Level 1

Create Employee Table : Create a table Employee with colomns EmpID, EmpName, Department, Joiningdate, and salary.

Create (Theory) : Create is a DDL (Data defination language) command used to create new objects like database, tables, views, indexes, functions, and procedures. It defines the structure and schema for the database objects.

Procedure :

```
CREATE TABLE EMPlayee (
    EmpID INT AUTO_Increment Primary Key,
    EmpName VARCHAR(100) NOT NULL,
    Department VARCHAR(50) NOT NULL,
    Joining Date DATE NOT NULL,
    Salary Decimal (10, 2) NOT NULL
);
```

Output:

conclusion:

In this procedure, a Employee table was created using SQL. The table include key colomns where EmpID is the primary key that uniquely identifies each employee, while fields like EmpName & Department store essential details of employees. This procedure demonstrate the fundamental concept ~~of~~ table creation in SQL, ~~white~~ i which is essential for managing and structuring data in relational database.

Insert Employee Records: Insert five records into the Employee table with varying salaries and joining dates.

Insert (theory): An SQL Insert statements adds one or more records to any single table in a relational database. (INSERT INTO)

procedure:

```
INSERT INTO Employee (EmpName, Department,
JoiningDate, Salary) VALUES
('Anjan', 'IT', '2021-01-15', 50000.00),
('Prachy', 'HR', '2021-08-01', 45000.00),
('shreya', 'Finance', '2019-04-12', 60000.00),
('Risap', Marketing', '2018-11-23', 55000.00),
('Sunil', 'Sales', '2017-09-10', 47000.00);
```

Output:

Conclusion: The five records of Employees with varying salaries and joining dates is sucessfully inserted into Employee table.

Retrive Employee Data: Write a query to retrive all employee who joined after 2020-01-01.

SELECT * (Theory):

Select - The SQL SELECT statement is used to query and retrive data from a database. It is commonly used SQL statement which allows you to select specific column and row of data from database objects.

* - all

SELECT * - select all

Procedure :

```
SELECT * FROM Employee
where JoiningDate> '2020-01-01';
```

Output :

Conclusion : In this procedure this query retrive all employee who have joined after 2020-01-01

Sorting Employee Data : Write a query to display employee names sorted by their salary in ascending order.

Theory : Select EmpName, Salary : This retrives Employees name and salary columns from the Employee table.

ORDER BY SALARY ASC : This sorts the result based on the salary column in ascending order (ASC)

Procedure :

```
SELECT EmpName, Salary
From Employee
ORDER BY Salary Asc;
```

Output :

Conclusion :

In this procedure Employee names are display according to their salary in acending order.

Select Speific Columns: Write a query to retrive only EmpName and Department from the Employee table.

Theory: In the select query if we have to ~~pri~~ retrive / display data from any specific ~~tab~~ column we have to Specify / include the particular column ~~name~~ name in the select query.

procedure :

```
SELECT EmpName, Department
From Employee;
```

Output:

Conclusion:

This ~~se~~ SQL query in above procedure only retrive ~~da~~ data from EmpName and Department from the Employee table.

Level 2:

Table for Suppliers : Create table Suppliers with column Supplier ID, Supplier Name, City, and Phone Number. Insert five records.

Theory : Supplier table is used to store information about suppliers and their column is:

Supplier ID : A unique identifier (PK)
Supplier Name : name of the supplier.
City : location
phone Number : Contact details.

procedure : Creating table

~~CREATE TABLE AUTO_INCREMENT PRIMARY KEY,~~

```
CREATE TABLE Supplier(
 SupplierID INT AUTO_INCREMENT PRIMARY KEY,
 SupplierName VARCHAR(100) NOT NULL,
 City VARCHAR(100) NOT NULL,
 PhoneNumber VARCHAR(15) NOT NULL );
```

Inserting Records.

```
INSERT INTO Suppliers (SupplierName, City, PhoneNumber)
Values
 ('Himalayan Supplies', 'Kathmandu', '9801234567'),
 ('Everest Traders', 'Pokhara', '9812345678'),
 ('Nepal Mart', 'Bhaktapur', '9823456789'),
 ('Mountain Goods', 'Kathmandu', '9834567890'),
 ('Valley Distributors', 'Pokhara', '9845678901');
```

Output :

Conclusion: In this procedure, Supplier table was created with four colums and five records were successfully inserted into the table.

Filter by City: Write a query to display suppliers located in "Kathmandu".

Theory: filtering records from DB is a curcial part of data retrival. To display suppliers located in Kathmandu, we use the where clause in SQL.

where clause - allows us to specify a condition to filter.

Procedure:

```
Select * FROM Suppliers
where City = 'Kathmandu';
```

Output:

Conclusion:

This query sucessfully retrives all supplier records from the Supplier table where the city is Kathmandu. where clause can be used to filter data based on specific conditions.

Update Supplier Info: Update the phone number of a supplier of a supplier with Supplier ID = 8

Theory:

UPDATE - This statement in SQL is used to modify existing records in a table. To update a specific we use where clause

where clause - clause to target the row that matches the given condition.

Procedure: UPDATE Supplier SET
phoneNumber = '9856789012'
where supplierID = 3

Output:

Conclusion:
This query successfully updates the phoneNumber of the supplier with supplierID = 3. This method allows businesses to maintain accurate and up-to-date supplier information in the database.

~~Delete old suppliers: write a query to count the total number of suppliers in the from~~

Delete old suppliers: write a query to delete suppliers from the city "Pokhara".

Theory:

DELETE: This statement in SQL is used to remove one or more rows from a table

To target specific rows for deletion, the where clause is used to set a condition, the where clause is used to set a condition.

procedure:

```
DELETE FROM Supplier
where city = 'Pokhara';
```

Output: Before

Conclusion:

This Query successfully deletes all suppliers from the supplier table where the city is 'Pokhara'. This demonstrate the use of the DELETE statement ~~with~~ with the where clause.

Count Suppliers: Write a query to count the total number of suppliers in the database.

Theory:

Count() : This function in SQL is used to return the total number of rows in a table or the number of rows that satisfy a specific condition.

AS : The AS command is used to rename a column or table with an ailias. helps to shorten the queries and improves code readability.

* : all

procedure :

```
SELECT COUNT(*) AS TotalSuppliers
FROM Supplier;
```

Output:

Conclusion:

This query sucessfully counts and displays the total number of suppliers in the Supplier table. This function is an essential tool for summarizing and analysing data in SQL database.

Level 3.

Table for Orders: Create a table Orders with columns OrdersID, CustomerID, OrderDate, and TOtalAMount.

Theory:

The Create table statement in SQL is used to create a new table in database. We are creating table named orders with the following columns:

OrderID - Unique identifier (PK)
CustomerID - Represent the customer placing the order.
OrderDate - store the date the order was placed.
Total Amount - stores the total monetary value of the order.

Procedure:

```
CREATE TABLE Orders(
    OrderID INT Primary key,
    CustomerID INT NOT NULL,
    OrderDate DATE,
    Total AMount DECIMAL (10,2));
```

Output:

10

Conclusion: This query successfully creates the "orders" table with the specified column. Each column has been defined with appropriate data type, ensuring data integrity & proper structure within the database.

Customer Table: Create another table customer with columns, customerID, customerName, and City. Insert at least five records.

Theory: CREATE TABLE statement is used to create a table in SQL. In this task, we are creating a customer tables with columns.
CustomerID - unique identifier for each customer.
CustomerName - Name of the customer.
City - City where the customer resides.

procedure:

```
CREATE TABLE ~~CUSTOMER~~ Customer (
CustomerID int Primary key,
CustomerName VARCHAR(50) NOT NULL,
City VARCHAR(50) NOT NULL );
```

Inserting values.

```
INSERT INTO Customer (CustomerID, CUSto
merName, City) VALUES
(1, 'Anjan', 'Bhaktapur'),
(2, 'Risap', 'Kathmandu'),
(3, 'Sunil', 'Lalitpur'),
~~(4, 'Prachy', 'Kathmandu'),~~ ~~('Bhaktapur')~~
(4, 'Prachy', 'Bhaktapur'),
(5, 'Shreya', 'Kathmandu');
```

Output:

Conclusion: The Customer table is sucessfully created with ~~Cole~~specified column and 5 records of customer is also inserted sucessfully.

Join Query: Write a query to display all orders with the corresponding customer name and city by joining orders and customer.

Theory: The join operation in SQL is used to combine data from two or more tables based on a related colum. A Join helps retrive meaningful information by merging rows that satisfy a specific condition.

In this case:

↳ we are joining the orders table with customer table.

↳ common column in both table : CustomerID

Select - specifes the column to retrive.

From Order : specifies the pk (orders) to start the query.

INNER JOIN : Joins the Customer table with the Order table.

ON orders . Customer ID = Customer . CustomerID;

Defines the condition for the join, where the CustomerID in both tables matches.

procedure :

```
SELECT Orders.OrderID, Orders.OrderDate,
Orders.TotalAmount, Customer.CoustomerNome,
Customer.city
From Orders
INNER JOIN ~~CUSTOM~~ Customer
ON Orders.CustomerID = Customer.CustomerID;
```

Output :

Conclusion :

This query sucessfully retrives all orders along with the customer name and city using the Inner Join. By joining both table on the customerID, we link orders to their customers for meaningful result.

Total Sales : Write a query to calculate the total sales (Total Amount) for each customer.

Theory : Sum() : function calculates the total of a numeric column.

Group by : clause, to group the totals by customer

By joining the order table with the customer table, we can display the customer Name instead of customerId.

Procedure :

```
SELECT
    Customer.CustomerName,
    SUM(Orders.TotalAmount) As TotalSales
FROM Orders
INNER JOIN Customer
ON Orders.CustomerID = Customer.CustomerID
GROUP BY Customer.CustomerName;
```

Output:

Conclusion:

This query calculates the total sales for each customer by summing up the TotalAmount from the Orders table. The Group by clause ensures the totals are calculated seperately for each customer.

MOST frequent Customers : Write a query to find customers who have placed more than 2 orders.

Theory:

Count() : function calculates the number of orders for each customer.

Group By : clause groups the order by CustomerID (or Customer Name).

Having : clause filters customers whose order count exceeds 2.

Procedure:

```
SELECT
    Customer.CustomerName,
    COUNT(Order.OrderID) As TOtalOrders.
FROM Orders
INNER JOIN ~~CUSTOMER~~ Customer
ON orders.CustomerID = Customer.CustomerID
GROUP BY Customer.CustomerName
HAVING COUNT(Order.OrderID) > 2;
```

(4) Output:

conclusion:
This query identifies customers who have placed more than 2 orders by counting their orders using the Count() function. Having clause filter result to include only those customer whose order count exceeds 2. Result display customer Name and total no. of orders.

Level_4

Normalization Task : Given a table Inventory (ProductID, Product Name, SupplierName, SupplierCity, stock), normalize it to 3NF and create the new tables.

Theory

Normalization : is the process of organizating data to reduce redundancy and improve data integrity.

To normalize the Inventory table to 3NF, follow these steps:

1NF : Ensure atomic values (no repeating groups or multi valued attributes).

2NF : Remove partial dependencies (when non-key attributes depend on part of a composite key).

3NF : Remove transitive dependencies (when non-key attributes depend on other non-key attributes).

Here supplies ID is extra taken to avoid redundancy while SupplierName can create issues with duplication.

15

Procedure:

```
CREATE TABLE Supplier (
    SupplierID INT PRIMARY KEY
    SupplierName Varchar (100,
    SupplierCity varchar (100) );
```

```
INSERT INTO Supplier (SupplierID, SupplierName
    SupplierCity) VALUES
    (1, 'ABC Suppliers','Kathmandu'),
    (2, 'Global Traders', 'Pokhara');
    (3, 'Fresh Mart', 'Bhaktapur');
    (4, 'Eco Goods', 'Kathmandu'),
    (5, 'Techworld', 'Pokhara');
```

```
CREATE TABLE Product(
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100),
    SupplierID INT,
    Stock INT,
    FOREIGN KEY (SupplierID) REFERENCES
    Supplier (SupplierID) );
```

```
INSERT INTO Product(ProductID, ProductName
    ,SupplierID, Stock) VALUES
    (1, 'Soap', 1, 50),
    (2, Shampoo', 2, 30),
    (3, 'Detergent', 3, 40),
    (4, 'Toothpaste', 4, 25),
    (5, 'Facewash', 5, 60);
```

16

Output:

Conclusion:

1 NF: Ensure atomicity.

2 NF: Removed partial dependencies by creating seperate tables Producs & suppliers.

3 NF: Ensure no transitive dependencies.

The final database schema includes two tables Product & Supplies linked by SupplierID. This structure eliminates redundancy, reduce anomalies, and maintain data integrity.

17 Trigger for sales : Create a trigger on the sales table to update inventory stock whenever a new sale is recorded. also explain in simple

Theory :

Trigger is a special database mechanism that automatically ~~per~~ performs an action in response to an event on a table (like INSERT, UPDATE, or DELETE).

DELIMITER → Helps to create the stored procedure.
↳ saves great time if there's a query that you write often.
↳ like a period end of the sentence $$ or //
changing delimiter temporarily.

For this question :

↳ we have created inventory table with some item and sales table when we insert data sold item ~~item~~s in Sales the trigger will automatically set and updates the Inventory table.

Procedure :

```
DELIMITER //

    CREATE TRIGGER Update_Stock_After_Sale
    AFTER INSERT ON Sales
    FOR EACH ROW
    BEGIN
        UPDATE ~~Inventory~~ Inventory
        SET stock = stock - NEW.QuantitySold
        WHERE ~~PRO~~
          ProductID = NEW.ProductID;
    END;
//
DELIMITER;
```

Output:

Conclusion:

The trigger is used to automate stock updates in the Inventory table when a sale is made. This process ensures that the stock remains accurate, & we don't need to manually upate the Inventory table.

Stored Procedure for Payroll : Create a table Payroll with EmpID, BasicSalary, Allowances and Deductions. Write a stored procedure to calculate the net salary (Basic Salary + Allowances - Deductions).

Theory: A stored procedure is a resuable block of SQL code that can be executed with a simple call. It allows for automation & simplification of repetitive database task.

Here we have created Payroll table and Inserted some values in which EmpID will act as primarky key which will used to calculate specific employee net salary.

procedure: Creating table.

```
CREATE TABLE Payroll (
    EmpID INT PRIMARY KEY,
    Basic salary   DECIMAL (10,2),
    Allowances     DECIMAL (10,2),
    Deductions     DECIMAL (10,2) );
```

Inserting values:

```
INSERT INTO Payroll values
(1, 50000, 5000, 2000),
(2, 45000, 3000, 1500),
(3, 60000, 10000, 4000);
```

Creating Stored procedure

```
DELIMITER //

    CREATE PROCEDURE CalculateNetSalary
    (IN emp_id INT, OUT net_salary DECIMAL
      (10,2))
    BEGIN
```

```
    SELECT (BasicSalary + Allowances - Deductions)
      INTO  net_salary
      FROM  Payroll
      where EmpID = empid_id;
  END;
//
DELIMITER;
```

Calling the stored procedure.

```
CALL CalculateNetSalary (1, @NetSalary);
SELECT @NetSalary;
```

Output:

Conclusion:

The Payroll table stores employee salary details and the stored procedure calculate Net salary will calculate the net salary of any employee based on EmpID Using CALL command, we can easily calculate the salary of any employee, and it return the NetSalary as output.

21 ER- Diagram Task: Design an ER diagram for a college database with entities like Student, Course, faculty, and Department. Convert it into relational tables.

Theory:

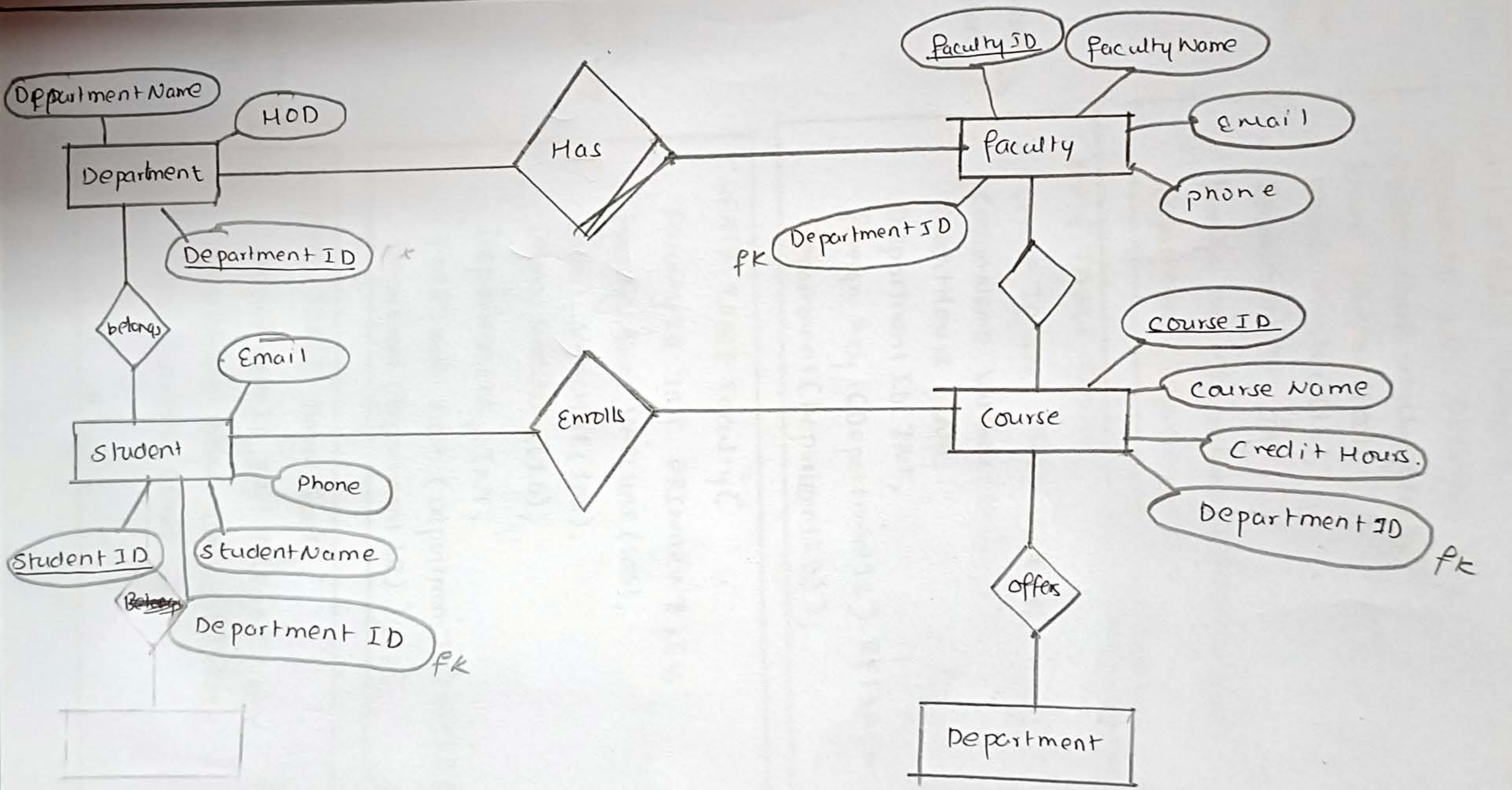
An ER (Entity Relationship) diagram is a visual representation of a database's structure. It uses specific symbols to show entities, their attributes, and the relationships between them.

Entity - is something having physical form.

Attributes - Properties of entities.

Relationship - Association between entities.

ER- Diagram (Procedure).


Department Name
HOD
Department
Department ID
Has
Faculty ID
Faculty Name
Faculty
email
phone
fk
Department ID
belongs
Email
Enrolls
Course ID
Course Name
Course
Credit Hours.
Department ID
fk
Student
Phone
Student ID
Student Name
Department ID
fk
offers
Department

23

Procedure:

Student table →

```
CREATE TABLE Student (
    StudentID INT PRIMARY KEY,
    Student Name Varchar (100),
    Email Varchar (100),
    Phone varchar (15),
    Department ID INT,
    Foreign key (Department ID) References
    Department (DepartmentID) );
```

Course table →

```
CREATE TABLE CourseC
    Course ID INT PRIMARY KEY,
    CourseName Varchar (100),
    CreditHours INT,
    DepartmentID INT,
    Foreign key CDepartmentID ) REFERENCES
    Department (DepartmentID) );
```

Faculty Table →

```
CREATE TABLE FacultyC
    FacultyID INT PRIMARY KEY,
    FacultyName VARCHAR (100),
    Email VARCHAR (100),
    Phone VARCHAR (15),
    DepartmentID . INT,
    FOREIGN KEY (DepartmentID) REFERENCES
    Department (Department ID) );
```

Department Table →

```
CREATE TABLE Department (
    DepartmentID INT PRIMARY KEY,
    Department Name VARCHAR (100),
    HOD VARCHAR (100)
);
```

24 Output:

~~Conclu~~

Conclusion:

The ER helps visualize the structure of the college database. The relational tables convert the ER diagram into SQL table structures, allowing for achal implementation in relational database.

25 Transaction Management : Simulate a banking transaction where an amount is transferred from one acount to another. Ensure rollback in case of insufficient funds.

Theory

Stored procedure - The transaction logic is encapsulated within a stored procedure. This is necessary to use the if statement and other ~~conto~~ control flow logic.

Transaction control: The start transaction, commit, and rollback statement are used within the stored procedure to ensure the transaction is either committed or rolled bucle as need.

Here we have use a local variable v_balance

The stored procedure checks if John Doe has sufficient funds, if he does, the money is transferred to Jane Smith's account, and the transaction is committed. if he does not, the transaction is rolled back, and an error message is returned.

procedure:

```
CREATE TABLE Accounts(
    AccountID INT PRIMARY KEY,
    AccountHolderName VARCHAR(50),
    Balance DECIMAL(10,2));
```

```
INSERT INTO Accounts VALUE
(1, 'John Doe', 5000.00),
(2, 'Jane Smith', 3000.00);
```

```
DELIMITER $$

CREATE PROCEDURE TransferAmount()
BEGIN

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

```
START TRANSACTION;

SELECT Balance
INTO v_balance
FROM Accounts
WHERE AccountID = 1
FOR UPDATE;
```

```
IF v_balance >= 1000 THEN

    UPDATE Accounts
    SET Balance = Balance - 1000
    WHERE AccountID = 1;

     UPDATE Accounts
      SET Balance = Balance + 1000
      WHERE AccountID = 2;

      COMMIT;
      SELECT 'Transaction Successful: Amount Transfer-
        - red ' As Message;
    ELSE
        ROLLBACK;
        SELECT 'Transaction Failed: Insufficient
          Balance' As Message;
    END IF;
  END $$
DELIMITER;

CALL TransferAmount()
```

Output:

Conclusion: The query sucessfully simulate a banking transaction where an amount is transferred from one account to another.

Level 5

Dynamic SQl : Write a dynamic SQl query to retrive table name from the database based on user input.

Theory

Dynamic SQl allows flexibility to interact with database objects dynamically. This approach particularly useful for queries where parts of the statement, such as table name or column names, are not known until runtime.

i) Define the user input

ii) Use dynamic SQl to query the information schema.

iii) Execute the query.

Procedure :

```
SET @ user_input = 'Marks';

SET @ query = CONCAT(

        'Select TABLE_NAME FROM information
        _schema.tables WHERE TABLE_SCHEMA
```

```
= \'level5\' AND TABLE_NAME LIKE \'%.',
@user_input,
    '%.1' ;'
);
PREPARE stmt FROM @query;
EXECUTE stmt;
DEALLOCATE PREPAE stmt;
```

Output:

conclusion:

Dynamic SQL allows flexibility to interact with database objects dynamically. The example above retrieves all tables name containing a user specified keyword.

5-2

Data Integrity: Create a table with columns StudentID, Subject, Marks. Add a constraint to ensure marks cannot exceed 100.

Theory:

Here we are creating a table ~~where~~ of ~~student~~ MARKS in which there are 3 column studonID, subject, marks.
In this table we are putting a check constraints. It ~~will help~~ is the one way to enfore data integrity.

Constraint : rules applied to table column.

CHECK : It is a constraint is used to limit the value range that can be placed in a column.

Procedure :
```
CREATE TABLE Marks (
        StudentID INT,
        Subject VARCHAR(50)
        Marks INT,
        CONSTRAINT chk_marks CHECK CMarks
        <= 100) );
```

Here, try inserting marks greater than 100 it will raise an error due to constraint.

Output:

Conclusion: This query sucessfully create table and puts the constraints in marks column.

5.3

User defined function: Write a function to calculate the average order value for a given CustomerID in the Orders table

Theory

A user-defined function (UDF) in SQL is a function created by the user to encapsulate specific logic and can be used within queries, just like built-in SQL functions.

↳ A stored program you can pass parameter into to return a value.

↳ This function can be called to calculate the average order value for any given customer

procedure:

```
DELIMITER $$

CREATE FUNCTION GetAverageOrderValue (customer_id
 INT) RETURNS DECIMAL(10,2)
  DETERMINISTIC
BEGIN
    DECLARE avg_order_value DECIMAL(10,2);
    SELECT AVG(OrderAmount)
    INTO avg_order_value
     FROM Orders
     WHERE CustomerID = Costomer_id;

    RETURN avg_order_value;
END $$

DELIMITER;
```

Output:

5.3

Conclusion:
This query sucessfully creates function GetAverageOrderValue Calculates the average order value for a specific customer.

Nested Subqueries: Write a query to find employees from the Employee table who earn more than the average salary of their department.

~~Nested subque~~

Theory

Nested subquery is a query within another query. The outer query can reference the result of the inner query (subquery) to filter or process the data further.

Consider a employee table having Employee id, EmployeeName, Salary, DepartmentID.

procedure:

```
SELECT ~~EMPLOYEE~~ EmployeeID, EmployeeName,
Salary, DepartmentID From Employee e
where Salary > (
    SELECT AVG(Salary)
    FROM Employee
    WHERE DepartmentID = e.DepartmentID
);
```

5.4

Output:

Conclusion:

This query ~~example~~ sucessfully demonstrate the nested subquery to find employees from the Employee table who earns more than the average salary in their respective departments. The soln demonstrate how SQL allows us to compare values dynamically by using subqueries.

Level 6

Hotel Management System: Create tables Room(RoomID, Type, Price) and Booking(BookingID, CustomerName, RoomID, CheckInDate, CheckOutDate). write a query to find all available rooms for a specific data range.

Theory:

For a Hotel management system, we require two tables.

Room

↳ stores details about hotel rooms like (RoomID, type, price)

Booking

↳ stores booking details like (BookingID, CustomerName, RoomID, checkInDate, checkoutdate)

To find available rooms:

Procedure: -- creating Room & Booking table

```
CREATE TABLE Room(
    RoomID INT PRIMARY KEY,
    Type VARCHAR(50),
    Price DECIMAL(10,2));

CREATE TABLE Booking(
    BookingID INT PRIMARY KEY,
    CustomerName VARCHAR(50),
    RoomID INT,
    CheckInDate DATE,
    CheckOutDate DATE,
    FOREIGN KEY (RoomID) REFERENCES
    Room(RoomID)
    );
```

6-1

```
-- Inserting data on both tables
INSERT INTO Room VALUES
(1, 'single', 1000.00),
(2, 'Double', 2000.00),
(3, 'suite', 3000.00);

INSERT INTO Booking (BookingID, CustomerName,
 RoomID, CheckInDate, CheckOutDate) VALUES
(1, 'Raven', 1, '2024-12-20', '2024-12-25'),
(2, 'Ronny', 2, '2024-12-22', '2024-12-26');
```

```
-- Finding Available Rooms
SELECT Room.RoomID, Room.Type, Room.Price
 FROM Room
 WHERE Room.RoomID NOT INC
  SELECT RoomID
  FROM Booking
  WHERE '2024-12-21' < CheckOutDate
  AND
  '2024-12-23' > CheckInDate);
```

Output :

Conclusion:

This query identifies all rooms that are not booked within the specified date range by using a subquery to exclude rooms overlapping with existing bookings.

6-2

School Database : Create table Teacher (TeacherID, Name, Subject) and classes ~~Taught by~~ (Class ID, Class Name, Teacher ID). . write a query to display all classes taught by a specific teacher.

Theory

A school Database consist of two tables :

Teacher

↳ stores information about teachers, including the subjects they teach.

class

↳ stores information about classes and their respective teachers.

Here we can use a join query to connect the Teacher and Class tables through the TeacherID column.

Procedure :

```
Create TABLE Teacher (
    TeacherID  INT Primary KEY,
    Name   VARCHAR(50),
    Subject VARCHAR(50) );

Create TABLE Class (
    ClassID  INT  PRIMARY KEY,
    ClassName VARCHAR(50),
    Teacher ID INT,
    FOREIGN  KEY (TeacherID ) REFERENCES
     Teacher (TeacherID);
```

6-2

-- Inserting values.

```
INSERT INTO Teacher VALUES
    (1, 'Arun', 'DBMS'),
    (2, 'Ayesh', 'Operating System'),
    (3, 'Susan', 'Numerical Methods'),
    (4, 'Arun', 'Software engineering'),
    (5, 'Raju', 'scripting language');

INSERT INTO class VALUES
    (1, 'DBMS-class-A', 1),
    (2, 'OS-class-B', 2),
    (3, 'NM-class-C', 3),
    (4, 'SE-class-D', 4),
    (5, 'SL-class-E', 5);
```

-- Query to Display classes Taught by a specific Teacher.

```
SELECT class.classID, class.ClassName, Teacher.
Name, Teacher.Name, Teacher.Subject)
  FROM class
    JOIN Teacher ON Class.TeacherID =
      Teacher.TeacherID WHERE Teacher.Name
      = 'Arun';
```

Output :

6.2

Conclusion:

This query sucessfully fetches ~~th~~ all classes taught by the specified teacher (e.g. Arun) by performing a join on the Teacher and class tables.

Online Store Discounts : Create a table Discounts (ProductID, DiscountRate) and Products (ProductID, ProductName, Price). write a query to calculate the final price after applying discounts for all products.

Theory:

An online store database typically includes two tables:

Discounts

↳ stores information about the discount rate for each product.

Products:

↳ stores product details such as ID, name, and price.

Final price = price - (Price * DiscountRate / 100).

procedure: - - Creating Discount & Products table.

~~product ID II~~

```
CREATE TABLE DISCOUNTS (
    ProductID INT ~~PR~~ PRIMARY KEY,
    DiscountRate ~~Decim~~ Decimal (5, 2)
);

CREATE TABLE Products (
    ProductID INT Primary KEY,
    ~~Dis~~ ProductName VARCHAR (100),
    Price DECIMAL (10, 2)
);
```

6-3

-- Inserting data

```
INSERT INTO Discounts VALUES
 (1, 10.00),
 (2, 15.00),
 (3, 0.00),
  (4, 20.00),
  (5, 5.00);

INSERT INTO Products VALUES
  (1, 'Laptop', 50000.00),
  (2, 'smartphone', 30000.00),
  (3, 'Tablet', 20000.00),
  (4, 'smartwatch', 10000.00),
  (5, 'Headphones', 5000.00);
```

-- Query to calculate final price.

```
SELECT
    Products.ProductID,
    Products.ProductName,
    Products.Price As OrginalPrice
    Discounts.DiscountRate,
(Products.Price - (Products.Price * Discounts.
 DiscountRate / 100) As FinalPrice)
FROM
   Products
LEFT JOIN
    Discounts
ON
  products.ProductID = Discounts.ProductID;
```

Output:

Conclusion:

This query joins the products table with the Discounts table using the ProductID column. It calculates the final price for each product by considering the discount rate.

Level 7

Create a table "WORKSHOP" with the following fields: (Name of THE_PARTICIPANT, INSTITUTION, DESIGNATION, CITY, PARIICIPANT_ID, FEE_PAID (YES/NO) AMOUNT, ACCOMMODATION_REQUIRED (YES/NO), MOB_NO, EMAIL, ADDRESS, CITY, STATE)

PROCEDURE:

```
CREATE TABLE WORKSHOPC
PARTICIPATION_ID INT PRIMARY KEY,
Name_OF_THE_PARTICIPANT VARCHAR(100),
INSTITUION VARCHAR(150),
DESIGNATION VARCHAR(100),
CITY VARCHAR(50),
FEE_PAID VARCHAR (3) CHECK (FEE_PAID
IN('YES','NO')),
```

7-1

```
AMOUNT  DECIMAL(10,2),
ACCOMMODATION_REQUIRED VARCHAR(3) CHECK
(ACCOMMODATION_REQUIRED IN ('YES','NO')),
 MOB_NO VARCHAR(15),
 EMAIL  VARCHAR(100),
 ADDRESS VARCHAR(255),
STATE   VARCHAR(50)
);
```

- INSERTING DATA

```
INSERT INTO WORKSHOP VALUES
(1, 'Anjan ', 'Tribhuvan University', 'Professor',
  'Kathmandu', 'YES', 1000.00, 'YES' ;9801234567',
   'anjan@example.com', 'DubarMarg ', 'Baymati''),
(2, 'Prachy', 'Patan Campus', 'student', 'Lalitpur',
 'No' , 0.00, 'No' , '9812345678', 'prachy@example
  .com';'Patan Dhoka', 'Bagmati'),
(3, 'Kriti', 'Pokhara University', 'Lecturer', 'Pokhara',
 'YES', 1200.00, 'YES', 9823456789', 'Kriti@example
  .com', 'Lakeside', 'Gandaki'),
(4, 'Sunil', 'Kathmandu University', 'Professor',
  'Dhulikhel', 'YES', 1500.00, 'YES', '9845678901',
  'sunil@example.com', 'KU Road', 'Bagmati'),
(5, 'Shreya', 'IOE Pulchowk', 'student', 'Lalitpur',
  'NO', 0.00, 'NO', 0.00, 'NO', '9867890123',
   shreya@example.com', 'pulchowk', 'Bagmati'),
(6, 'Suraj', Tribhuvan University', 'Lecturer',
   'Bhaktapur', 'YES', 800.00, 'NO', '98111223344',
```

-7-1

```
'suraj@example-com', 'Kamalbinayak', 'Bagmati'),
(7, 'Risap', 'Kathmandu University', 'student', 'Dhulikhel',
 'NO', 0.00, 'YES', '9811223344', 'risap@example.
  com', 'Banepa', 'Bagmati'),
(8, 'Raven', 'Pokhara University', 'Lecturer', 'Pokhara',
 'YES', 1200.00, 'NO', '9822334455', 'raven@
  example.com', 'Pokhara Lakeside', 'Gandaki'),
(9, 'Riya', 'Patan Campus', 'Professor', 'lalitpur',
 'YES', 1000.00, 'YES', '9845544332', 'riya@
  example.com', 'Mangal Bazer', 'Bagmati'),
(10, 'Rohan', 'Tribhuvan University', 'student',
 'Kathmandu', 'NO', 0.00, 'NO', '9801289876',
 'rohan@example.com', 'New Road', 'Bagmati');
```

Theory

The table WORKSHOP stores details about participants of a workshop. Fields include participant information (name, institution, deasignation, contact details, etc.)

Here Participation_ID is the primarly key to uniquely identify each participant.

Output

7-2

For the above table create in Q.1 Answer the following queries using SQL.

a) To display all participant with designations "Professor".

Theory:

To display all participant with designation "Professor" we can user select statement with where clause.

procedure:

```
SELECT * FROM WORKSHOP
WHERE DESIGNATION = "Professor";
```

output:

conclusion: This query sucessfully display all participant with designation "Professor".

b) To display all the Participants who are form "Bagmati pradesh"

Theory: This query displays participants who are from bagmati pradesh..we can used select statement with where clause.

procedure:

```
SELECT * FROM WORKSHOP
WHERE STATE = 'Bagmati';
```

7-2

Output:

Conclusion: This query sucessfully displays participant whose are from 'Bagmati' Pradesh.

c) Display the total amount paid by all the participants.

Theory:

sum() : calculates the total sum of column.

procedure:

```
Select SUM(AMOUNT) AS Total_Amount_Paid
FROM WORKSHOP;
```

Output:

Conclusion:

This query sucessfully display the total amount paid by all the participants.

d) Display all the participants who require accomono-dation.

Theory

This query attempts to display all the participant who requère accommodation. we use select statement with where clause.

Procedure:

```
SELECT *
 FROM WORKSHOP
 WHERE ACCOMODATION_REQUIRED = 'YES';
```

Output:

Conclusion

This query sucessfully display participant whose ACCOMODATION_REQUIRED = 'YES';