Ex. No.: 1	CASE STUDY SUBMISSION FOR ER DIAGRAMS
Date: 22/07/2024	

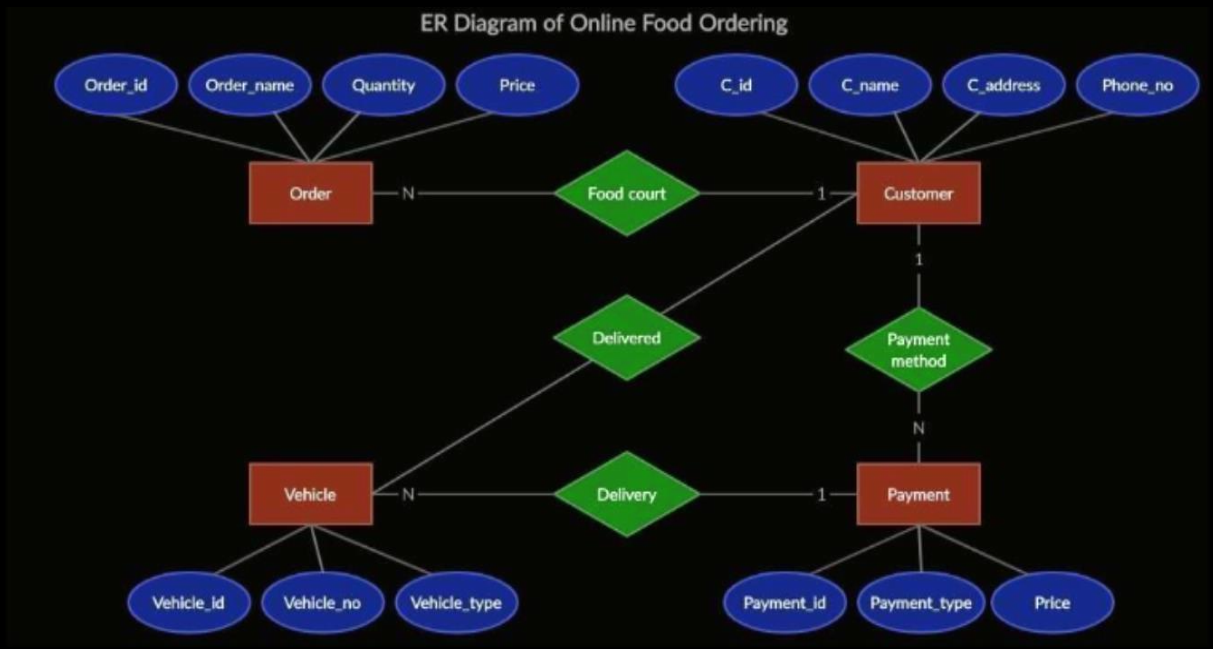
Aim:

To understand the basic concepts and terminology related to DBMS and Relational Database Design using ER Diagram.

Procedure:

1. Select the application / system
2. Represent all possible Entity sets
3. Decide the attributes of all entity set
4. Provide the relationship between entity sets
5. Design the ER Diagram with necessary symbols.

Sample ER- Diagram:



Result:

Hence, the ER Diagram drawn.

Ex. No.: 2	SQL QUERIES FOR STUDENTS DATABASE
Date: 22/07/2024	

Aim:

To create a Student table in a database to store student information, including registration number, name, marks, total, and age. The program demonstrates basic operations like creating a table, inserting records, altering the table structure, and updating data.

Procedure:

1. **Create the Student Table**:
    - The table Student is created with the following fields:
        - Regno (Registration Number) - Primary Key
        - Name - Student's Name
        - Mark1 and Mark2 - Marks scored by the student in two subjects
        - Total - Total of Mark1 and Mark2
2. **Insert Data into the Table**:
    - Five students' records are inserted with registration numbers, names, marks, and total marks.
3. **Select All Data**:
    - Display all the records in the table to view the inserted data.
4. **Alter the Table to Add an Age Column**:
    - Adds a new column Age to the Student table to store students' ages.
5. **Update the Age Data**:
    - Sets the age for all students to 21.
    - Updates specific ages for students PRIYA and ALBERT as 23 and 22, respectively.
6. **Select All Data Again**:
    - Displays the updated table with the Age column and modified data.

Source code:

```
-- Create Student table
CREATE TABLE Student (
    Regno VARCHAR(50) PRIMARY KEY,
    Name VARCHAR(50),
    Mark1 INT,
    Mark2 INT,
    Total INT
);
-- Insert data into Student table
INSERT INTO Student (Regno, Name, Mark1, Mark2, Total) VALUES ('01', 'MOHANA', 40, 45, 85);
INSERT INTO Student (Regno, Name, Mark1, Mark2, Total) VALUES ('02', 'ALBERT', 45, 45, 90);
INSERT INTO Student (Regno, Name, Mark1, Mark2, Total) VALUES ('03', 'VASU', 30, 50, 80);
INSERT INTO Student (Regno, Name, Mark1, Mark2, Total) VALUES ('04', 'PRIYA', 37, 42, 79);
INSERT INTO Student (Regno, Name, Mark1, Mark2, Total) VALUES ('05', 'JEEVA', 50, 45, 95);
-- Display all records
SELECT * FROM Student;
-- Alter table to add Age column
ALTER TABLE Student ADD Age INT;
-- Display updated table structure
SELECT * FROM Student;
-- Update Age data
UPDATE Student SET Age = 21;
UPDATE Student SET Age = 23 WHERE Name = 'PRIYA';
UPDATE Student SET Age = 22 WHERE Name = 'ALBERT';
-- Display final records
SELECT * FROM Student;
```

Output:

1. **Initial Table Creation and Data Insertion:**

   After inserting the student records, the table will look like this:

```
+-------+--------+-------+-------+-------+------+
| Regno | Name   | Mark1 | Mark2 | Total |
+-------+--------+-------+-------+-------+------+
| 01    | MOHANA |    40 |    45 |    85 |
| 02    | ALBERT |    45 |    45 |    90 |
| 03    | VASU   |    30 |    50 |    80 |
| 04    | PRIYA  |    37 |    42 |    79 |
| 05    | JEEVA  |    50 |    45 |    95 |
+-------+--------+-------+-------+-------+
```

2. **After Altering the Table to Add the Age Column:**

   The Age column is added, but initially, it will contain NULL values since no age data is set yet.

```
+-------+--------+-------+-------+-------+------+
| Regno | Name   | Mark1 | Mark2 | Total | Age  |
+-------+--------+-------+-------+-------+------+
| 01    | MOHANA |    40 |    45 |    85 | NULL |
| 02    | ALBERT |    45 |    45 |    90 | NULL |
| 03    | VASU   |    30 |    50 |    80 | NULL |
| 04    | PRIYA  |    37 |    42 |    79 | NULL |
| 05    | JEEVA  |    50 |    45 |    95 | NULL |
+-------+--------+-------+-------+-------+------+
```

3. **After Updating the Age for All Students:**

   Sets all students' ages to 21, then modifies PRIYA and ALBERT's ages.

```
+-------+--------+-------+-------+-------+------+
| Regno | Name   | Mark1 | Mark2 | Total | Age  |
+-------+--------+-------+-------+-------+------+
| 01    | MOHANA |    40 |    45 |    85 |   21 |
| 02    | ALBERT |    45 |    45 |    90 |   22 |
| 03    | VASU   |    30 |    50 |    80 |   21 |
| 04    | PRIYA  |    37 |    42 |    79 |   23 |
| 05    | JEEVA  |    50 |    45 |    95 |   21 |
+-------+--------+-------+-------+-------+------+
```

Result:

Hence, the program accurately creates the Student table, inserts the records, alters the structure to add the Age column, and updates the values as specified. The final output correctly reflects all changes and updates, confirming the intended functionality.

Ex. No.: 3	SQL QUERIES FOR EMPLOYEE DATABASE
Date: 29/07/2024	

Aim:

To create an Employee table to store employee information, including employee ID, name, designation, salary, and age. The program demonstrates creating a table, inserting data, altering the table, updating specific records, and finally deleting all records.

Procedure:

1. **Create the Employee Table**:
    - A table named Employee is created with the following fields:
        - Emp_id - Employee ID (Primary Key)
        - Emp_name - Employee Name
        - Designation - Job Title
        - Salary - Employee's Salary
2. **Insert Records into the Table**:
    - Five employee records are inserted, each with an ID, name, designation, and salary.
3. **Select All Data**:
    - Displays all the current records in the Employee table.
4. **Alter the Table to Add an Age Column**:
    - Adds a new column Age to store the age of employees.
5. **Update Age Data**:
    - Sets the age for specific employees based on their names.
6. **Display the Updated Table**:
    - Shows the final state of the table with the new Age data.
7. **Delete All Records**:
    - Deletes all records from the Employee table, leaving it empty.

Source Code:

```
-- Create Employee table
CREATE TABLE Employee (
    Emp_id VARCHAR(50) PRIMARY KEY,
    Emp_name VARCHAR(50),
    Designation VARCHAR(50),
    Salary VARCHAR(50)
);
-- Insert records into Employee table
INSERT INTO Employee VALUES ('1', 'Andrew', 'Manager', '35000');
INSERT INTO Employee VALUES ('2', 'Aslam', 'Manager', '35000');
INSERT INTO Employee VALUES ('3', 'Gopika', 'Analyst', '25000');
INSERT INTO Employee VALUES ('4', 'Pradeep', 'Clerk', '10000');
INSERT INTO Employee VALUES ('5', 'Jeeva', 'Salesman', '12000');

-- Display all records
SELECT * FROM Employee;

-- Alter table to add Age column
ALTER TABLE Employee ADD Age VARCHAR(50);

-- Display updated table structure
SELECT * FROM Employee;

-- Update Age data for specific employees
UPDATE Employee SET Age = '45' WHERE Emp_name = 'Andrew';
UPDATE Employee SET Age = '48' WHERE Emp_name = 'Aslam';
UPDATE Employee SET Age = '30' WHERE Emp_name = 'Gopika';
UPDATE Employee SET Age = '35' WHERE Emp_name = 'Clare'; -- Note: No
employee named 'Clare' exists.
UPDATE Employee SET Age = '38' WHERE Emp_name = 'Jeeva';

-- Display the final updated records
SELECT * FROM Employee;
```

```
-- Delete all records from the Employee table
DELETE FROM Employee;
```

Output:

1. **Initial Table Creation and Data Insertion:**

   After inserting the employee records, the table will display:

```
+--------+----------+-------------+--------+
| Emp_id | Emp_name | Designation | Salary |
+--------+----------+-------------+--------+
| 1      | Andrew   | Manager     | 35000  |
| 2      | Aslam    | Manager     | 35000  |
| 3      | Gopika   | Analyst     | 25000  |
| 4      | Pradeep  | Clerk       | 10000  |
| 5      | Jeeva    | Salesman    | 12000  |
+--------+----------+-------------+--------+
```

2. **After Adding the Age Column:**

   The Age column is added but initially contains NULL values.

```
+--------+----------+-------------+--------+------+
| Emp_id | Emp_name | Designation | Salary | Age  |
+--------+----------+-------------+--------+------+
| 1      | Andrew   | Manager     | 35000  | NULL |
| 2      | Aslam    | Manager     | 35000  | NULL |
| 3      | Gopika   | Analyst     | 25000  | NULL |
| 4      | Pradeep  | Clerk       | 10000  | NULL |
| 5      | Jeeva    | Salesman    | 12000  | NULL |
+--------+----------+-------------+--------+------+
```

3. **After Updating the Age Data:**

   Updates specific employees' ages:

   Note: The update for Clare will not affect the table as no such employee exists.

```
+--------+----------+-------------+--------+------+
| Emp_id | Emp_name | Designation | Salary | Age  |
+--------+----------+-------------+--------+------+
| 1      | Andrew   | Manager     | 35000  | 45   |
| 2      | Aslam    | Manager     | 35000  | 48   |
| 3      | Gopika   | Analyst     | 25000  | 30   |
```

```
| 4      | Pradeep  | Clerk       | 10000  | NULL |
| 5      | Jeeva    | Salesman    | 12000  | 38   |
+--------+----------+-------------+--------+------+
```

4. **After Deleting All Records:**

   The table is emptied after the delete operation.

```
+--------+----------+-------------+--------+------+
| Emp_id | Emp_name | Designation | Salary | Age  |
+--------+----------+-------------+--------+------+
```

Result:

Hence, the program successfully demonstrates creating the Employee table, inserting records, altering the table to add an Age column, updating records based on specific conditions, and finally deleting all records. The operations reflect the correct structure and manipulation of data as intended.

Ex. No.: 4	EXECUTION OF JOIN OPERATIONS
Date: 05/08/2024	

Aim:

To create two tables, Customer and Orders, in a database to manage customer information and order details. The program demonstrates inserting data into these tables and performing different types of SQL joins to retrieve combined data from both tables.

Procedure:

1. **Create the Customer Table**:
    - A table named Customer is created with the following fields:
        - Id - Customer ID (Primary Key)
        - Name - Customer's Name
        - Age - Customer's Age
        - Address - Customer's Address
        - Salary - Customer's Salary
2. **Insert Records into the Customer Table**:
    - Inserts three customer records into the Customer table.
3. **Display All Data from the Customer Table**:
    - Retrieves and displays all the current records in the Customer table.
4. **Create the Orders Table**:
    - A table named Orders is created with the following fields:
        - O_id - Order ID (Primary Key)
        - Quantity - Quantity of items in the order
        - Customer_id - ID of the customer who placed the order
        - Amount - Total amount of the order
5. **Insert Records into the Orders Table**:
    - Inserts three records into the Orders table.
6. **Display All Data from the Orders Table**:
    - Retrieves and displays all the current records in the Orders table.
7. **Perform Various Joins Between Customer and Orders Tables**:
    - **INNER JOIN**: Combines records from both tables where there is a match on Customer.Id and Orders.Customer_id.
    - **LEFT JOIN**: Retrieves all records from the Customer table, and the matched records from the Orders table.

- **RIGHT JOIN**: Retrieves all records from the Orders table, and the matched records from the Customer table.
- **FULL OUTER JOIN**: Retrieves all records when there is a match in either the Customer or Orders table.

Source Code:

```
-- Create Customer table
CREATE TABLE Customer (
    Id VARCHAR(50) PRIMARY KEY,
    Name VARCHAR(50),
    Age VARCHAR(50),
    Address VARCHAR(50),
    Salary VARCHAR(50)
);
-- Insert records into Customer table
INSERT INTO Customer VALUES ('1', 'Ramesh', '32', 'Ahmedabad', '2000');
INSERT INTO Customer VALUES ('2', 'Khinal', '25', 'Delhi', '1500');
INSERT INTO Customer VALUES ('3', 'Komal', '22', 'Mumbai', '4500');

-- Display all records from Customer table
SELECT * FROM Customer;

-- Create Orders table
CREATE TABLE Orders (
    O_id VARCHAR(50) PRIMARY KEY,
    Quantity VARCHAR(50),
    Customer_id VARCHAR(50),
    Amount VARCHAR(50)
);
-- Insert records into Orders table
INSERT INTO Orders VALUES ('101', '2', '3', '3000');
INSERT INTO Orders VALUES ('102', '3', '2', '1500');
INSERT INTO Orders VALUES ('103', '3', '4', '2060'); -- Note: Customer_id '4' does not exist in Customer table

-- Display all records from Orders table
```

```
SELECT * FROM Orders;

-- Perform INNER JOIN between Customer and Orders
SELECT Id, Name, Amount, Quantity FROM Customer
INNER JOIN Orders ON Customer.Id = Orders.Customer_id;

-- Perform LEFT JOIN between Customer and Orders
SELECT Id, Name, Amount, Quantity FROM Customer
LEFT JOIN Orders ON Customer.Id = Orders.Customer_id;

-- Perform RIGHT JOIN between Customer and Orders
SELECT Id, Name, Amount, Quantity FROM Customer
RIGHT JOIN Orders ON Customer.Id = Orders.Customer_id;
-- Perform FULL OUTER JOIN between Customer and Orders
SELECT Id, Name, Amount, Quantity FROM Customer
FULL OUTER JOIN Orders ON Customer.Id = Orders.Customer_id;
```

Output:

1. **Initial Table Creation and Data Insertion (Customer):**

   After inserting records into the Customer table:

```
+----+--------+-----+-----------+--------+
| Id | Name   | Age | Address   | Salary |
+----+--------+-----+-----------+--------+
| 1  | Ramesh | 32  | Ahmedabad | 2000   |
| 2  | Khinal | 25  | Delhi     | 1500   |
| 3  | Komal  | 22  | Mumbai    | 4500   |
+----+--------+-----+-----------+--------+
```

2. **Table Creation and Data Insertion (Orders):**

   After inserting records into the Orders table:

```
+------+----------+-------------+--------+
| O_id | Quantity | Customer_id | Amount |
+------+----------+-------------+--------+
| 101  | 2        | 3           | 3000   |
| 102  | 3        | 2           | 1500   |
| 103  | 3        | 4           | 2060   | -- No matching Customer_id '4'
+------+----------+-------------+--------+
```

3. **INNER JOIN Output:**

   Displays rows where there are matches between Customer and Orders:

```
+----+--------+--------+----------+
| Id | Name   | Amount | Quantity |
+----+--------+--------+----------+
| 2  | Khinal | 1500   | 3        |
| 3  | Komal  | 3000   | 2        |
+----+--------+--------+----------+
```

4. **LEFT JOIN Output:**

   Displays all records from Customer and matching rows from Orders:

```
+----+--------+--------+----------+
| Id | Name   | Amount | Quantity |
+----+--------+--------+----------+
| 1  | Ramesh | NULL   | NULL     |
| 2  | Khinal | 1500   | 3        |
| 3  | Komal  | 3000   | 2        |
+----+--------+--------+----------+
```

5. **RIGHT JOIN Output:**

   Displays all records from Orders and matching rows from Customer:

```
+----+--------+--------+----------+
| Id | Name   | Amount | Quantity |
+----+--------+--------+----------+
| 2  | Khinal | 1500   | 3        |
| 3  | Komal  | 3000   | 2        |
| NULL | NULL | 2060   | 3        | -- Customer_id '4' does not exist
+----+--------+--------+----------+
```

6. **FULL OUTER JOIN Output:**

   Displays all records from both Customer and Orders:

```
+----+--------+--------+----------+
| Id | Name   | Amount | Quantity |
+----+--------+--------+----------+
| 1  | Ramesh | NULL   | NULL     |
| 2  | Khinal | 1500   | 3        |
| 3  | Komal  | 3000   | 2        |
| NULL | NULL | 2060   | 3        | -- Customer_id '4' does not exist
+----+--------+--------+----------+
```

Result:

Hence, the program successfully demonstrates creating the Customer and Orders tables, inserting data, and performing various types of SQL joins to analyse relationships between customers and their orders. The joins effectively illustrate how data can be combined and interpreted based on different join conditions.

Ex. No.: 5	Practice of triggers – SQL Trigger | Students Database
Date: 12/08/2024	

Aim:

To create a student table to store student information and implement a trigger that captures and displays changes to the Scholar_amt field whenever records are inserted, updated, or deleted. The program demonstrates how to use triggers in SQL and outputs the changes made to the scholarship amounts.

Procedure:

1. **Create the student Table:**
    - A table named student is created with the following fields:
        - **St_Id -** Student ID (Primary Key)
        - **St_name -** Student's Name
        - **Dept -** Student's Department
        - **Scholar_amt -** Scholarship Amount
2. **Insert Records into the student Table:**
    - Five student records are inserted into the student table.
3. **Select All Records:**
    - Retrieves and displays all current records in the student table.
4. **Create a Trigger:**
    - A trigger named amt is created that executes before any DELETE, INSERT, or UPDATE operation on the student table.
    - The trigger calculates the change in Scholar_amt and uses DBMS_OUTPUT.PUT_LINE to display the old amount, new amount, and the total amount.
5. **Update the Scholar_amt:**
    - The scholarship amount for the student with St_Id = 104 is increased by 500.
6. **Enable Output:**
    - SET SERVEROUTPUT ON; is used to enable the output display for the trigger messages.

Source Code:

```
-- Create the student table
CREATE TABLE student (
   St_Id NUMBER(10) PRIMARY KEY,
   St_name VARCHAR(20),
   Dept VARCHAR(30),
   Scholar_amt NUMBER(10)
);

-- Insert values into the student table
INSERT INTO student (St_Id, St_name, Dept, Scholar_amt) VALUES (100, 'Andrew', 'BTECH', 1200);
INSERT INTO student (St_Id, St_name, Dept, Scholar_amt) VALUES (101, 'Aslam', 'MBBS', 1100);
INSERT INTO student (St_Id, St_name, Dept, Scholar_amt) VALUES (102, 'Akash', 'BCA', 1000);
INSERT INTO student (St_Id, St_name, Dept, Scholar_amt) VALUES (103, 'Albert', 'Viscom', 900);
INSERT INTO student (St_Id, St_name, Dept, Scholar_amt) VALUES (104, 'Praveen', 'BBA', 800);

-- Select all records from the student table
SELECT * FROM student;

-- Create or replace trigger
CREATE OR REPLACE TRIGGER amt
BEFORE DELETE OR INSERT OR UPDATE ON student
FOR EACH ROW
DECLARE
   Total_amt NUMBER;
BEGIN
   IF INSERTING OR UPDATING THEN
      Total_amt := :NEW.Scholar_amt - NVL(:OLD.Scholar_amt, 0);
      DBMS_OUTPUT.PUT_LINE('OLD amount: ' || NVL(:OLD.Scholar_amt, 0));
      DBMS_OUTPUT.PUT_LINE('NEW amount: ' || :NEW.Scholar_amt);
      DBMS_OUTPUT.PUT_LINE('Total amount: ' || Total_amt);
   END IF;
```

```
END;
/


-- Update the scholar_amt for St_ID = 104
UPDATE student SET Scholar_amt = Scholar_amt + 500 WHERE St_Id = 104;

-- To check the output, run:
SET SERVEROUTPUT ON;
```

Output:

1. **Initial Table Creation and Data Insertion:**

   After inserting the records into the student table:

```
+-------+--------+--------+------------+
| St_Id | St_name | Dept   | Scholar_amt|
+-------+--------+--------+------------+
| 100   | Andrew  | BTECH  |       1200 |
| 101   | Aslam   | MBBS   |       1100 |
| 102   | Akash   | BCA    |       1000 |
| 103   | Albert  | Viscom |        900 |
| 104   | Praveen | BBA    |        800 |
+-------+--------+--------+------------+
```

2. **Trigger Output During Update**:

   When the scholarship amount for St_Id = 104 is updated, the output will be:

   OLD amount: 800

   NEW amount: 1300

   Total amount: 500

Result:

The program successfully creates the student table, inserts records, and implements a trigger that captures changes to the Scholar_amt. When the scholarship amount is updated, the trigger correctly outputs the old amount, new amount, and the total difference. This demonstrates how triggers can be used for auditing changes in a database.

Ex. No.: 6	Practice of trigger – SQL Trigger | Employee Database
Date: 19/08/2024	

Aim:

To create an employee table that stores employee information and implement a trigger that captures and displays changes to the salary field whenever records are inserted, updated, or deleted. This program illustrates how to use triggers in SQL and outputs the changes made to the salary amounts.

Procedure:

1. **Create the employee Table**:
    - A table named employee is created with the following fields:
        - eid - Employee ID (Primary Key)
        - ename - Employee Name
        - age - Employee Age
        - salary - Employee Salary
2. **Insert Records into the employee Table**:
    - Six employee records are inserted into the employee table.
3. **Select All Records**:
    - Retrieves and displays all current records in the employee table.
4. **Create a Trigger**:
    - A trigger named salary is created that executes before any DELETE, INSERT, or UPDATE operation on the employee table.
    - The trigger calculates the change in salary and uses DBMS_OUTPUT.PUT_LINE to display the old salary, new salary, and the salary difference.
5. **Update the salary**:
    - The salary for the employee with eid = 105 is increased by 500.
6. **Enable Output**:
    - SET SERVEROUTPUT ON; is used to enable the output display for the trigger messages.

Source Code:

```
-- Create the employee table
CREATE TABLE employee (
    eid NUMBER(10) PRIMARY KEY,
    ename VARCHAR(30),
    age NUMBER(3),
    salary NUMBER(10)
);

-- Insert values into the employee table
INSERT INTO employee (eid, ename, age, salary) VALUES (100, 'Senga', 27, 40000);
INSERT INTO employee (eid, ename, age, salary) VALUES (101, 'Praveen', 29, 70000);
INSERT INTO employee (eid, ename, age, salary) VALUES (102, 'Lavanya', 31, 20000);
INSERT INTO employee (eid, ename, age, salary) VALUES (103, 'Shree', 36, 50000);
INSERT INTO employee (eid, ename, age, salary) VALUES (104, 'Muthupandi', 25, 35000);
INSERT INTO employee (eid, ename, age, salary) VALUES (105, 'Dhana Lakshmi', 35,
10000);

-- Select all records from the employee table
SELECT * FROM employee;

-- Create or replace trigger
CREATE OR REPLACE TRIGGER salary
BEFORE DELETE OR INSERT OR UPDATE ON employee
FOR EACH ROW
DECLARE
   Sal_diff NUMBER;
BEGIN
   IF INSERTING OR UPDATING THEN
      Sal_diff := :NEW.salary - NVL(:OLD.salary, 0); -- Use NVL for OLD value during
INSERT
      DBMS_OUTPUT.PUT_LINE('OLD salary: ' || NVL(:OLD.salary, 0));
      DBMS_OUTPUT.PUT_LINE('NEW salary: ' || :NEW.salary);
      DBMS_OUTPUT.PUT_LINE('Salary difference: ' || Sal_diff);
   END IF;
END;
/

-- Update the salary for eid = 105
UPDATE employee SET salary = salary + 500 WHERE eid = 105;

-- To check the output, run:
SET SERVEROUTPUT ON;
```

Output:

1. **Initial Table Creation and Data Insertion:**

   After inserting the records into the employee table:

```
+------+-----------------+-----+--------+
| eid  | ename           | age | salary |
+------+-----------------+-----+--------+
| 100  | Senga           |  27 | 40000  |
| 101  | Praveen         |  29 | 70000  |
| 102  | Lavanya         |  31 | 20000  |
| 103  | Shree           |  36 | 50000  |
| 104  | Muthupandi      |  25 | 35000  |
| 105  | Dhana Lakshmi   |  35 | 10000  |
+------+-----------------+-----+--------+
```

2. **Trigger Output During Update**:

   When the salary for eid = 105 is updated, the output will be:

   OLD salary: 10000

   NEW salary: 10500

   Salary difference: 500

Result:

The program successfully creates the employee table, inserts records, and implements a trigger that captures changes to the salary. When the salary is updated, the trigger correctly outputs the old salary, new salary, and the difference. This demonstrates how triggers can be effectively used for auditing changes in a database.

Ex. No.: 7	Sample Program for Curser
Date: 29/08/2024	

Aim:

To create a student100 table that stores student information, including registration number, name, marks, total, and average. The program also includes PL/SQL blocks to fetch and display details of a specific student and all students from the table, demonstrating the use of cursors and output in PL/SQL.

Procedure:

1. **Create the student100 Table:**
   - A table named **student100** is created with the following fields:
     - **regno** - Registration number (Primary Key)
     - **name** - Student Name
     - **mark1** - First Exam Mark
     - **mark2** - Second Exam Mark
     - **total** - Total Marks
     - **avg** - Average Marks
2. **Insert Records into the student100 Table:**
   - Six student records are inserted into the student100 table.
3. **Select All Records:**
   - Retrieves and displays all current records in the student100 table.
4. **Enable Output for PL/SQL Block:**
   - SET SERVEROUTPUT ON; is used to enable output display for the DBMS messages.
5. **PL/SQL Block to Fetch Details of a Specific Student:**
   - A PL/SQL block is declared that uses a cursor to fetch details of the student with regno = 203 (Priya).
   - It outputs the details of the specified student.
6. **PL/SQL Block to Fetch and Display All Students:**
   - A second PL/SQL block is declared that uses a cursor to fetch and display details of all students in the student100 table.

Source Code:

```
-- Create the student100 table
CREATE TABLE student100 (
    regno NUMBER(10) PRIMARY KEY,
    name VARCHAR(20),
    mark1 NUMBER(5),
    mark2 NUMBER(5),
    total NUMBER(5),
    avg NUMBER(4,2)
);

-- Insert values into the student100 table
INSERT INTO student100 VALUES (200, 'Andrew', 75, 89, 164, 82.00);
INSERT INTO student100 VALUES (201, 'Vel', 92, 87, 179, 89.50);
INSERT INTO student100 VALUES (202, 'Vasu', 79, 80, 159, 79.50);
INSERT INTO student100 VALUES (203, 'Priya', 75, 70, 145, 72.50);
INSERT INTO student100 VALUES (204, 'Jeeva', 70, 67, 137, 68.50);
INSERT INTO student100 VALUES (205, 'Simbu', 73, 71, 144, 72.00);

-- Select all records from the student100 table
SELECT * FROM student100;

-- Enable output for PL/SQL block
SET SERVEROUTPUT ON;

-- PL/SQL block to fetch details for a specific student
DECLARE
    mark student100%ROWTYPE;
    CURSOR details IS SELECT * FROM student100 WHERE regno = 203;
BEGIN
    OPEN details;
    DBMS_OUTPUT.PUT_LINE('Student details');
    LOOP
        FETCH details INTO mark;
        EXIT WHEN details%NOTFOUND;  -- Exit loop if no more rows are found
        DBMS_OUTPUT.PUT_LINE('Student name: ' || mark.name);
        DBMS_OUTPUT.PUT_LINE('Mark1: ' || mark.mark1);
        DBMS_OUTPUT.PUT_LINE('Mark2: ' || mark.mark2);
        DBMS_OUTPUT.PUT_LINE('Total: ' || mark.total);
        DBMS_OUTPUT.PUT_LINE('Average: ' || mark.avg);
    END LOOP;
    CLOSE details;
    DBMS_OUTPUT.PUT_LINE('Student details displayed');
END;
/

-- PL/SQL block to fetch and display all students from student100
DECLARE
    s_regno student100.regno%TYPE;
    s_name student100.name%TYPE;
    s_mark1 student100.mark1%TYPE;
```

```
    s_mark2 student100.mark2%TYPE;
    s_total student100.total%TYPE;
    s_avg student100.avg%TYPE;
    CURSOR s_student106 IS SELECT regno, name, mark1, mark2, total, avg FROM
student100;
BEGIN
    OPEN s_student106;
    LOOP
        FETCH s_student106 INTO s_regno, s_name, s_mark1, s_mark2, s_total, s_avg;
        EXIT WHEN s_student106%NOTFOUND;   -- Exit loop if no more rows are
found
        DBMS_OUTPUT.PUT_LINE(s_regno || ' ' || s_name || ' ' || s_mark1 || ' ' || s_mark2
|| ' ' || s_total || ' ' || s_avg);
    END LOOP;
    CLOSE s_student106;
END;
/
```

Output:

1. **Initial Table Creation and Data Insertion**:

   After inserting the records into the student100 table:

   +-------+--------+-------+-------+-------+-------+

   | regno | name | mark1 | mark2 | total | avg |

   +-------+--------+-------+-------+-------+-------+

   | 200 | Andrew | 75 | 89 | 164 | 82.00 |

   | 201 | Vel | 92 | 87 | 179 | 89.50 |

   | 202 | Vasu | 79 | 80 | 159 | 79.50 |

   | 203 | Priya | 75 | 70 | 145 | 72.50 |

   | 204 | Jeeva | 70 | 67 | 137 | 68.50 |

   | 205 | Simbu | 73 | 71 | 144 | 72.00 |

   +-------+--------+-------+-------+-------+-------+

2. **Output for Specific Student (Priya)**:

   The output when fetching details for the student with regno = 203 will be:

   Student details

   Student name: Priya

   Mark1: 75

   Mark2: 70

   Total: 145

   Average: 72.50

   Student details displayed

3. **Output for All Students**:

   The output when fetching and displaying all students:

   200 Andrew 75 89 164 82.00

   201 Vel 92 87 179 89.50

   202 Vasu 79 80 159 79.50

   203 Priya 75 70 145 72.50

   204 Jeeva 70 67 137 68.50

   205 Simbu 73 71 144 72.00

Result:

The program successfully creates the student100 table, inserts student records, and retrieves specific and all student details using PL/SQL blocks with cursors. The output confirms that the program works as intended, showcasing how to use cursors and DBMS_OUTPUT for displaying results in PL/SQL.

Ex. No.: 8	Case study submission for JDBC
Date: 12/09/2024	

Aim:

To understand the concept of advanced Database Application Development using JDBC.

Procedure:

1. Create database using MySQL.
2. Implement JDBC concepts.
3. Produce the output.

Java Database Connectivity(JDBC):

JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

JDBC Connectivity:

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

- Register the Driver class
- Create connection
- Create statement

- Execute queries
- Close connection

1) Register the driver class

The forName() method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax of forName() method

public static void forName(String className)throws ClassNotFoundException

Example to register the OracleDriver class

Here, Java program is loading oracle driver to establish database connection.

```
Class.forName(&quot;oracle.jdbc.driver.OracleDriver&quot;);
try {
Class.forName(&quot;oracle.jdbc.driver.OracleDriver&quot;);
}
catch(ClassNotFoundException ex)
{
System.out.println(&quot;Error: unable to load driver class!&quot;);
System.exit(1);
}
```

2) Create the connection Object

The getConnection() method of DriverManager class is used to establish connection with the database.

Syntax:

1) public static Connection getConnection(String url) throws SQLException
2) public static Connection getConnection(String name, Strng password) throws SQLException

Example:

Connection conn=DriverManager.getConnection(“jdbc:oracle:thin:@localhost:xe”,”system”,

”password”

String URL =
"jdbc:oracle:thin:username/password@amrood:1521:EMP"; Connection
conn = DriverManager.getConnection(URL);

3) Create the Statement Object:

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

Syntax-

public Statement createStatement()throws SQLException

Example:

Statement stmt=con.createStatement();

4) Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax -public ResultSetexecuteQuery(String sql)throws SQLException

Example:

```
ResultSet rs=stmt.executeQuery(“select * from emp”);
While(rs.next())
{

System.out.println(1)+” “ +rs.getString(2));

}
```

5) Close the Connection Object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection**.**

Syntax - public void close()throws SQLException

Example:

```
con.close()
```

Example to Connect Java Application with Oracle database:

In this example, we are connecting to an Oracle database and getting data from emp table. Here, system and oracle are the username and password of the Oracle database.

```
import
java.sql.*;
class
OracleCo
n{
public static void main(String
args[]){ try{
Class.forName(&quot;oracle.jdbc.driver.OracleDriver&quot;);
Connection con=DriverManager.getConnection(
&quot;jdbc:oracle:thin:@localhost:1521:xe&quot;,&quot;system&quot;,&quot;or
acle&quot;); Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery(&quot;select * from
emp&quot;); while(rs.next())
System.out.println(rs.getInt(1)+&quot; &quot;+rs.getString(2)+&quot;
&quot;+rs.getString(3)); con.close();
}catch(Exception e){ System.out.println(e);}

}
```

Ex. No.: 9	Creating a Student Database
Date: 12/09/2024	

Aim:

To create, insert, update, retrieve, and delete records in a Student database using SQL commands and manage data in a relational table format.

Procedure:

1. Create a table named Student to store student information.

- **Columns include:**
- **StudentID:** Unique identifier for each student.
- **FirstName:** Student's first name.
- **LastName:** Student's last name.
- **Age:** Student's age.
- **Major:** Student's major field of study.

2. Insert records into the Student table:

- Add entries for a few students with unique StudentIDs.

3. Update a student's age:

- Change the age of a specific student.

4. Retrieve all records:

- Use the SELECT statement to display all records in the Student table.

5. Delete a student record:

- Remove a student with a specific StudentID.

6. Display the final table:

- **Use SELECT again to show the remaining records in the table.**

Code:

```
CREATE TABLE Student (
    StudentID VARCHAR(50) PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Age INT,
    Major VARCHAR(50)
);

INSERT INTO Student (StudentID, FirstName, LastName, Age, Major)
VALUES ('S001', 'John', 'Doe', 20, 'Computer Science');

INSERT INTO Student (StudentID, FirstName, LastName, Age, Major)
VALUES ('S002', 'Jane', 'Smith', 22, 'Mathematics');

INSERT INTO Student (StudentID, FirstName, LastName, Age, Major)
VALUES ('S003', 'Alice', 'Johnson', 21, 'Biology');

UPDATE Student
SET Age = Age + 1
WHERE StudentID = 'S001';

SELECT * FROM Student;

DELETE FROM Student
WHERE StudentID = 'S003';

SELECT * FROM Student;
```

Output:

After executing the SQL commands, the final table will look like this:

StudentID	*FirstName*	*LastName*	*Age*	*Major*
S001	John	Doe	21	Computer Science
S002	Jane	Smith	22	Mathematics

Result:

The student table was successfully created, and operations like inserting new records, updating existing records, and deleting unwanted records were performed as intended. The final state of the Student table reflects the changes.

Ex.No.: 10	Create an XML document for employee information
Date: 19/09/2024	

Aim:

To understand the concept of advanced Database Application Development using XML.

Procedure:

1. Create employee database using XML.
2. Retrieve the database information.
3. Produce the output.

XML CODE:

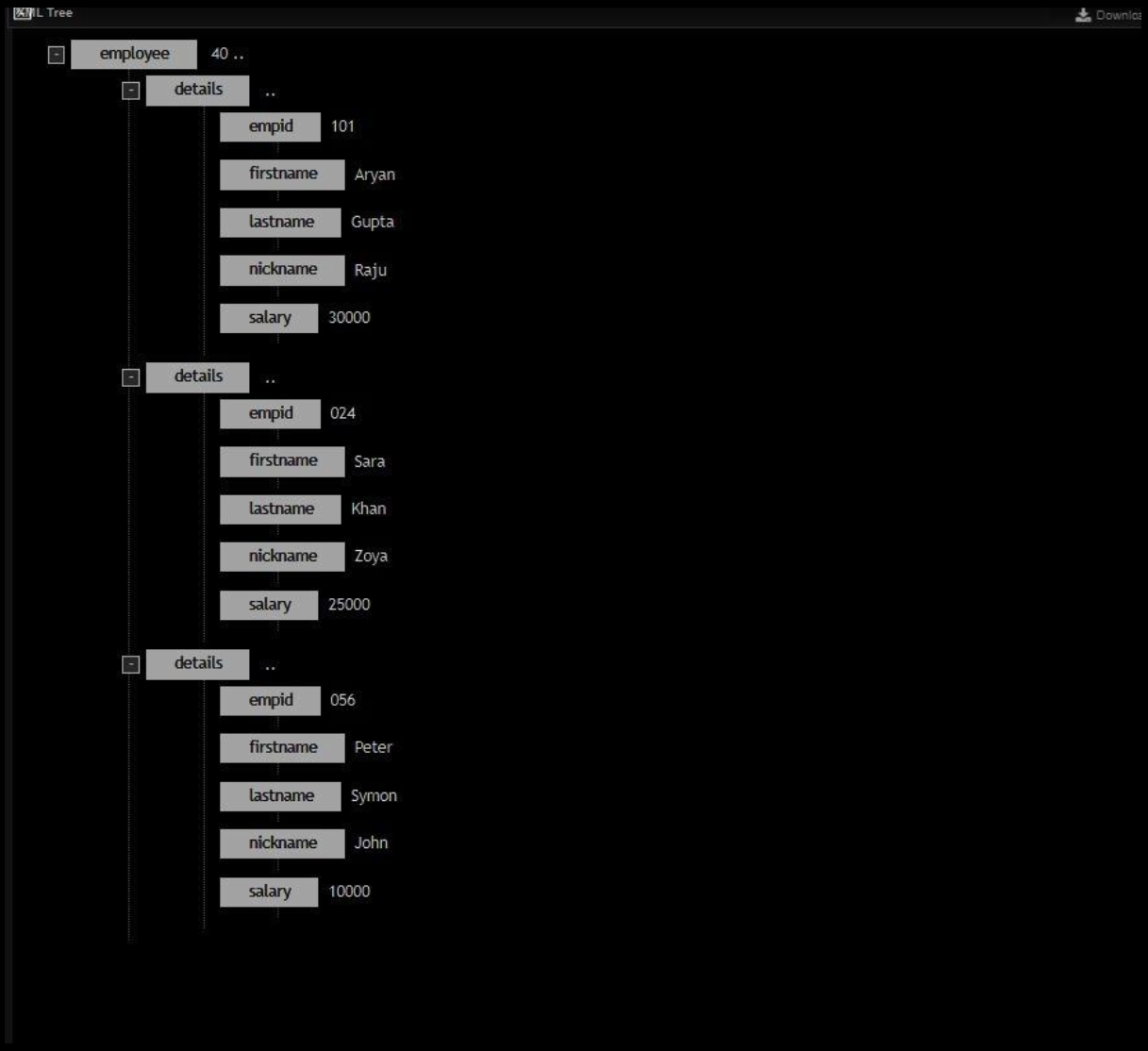
```
<?xml version = “1.0”?>
<employee>
<details>
<empid>101</empid>
<firstname>Aryan</firstname>
<lastname>Gupta</lastname>
<nickname>Raju</nickname>
<salary>30000</salary>
</details>

<details>
<empid>024</empid>
<firstname>Sara</firstname>
<lastname>Khan</lastname>
<nickname>Zoya</nickname>
<salary>25000</salary>
</details>

<details>
<empid>056</empid>
<firstname>Peter</firstname>
```

```
<lastname>Symon</lastname>
<nickname>John</nickname>
<salary>10000</salary>
</details>
</employee>
```

OUTPUT:



Ex. No.: 11	Simple Program for Join
Date: 26/09/2024	

Aim:

To demonstrate the use of different types of SQL joins to retrieve related data from multiple tables.

Procedure:

1. **Students:** Contains student information.
2. **Courses:** Contains course information.
3. **Enrollments:** Links students to the courses they are enrolled in.

Code:

```
-- Create Students table
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);

-- Create Courses table
CREATE TABLE Courses (
    CourseID INT PRIMARY KEY,
    CourseName VARCHAR(50)
);

-- Create Enrollments table
CREATE TABLE Enrollments (
    EnrollmentID INT PRIMARY KEY,
    StudentID INT,
    CourseID INT,
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
);
```

```
-- Insert data into Students
INSERT INTO Students (StudentID, FirstName, LastName) VALUES
(1, 'John', 'Doe'),
(2, 'Jane', 'Smith'),
(3, 'Alice', 'Johnson');

-- Insert data into Courses
INSERT INTO Courses (CourseID, CourseName) VALUES
(101, 'Mathematics'),
(102, 'Computer Science'),
(103, 'Biology');

-- Insert data into Enrollments
INSERT INTO Enrollments (EnrollmentID, StudentID, CourseID) VALUES
(1, 1, 101),  -- John Doe enrolled in Mathematics
(2, 1, 102),  -- John Doe enrolled in Computer Science
(3, 2, 101),  -- Jane Smith enrolled in Mathematics
(4, 3, 103);  -- Alice Johnson enrolled in Biology

-- INNER JOIN
SELECT
    Students.FirstName,
    Students.LastName,
    Courses.CourseName
FROM
    Students
INNER JOIN
    Enrollments ON Students.StudentID = Enrollments.StudentID
INNER JOIN
    Courses ON Enrollments.CourseID = Courses.CourseID;
```

-

Explanation of Joins:

1. *INNER JOIN*:
   - This query retrieves students who are enrolled in courses. Only records that have matching entries in both Students and Enrollments are returned.

2. *LEFT JOIN*:
   - This query retrieves all students, even those who are not enrolled in any courses. For students without courses, the CourseName will be NULL.

3. *RIGHT JOIN*:
   - This query retrieves all enrollments and the corresponding students. If there are courses with no enrolled students, the student's name will be NULL.

Output:

INNER JOIN Result:

FirstName	*LastName*	*CourseName*
John	Doe	Mathematics
John	Doe	Computer Science
Jane	Smith	Mathematics
Alice	Johnson	Biology

LEFT JOIN Result:

FirstName	*LastName*	*CourseName*
John	Doe	Mathematics
John	Doe	Computer Science
Jane	Smith	Mathematics
Alice	Johnson	NULL

RIGHT JOIN Result:

FirstName	*LastName*	*CourseName*

John	Doe	Mathematics
John	Doe	Computer Science
Jane	Smith	Mathematics
NULL	NULL	Biology

Result:

This example demonstrates how to set up a simple database with tables and how to use different types of joins to retrieve related data effectively. Adjust the data and queries as needed for your lab requirements!

Ex. No.: 12	Study of Normalization Techniques
Date: 03/10/2024	

Aim:

To demonstrate the normalization of a Students table through various normal forms (1NF, 2NF, 3NF, and BCNF) in SQL. This process aims to eliminate data redundancy and ensure data integrity in the database.

Procedure:

1. Create the Initial Students Table (Not Normalized):

- Define the Students table with a Subjects column that violates the First Normal Form (1NF) by allowing non-atomic values.

2. Normalize to First Normal Form (1NF):

- Create a new table Students_1NF to ensure atomic values in the Subject column.

3. Normalize to Second Normal Form (2NF):

- Create separate tables Students_2NF and Subjects_2NF to remove partial dependencies.

4. Normalize to Third Normal Form (3NF):

- Create a Departments table to eliminate transitive dependencies and a new Subjects_3NF table to link with the Departments.

5. Normalize to Boyce-Codd Normal Form (BCNF):

- Analyze the tables for BCNF compliance, which requires no non-trivial functional dependencies.

6. Analyze and Test:

- Write a query to check data integrity and ensure the relationships among tables are working as intended.

Code:

```
sql
-- Step 1: Create the initial Students table (not normalized)
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(50),
    Subjects VARCHAR(100) -- This column violates 1NF
);

-- Insert initial data
INSERT INTO Students (StudentID, Name, Subjects) VALUES
(1, 'John Doe', 'Math, Science'),
(2, 'Jane Smith', 'Math, History');

-- Step 2: Normalize to First Normal Form (1NF)
-- Create a new table for 1NF
CREATE TABLE Students_1NF (
    StudentID INT,
    Name VARCHAR(50),
    Subject VARCHAR(50), -- Atomic values
    PRIMARY KEY (StudentID, Subject)
);

-- Insert data into Students_1NF
INSERT INTO Students_1NF (StudentID, Name, Subject) VALUES
(1, 'John Doe', 'Math'),
(1, 'John Doe', 'Science'),
(2, 'Jane Smith', 'Math'),
(2, 'Jane Smith', 'History');

-- Step 3: Normalize to Second Normal Form (2NF)
-- Create separate tables for 2NF
CREATE TABLE Students_2NF (
```

```
    StudentID INT PRIMARY KEY,
    Name VARCHAR(50)
);

CREATE TABLE Subjects_2NF (
    StudentID INT,
    Subject VARCHAR(50),
    PRIMARY KEY (StudentID, Subject),
    FOREIGN KEY (StudentID) REFERENCES Students_2NF(StudentID)
);

-- Insert data into Students_2NF
INSERT INTO Students_2NF (StudentID, Name) VALUES
(1, 'John Doe'),
(2, 'Jane Smith');

-- Insert data into Subjects_2NF
INSERT INTO Subjects_2NF (StudentID, Subject) VALUES
(1, 'Math'),
(1, 'Science'),
(2, 'Math'),
(2, 'History');

-- Step 4: Normalize to Third Normal Form (3NF)
-- Create Departments table to remove transitive dependencies
CREATE TABLE Departments (
    Subject VARCHAR(50) PRIMARY KEY,
    Department VARCHAR(50)
);

-- Insert data into Departments
INSERT INTO Departments (Subject, Department) VALUES
('Math', 'Science Department'),
('Science', 'Science Department'),
('History', 'Arts Department');
```

```
-- Create new Subjects table to link with Departments
CREATE TABLE Subjects_3NF (
    Subject VARCHAR(50),
    Department VARCHAR(50),
    PRIMARY KEY (Subject),
    FOREIGN KEY (Subject) REFERENCES Departments(Subject)
);

-- Step 5: Normalize to Boyce-Codd Normal Form (BCNF)
-- Assuming that we already have Subjects and Departments structured correctly, no further action needed.

-- Step 6: Analyze and Test
-- Query to check data integrity
SELECT
    s.Name,
    sub.Subject,
    dep.Department
FROM
    Students_2NF s
JOIN
    Subjects_2NF sub ON s.StudentID = sub.StudentID
JOIN
    Departments dep ON sub.Subject = dep.Subject;
```

Output:

After running the final query, the output will display a list of students, their respective subjects, and the departments associated with those subjects. The output would look something like this:

```
+-----------+---------+--------------------+
| Name      | Subject | Department         |
+-----------+---------+--------------------+
| John Doe  | Math    | Science Department |
| John Doe  | Science | Science Department |
| Jane Smith| Math    | Science Department |
| Jane Smith| History | Arts Department    |
+-----------+---------+--------------------+
```

Result:

The final output verifies that the normalization process has been successful:

- The data is organized to eliminate redundancy.
- Each subject is linked to the appropriate department.
- The relationships between tables maintain data integrity.

Ex. No.: 13	Case Study Submission For Database Administration
Date: 10/10/2024	

Aim:

To understand the concept of database administration

Procedure:

1. Select the system / application
2. Include database administration methods.
3. Design the system for database administration.

Introduction:

Database administration is the function of managing and maintaining database management systems (DBMS) software. Mainstream DBMS software such as Oracle, IBM DB2 and Microsoft SQL Server need ongoing management. As such, corporations that use DBMS software often hire specialized information technology personnel called database administrators or DBAs.

Types of DBAs:

- System DBA
- Development DBA
- Application DBA

Sample Case Study for Faculty of Open
Education The Data Definition Subsystem:

The data definition subsystem helps defining the structure of the files in the database. Data tables are designed on relational data tables. There are eighteen data tables provided by the relational database in this study (Table-1). In the relational data table, attributes represent necessary data summarized by the course book's printing, and royalty departments. The structure of the relational database in this paper is described.

The Data Manipulation Subsystem:

The data manipulation subsystem lets us add and delete records, change field contents, and view the database. The information in a database can be viewed by using queries.

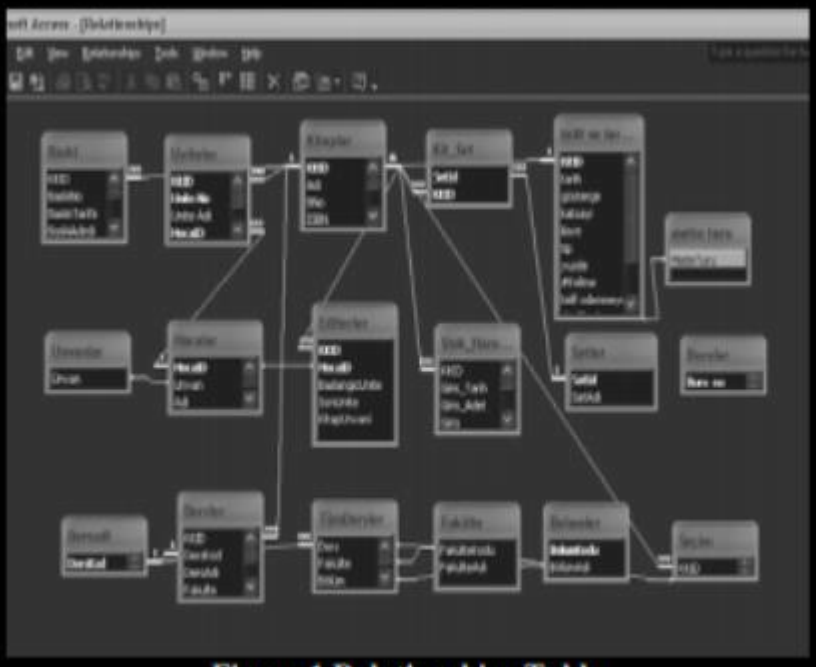


Figure.1 Relationships Table

Table-1: Data Tables

	DATA TABLES	
1.	Books	BookID, Name, Barcode Number, ISBN, publication numbers
2.	Printing	BookID, Printing number, date, number of pages
3.	Chapters	BookID, Chapter number and names, Author's ID
4.	Authors	AuthorID, title, name, surname, address, phone number
5.	Publishers	BookID, AuthorID, beginning and last chapters
6.	Titles	Title
7.	Royalty	BookID, date, pointer, coefficient, addition, text type
8.	Text type	Text type
9.	Stock	BookID, entrance and exit date, unit, explanation
10.	Book-Sets	SetID, BookID
11.	Sets	SetID, Set names
12.	Offices	OfficeID, Office name
13.	Course Names	CourseID, Course
14.	Faculties	FacultyID, Faculty name
15.	Departments	DepartmentId, Department name
16.	Courses	BookId, CourseID, Course name, Faculty, department and class
17.	Addition royalty	BookID, page number, number of words
18.	Selection	BookID, Course

The database is opening with the following start-up menu. The user has the following six main options: creation of necessary records of book's characteristics; control of book storage; preparing royalties and storing personal (authors and publishers) information, and preparing any kind of reports to see information about books. User can quit the database by clicking "Programdan Çıkış" command button.

User can add, edit, and delete records by using the following forms (Table 2):

Table-2: Database Forms

1.	New Course Book Definition
2.	Printing
3.	Courses
4.	New Course Definition / Modification of Courses
5.	Chapters
6.	Authors & Publishers Addresses
7.	Modification of Authors & Publishers
8.	Royalty
9.	Course Book's Sets Definition
10.	Stock Inputs & Outputs

For example, "Printing" form includes all the fields necessary to define course books printing information. It is used to enter new printing data into the database. An example of an input data screen is shown in Figure 3. User selects the book's name and fills out necessary fields. The user can add new course book to the database by clicking "yeni kitap Ekle" command button. He can also make modifications of existing data. The edit and delete operations are identical to the input data operations and work in the same way. In this form, there're also links for reports of printing data.

Figure. 3 Printing Form

The Database Administration Subsystem:

Storing all of the course book's information in a database has created the need for managing the database. In the relational database, data integrity and security are maintained by those who are authorized to use, update, and delete. The database administration subsystem lets us establish users of the database, specify who can update information, and develop methods for backing up the database and recovering the database in the event of a failure. For example, one form printing department could look at, but not change, information relating to the price of royalties.

Application Generation Subsystem :

Application generation subsystem contains tools that help us create and update other features
such as menus, data entry screen forms, reports, and application software. This study needs more
time to examine further development of system implementation. For example, new orders can be
taken from users, like new reports, and data entry forms.

Conclusion:

Until now, there is no problem with the database. Users can use it easily without any problem. Everyone related with these data are now uses only one database, and can reach easily. By this study, the data redundancy and integrity problems have been solved. Finally, in the future studies, according to this relational database management system, a web-based system will be constructed.

Ex. No.: 14	Case Study Submission For Recovery
Date: 22/10/2024	

Aim:

To understand the concept of database recovery

Procedure:

1. Select the system / application
2. Include database recovery methods.
3. Design the system for database recovery.

Recovery:

Recovery is the set of concepts, procedures, and strategies involved in protecting the database against data loss caused by media failure or users errors. In general, the purpose of a backup and recovery strategy is to protect the database against data loss and reconstruct lost data.

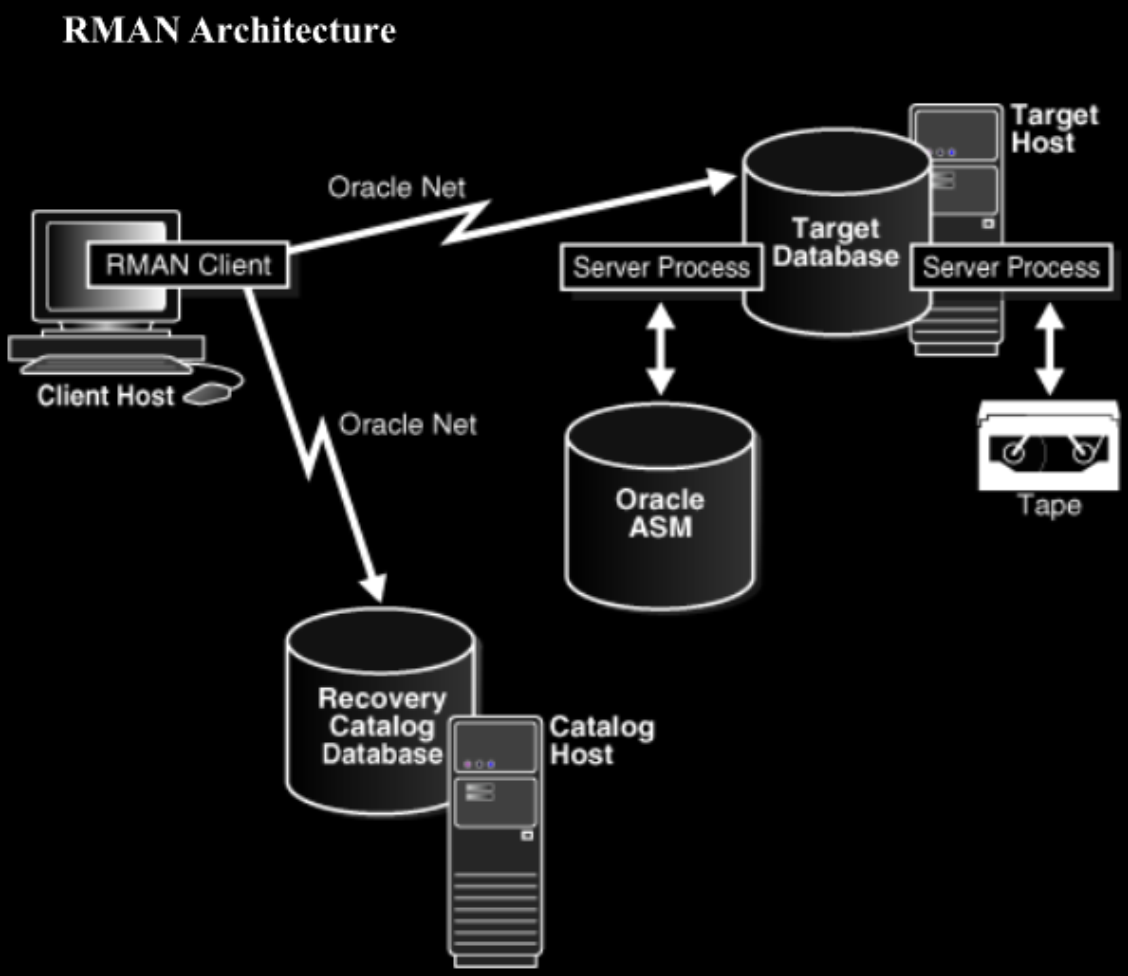
Recovery Techniques:

Recovery Manager (RMAN):

RMAN is an Oracle Database utility that integrates with an Oracle database to perform backup and recovery activities, including maintaining a repository of historical backup metadata in the control file of every database that it backs up. RMAN can also maintain a centralized backup repository called a recovery catalog in a different database. RMAN is an Oracle Database feature and does not require separate installation.

As an alternative to RMAN, you can use operating system commands such as the Linux dd for backing up and restoring files and the SQL*Plus RECOVER command for media recovery. User- managed backup and recovery is fully supported by Oracle, although RMAN is recommended because it is integrated with Oracle Database and simplifies administration.

The below diagram shows basic RMAN architecture. The RMAN client, accessible through Enterprise Manager, uses server sessions on a target database to back up data to disk or tape. RMAN can update an external recovery catalog with backup metadata.



Basic Concepts of Data Recovery Advisor:

This section explains the concepts that you must familiarize yourself with before using Data Recovery Advisor.

User Interfaces to Data Recovery Advisor:

Data Recovery Advisor has both a command-line and GUI interface. The GUI interface is available in Oracle Enterprise Manager Database Control and Grid Control.

Data Recovery Advisor:

The Data Recovery Advisor tool automatically diagnoses persistent data failures, presents appropriate repair options, and executes repairs at the user's request. By providing a centralized tool for automated data repair, Data Recovery Advisor improves the manageability and reliability of an Oracle database and thus helps reduce recovery time.

The database includes a framework called Health Monitor for running diagnostic checks. A checker is a diagnostic operation or procedure registered with Health Monitor to assess the health of the database or its components. The health assessment is known as a data integrity check and can be invoked reactively or proactively.

A failure is a persistent data corruption detected by a data integrity check. Failures are normally detected reactively. A database operation involving corrupted data results in an error, which automatically invokes a data integrity check that searches the database for failures related to the error. If failures are diagnosed, then the database records them in the Automatic Diagnostic Repository (ADR).

After failures have been detected by the database and stored in ADR, Data Recovery Advisor automatically determines the best repair options and their impact on the database. Typically, Data Recovery Advisor generates both manual and automated repair options.

Before presenting an automated repair option, Data Recovery Advisor validates it for the specific environment and for the availability of media components required to complete the proposed repair. If you choose an automatic repair, then Oracle Database executes it for you. The Data Recovery Advisor tool verifies the repair success and closes the appropriate failures.

Data File Recovery:

Data file recovery repairs a lost or damaged current data file or control file. It can also recover changes lost when a tablespace went offline without the OFFLINE NORMAL option. Data file media recovery differs depending on whether all changes are applied

- Complete recovery
- Incomplete recovery

Complete recovery:

Complete recovery applies all redo changes contained in the archived and online logs to a backup.

Typically, you perform complete media recovery after a media failure damages data files or the control file. You can perform complete recovery on a database, tablespace, or data file.

Incomplete recovery:

Incomplete recovery, also called database point-in-time recovery, results in a noncurrent version of the database. In this case, you do not apply all of the redo generated after the restored backup. Typically, you perform point-in-time database recovery to undo a user error when Flashback Database is not possible.

To perform incomplete recovery, you must restore all data files from backups created before the time to which you want to recover and then open the database with the RESETLOGS option when recovery completes.

Ex. No.: 15	Case study submission for database backups
Date: 22/10/2024	

Aim:

To understand the concept of database backups.

Procedure:

1. Select the system / application.
2. Include the backup methods.
3. Design the system for database backups.

Backup:

A backup is a copy of data. A backup can include crucial parts of the database such as data files, the server parameter file, and control file. A sample backup and recovery scenario is a failed disk drive that causes the loss of a data file. If a backup of the lost file exists, then you can restore and recover it. Media recovery refers to the operations involved in restoring data to its state before the loss occurred.

RMAN is integrated with Oracle Secure Backup, which provides reliable, centralized tape backup management, protecting file system data and Oracle Database files. The Oracle Secure Backup SBT interface enables you to use RMAN to back up and restore database files to and from tape and internet- based Web Services such as Amazon S3. Oracle Secure Backup supports almost every tape drive and tape library in SAN and SCSI environments.

Database Backups:

Database backups can be either physical or logical. Physical backups, which are the primary concern in a backup and recovery strategy, are copies of physical database files. You can make physical backups with RMAN or operating system utilities.

In contrast, logical backups contain logical data such as tables and stored procedures. You can extract logical data with an Oracle Database utility such as Data Pump Export and store it in a binary file. Logical backups can supplement physical backups.

Physical Backups:

Physical backups are copies of the physical files used in storing and recovering a database. These files include data files, control files, and archived redo logs. Ultimately, every

physical backup is a copy of files that store database information to another location, whether on disk or on offline storage media such as tape.

Logical Backups:

Logical backups contain logical data such as tables and stored procedures. You can use Oracle Data Pump to export logical data to binary files, which you can later import into the database. The Data Pump command-line clients expdp and impdp use the DBMS_DATAPUMP and DBMS_METADATA PL/SQL packages.

Whole and Partial Database Backups:

A whole database backup is a backup of every data file in the database, plus the control file. Whole database backups are the most common type of backup. A partial database backup includes a subset of the database: individual tablespaces or data files. A tablespace backup is a backup of all the data files in a tablespace or in multiple tablespaces. Tablespace backups, whether consistent or inconsistent, are valid only if the database is operating in ARCHIVELOG mode because redo is required to make the restored tablespace consistent with the rest of the database.

Backup techniques:

Feature	Recovery Manager	User-Managed	Data Pump Export
Closed database backups	Supported. Requires instance to be mounted.	Supported.	Not supported.
Open database backups	Supported. No need to use `BEGIN/END BACKUP` statements.	Supported. Must use `BEGIN/END BACKUP` statements.	Requires rollback or undo segments to generate consistent backups.
Incremental backups	Supported.	Not supported.	Not supported.
Corrupt block detection	Supported. Identifies corrupt blocks and logs in `V$DATABASE_BLOCK_CORRUPTION`.	Not supported.	Supported. Identifies corrupt blocks in the export log.
Automatic specification of files to include in a backup	Supported. Establishes the name and locations of all files to be backed up (whole database, tablespaces, data files, control files, and so on).	Not supported. Files to be backed up must be located and copied manually.	Not applicable.
Backup repository	Supported. Backups are recorded in the control file, which is the main repository of RMAN metadata. Additionally, you can store this metadata in a **recovery catalog**, which is a schema in a different database.	Not supported. DBA must keep own records of backups.	Not supported.
Backups to a media manager	Supported. Interfaces with a **media manager**. RMAN also supports proxy copy, a feature that allows a media manager to manage completely the transfer of data between disk and backup media.	Supported. Backup to tape is manual or controlled by a media manager.	Not supported.
Backup of initialization parameter file	Supported.	Supported.	Not supported.
Backup of password and networking files	Not supported.	Supported.	Not supported.
Platform-independent language for backups	Supported.	Not supported.	Supported.

Backup administration tasks include the following:

- Planning and testing responses to different kinds of failures
- Configuring the database environment for backup and recovery
- Setting up a backup schedule
- Monitoring the backup and recovery environment
- Troubleshooting backup problems
- Recovering from data loss if the need arises

Backup sets:

RMAN can also create backups in a proprietary format called a backup set. A backup set contains the data from one or more data files, archived redo log files, or control files or server parameter file. The smallest unit of a backup set is a binary file called a backup piece. Backup sets are the only form in which RMAN can write backups to sequential devices such as tape drives.

Backup sets enable tape devices to stream continuously. For example, RMAN can mingle blocks from slow, medium, and fast disks into one backup set so that the tape device has a constant input of blocks.

Image copies are useful for disk because you can update them incrementally, and also recover them.