table. Indexes can be created using one or more columns, providing the basis for both rapid random lookups and efficient ordering of access to records.

While creating index, it should be taken into consideration which all columns will be used to make SQL queries and create one or more indexes on those columns.

Practically, indexes are also a type of tables, which keep primary key or index field and a pointer to each record into the actual table.

The users cannot see the indexes, they are just used to speed up queries and will be used by the Database Search Engine to locate records very fast.

The INSERT and UPDATE statements take more time on tables having indexes, whereas the SELECT statements become fast on those tables. The reason is that while doing insert or update, a database needs to insert or update the index values as well.

```
mysql> Create Index IDX_RNO on stud(SNO);
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> Drop Index IDX_RNO on stud;
Query OK, 0 rows affected (0.01 sec)
Records: 0  Duplicates: 0  Warnings: 0
```