SRI SAI RAM ENGINEERING COLLEGE

SAI LEO NAGAR, WEST TAMBARAM,
CHENNAI 44

(An Autonomous Institution)

NAME :

REGISTER NUMBER :

24AMPT301- DATABASE MANAGEMENT SYSTEM LABORATORY WITH THEORY

(II YEAR / III SEM) (BATCH: 2024 – 2028)

B.E CSE (ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)

ACADEMIC YEAR: 2025 – 2026

Sri
SAI RAM ENGINEERING COLLEGE
An Autonomous Institution | Affiliated to Anna University & Approved by AICTE, New Delhi
Accredited by NBA and NAAC "A+" | An ISO 9001:2015 Certified and MHRD NIRF ranked institution
Sai Leo Nagar, West Tambaram, Chennai - 600 044. www.sairam.edu.in
Founder Chairman : MJF. Ln. Leo Muthu

Certificate

REGISTER NO. : _____

Certified that this is the Bonafide Record of work done by Mr./Ms. _____ in the SEMESTER – III of Degree Course B.E CSE (ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING) in the 24AMPT301 – DATABASE MANAGEMENT SYSTEMS THEORY WITH LABORATORY during the academic year 2025 - 2026.

Station: Chennai – 600 044

Date:

STAFF IN-CHARGE **HEAD OF THE DEPARTMENT**

Submitted for University Practical Examination held on _____

at **Sri Sai Ram Engineering College**, Chennai – 600 044.

INTERNAL EXAMINER **EXTERNAL EXAMINER**

INDEX

EXP NO.	DATE	NAME OF THE EXPERIMENT	PAGE NO.	SIGNATURE
1.a.		Implementation of DDL Command		
1.b.		Implementation of DML Command		
1.c.		Implementation of TCL Command		
2.a.		Basic SQL Queries on Student Records		
2.b.		Implementation of Sub-queries and Nested Queries in SQL		
2.c.		Implementation of SQL JOIN Operations		
3.a.		Write a SQL program for adding two numbers and displaying the sum		
3.b.		Write a SQL program to print a series of “n” numbers using a for loop.		
3.c.		Write a SQL program to print a series of “n” numbers using a while loop.		
4.a.		SQL program to Debiting an amount from an entry in a table		
4.b.		SQL Program to Find the Sum of First N Numbers Using Recursion		

EXP NO.	DATE	NAME OF THE EXPERIMENT	PAGE. NO.	SIGNATURE
4.c.		SQL Program to Find Factorial Using Recursion		
5.a.		Reversing a String Using SQL Loop		
5.b.		Fibonacci Series Generation Using SQL		
5.c.		Displaying a Specific Attribute from a Table		
6		Banking System– Oracle Database Integration with Visual Basic		

Ex.No.:1a	Implementation of DDL Command
Date:	

AIM:

To implement and demonstrate the use of DDL commands: CREATE, DESCRIBE, ALTER (ADD and MODIFY), TRUNCATE, and DROP in SQL.

ALGORITHM:

1. Use CREATE TABLE to define a new table.
2. Use DESC to view the table structure.
3. Use ALTER TABLE to:
   a. Add new columns.
   b. Modify existing column definitions.
4. Use TRUNCATE TABLE to remove all data from the table.
5. Use DROP TABLE to delete the table permanently.

SYNTAX:

(1) Create:

This DDL Command is used to create a new table

Syntax:

CREATE TABLE table_name(column-1 datatype, column-2 datatype, column-3 datatype,... ,column-N datatype);

(2) Describe:

This DDL command is used to describe the table structure (displays the fields and their types).

Syntax:

desc<table name>;

(3) Alter:

This DDL command is used to modify the already existing table, but we cannot change or rename the table.

(i) Add:

This DDL command is used to add the column to an existing table.

Syntax:

alter table <table name> add (fieldname 1 datatype(size), fieldname2 datatype(size)...);

(ii) Modify:

It is used to modify column in already existing table.

Syntax:

alter table <table name>modify (column datatype(size)...);

(4) Truncate:

This DDL command is used to delete data in the table.

Syntax:

truncate table <tablename>;

(5) Drop:

This DDL command is used to drop a table.

Syntax:

drop table <tablename>;

PROGRAM:

```
/** Create table **/
CREATE TABLE emp1 (
  ename VARCHAR2(7),
  eno NUMBER(5),
  addr VARCHAR2(15),
  pho NUMBER(7),

  dept NUMBER(5)
);
```

/ Describe table **/**

DESC emp1;

/ Add column **/**

ALTER TABLE emp1 ADD (salary NUMBER(8,2));

/ Modify column **/**

ALTER TABLE emp1 MODIFY (ename VARCHAR2(8));

/ Truncate table **/**

TRUNCATE TABLE emp1;

/ Drop table **/**

DROP TABLE emp1;

/ Describe after drop **/**

DESC emp1;

OUTPUT:

SQL> CREATE TABLE emp1(ename VARCHAR2(7), eno NUMBER(5), addr VARCHAR2(15), pho NUMBER(7), dept NUMBER(5));

Table created.

SQL> DESC emp1;

Name Type

-------- ----------

ENAME VARCHAR2(7)

ENO NUMBER(5)

ADDR VARCHAR2(15)

PHO NUMBER(7)

DEPT NUMBER(5)

SQL> ALTER TABLE emp1 ADD(salary NUMBER(8,2));

Table altered.

```
SQL> DESC emp1;
Name     Type
-------- ----------------
ENAME    VARCHAR2(7)
ENO      NUMBER(5)
ADDR     VARCHAR2(15)
PHO      NUMBER(7)
DEPT     NUMBER(5)
SALARY   NUMBER(8,2)

SQL> ALTER TABLE emp1 MODIFY(ename VARCHAR2(8));
Table altered.
SQL> DESC emp1;
Name     Type
-------- ----------------
ENAME    VARCHAR2(8)
ENO      NUMBER(5)
ADDR     VARCHAR2(15)
PHO      NUMBER(7)
DEPT     NUMBER(5)
SALARY   NUMBER(8,2)

SQL> TRUNCATE TABLE emp1;
Table truncated.

SQL> DROP TABLE emp1;
Table dropped.
SQL> DESC emp1;
ERROR: ORA-04043: object emp1 does not exist
```

Result:

Thus all DDL query successfully executed.

Ex.No.:1b	Implementation of DML Command
Date:	

AIM:

To implement and demonstrate the use of DML commands: INSERT, SELECT, UPDATE, and DELETE in SQL.

ALGORITHM:

1. Create a table using CREATE TABLE.
2. Insert records using INSERT INTO.
3. Retrieve data using SELECT.
4. Modify data using UPDATE.
5. Remove data using DELETE.

SYNTAX:

(1) Insert:

This DML command is used to insert the details into the table.

Syntax:

insert into tablename values ('&field1', '&field2',…);

(2) Select:

This command is used to show the details present in a table.

Syntax:

select * from <table name>;

(3) Update:

This command is used to change a value of field in a row.

Syntax:

update <tablename> set < field='new value'> where <field='old value'>;

(4) Delete:

This command is used to delete a row in table.

Syntax:

PROGRAM:

```
-- Create table
CREATE TABLE emp2 (
  empno NUMBER(5),
  empname VARCHAR2(7),
  age NUMBER(3),
  job VARCHAR2(20),
  deptno NUMBER(5),
  salary NUMBER(8,2)
);
-- Insert entries
INSERT INTO emp2 VALUES (100, 'John', 36, 'Manager', 455, 17500);
INSERT INTO emp2 VALUES (101, 'Smith', 29, 'Clerk', 43, 1200);
INSERT INTO emp2 VALUES (102, 'A', 32, 'Assistant Manager', 43, 13500);
INSERT INTO emp2 VALUES (103, 'B', 37, 'General Manager', 355, 19000);
INSERT INTO emp2 VALUES (104, 'E', 28, 'Clerk', 35, 1500);
INSERT INTO emp2 VALUES (105, 'F', 38, 'General Manager', 40, 19000);

-- Select all records
SELECT * FROM emp2;

-- Select specific columns
SELECT empno, salary FROM emp2;

-- Select with condition
SELECT empno, salary FROM emp2 WHERE salary > 15000;

-- Update all entries
UPDATE emp2 SET salary = salary + 500;

-- Update specific entries
UPDATE emp2 SET salary = salary + 500 WHERE age > 35;
```

-- Delete specific entries

DELETE FROM emp2 WHERE deptno = 43;

-- Delete all entries

DELETE FROM emp2;

-- Final select

SELECT * FROM emp2;

OUTPUT:

SQL> CREATE TABLE emp2(...);

Table created.

SQL> INSERT INTO emp2 VALUES(...);

1 row created. (Repeated for all 6 entries)

SQL> SELECT * FROM emp2;

EMPNO EMPNAME AGE JOB DEPTNO SALARY

----- -------- --- ----------------- ------ -------

100 John 36 Manager 455 17500

101 Smith 29 Clerk 43 1200

102 A 32 Assistant Manager 43 13500

103 B 37 General Manager 355 19000

104 E 28 Clerk 35 1500

105 F 38 General Manager 40 19000

SQL> SELECT empno, salary FROM emp2;

(Displays empno and salary for all 6 rows)

SQL> SELECT empno, salary FROM emp2 WHERE salary > 15000;

(Displays rows with salary > 15000)

```
SQL> UPDATE emp2 SET salary = salary + 500;

6 rows updated.

SQL> UPDATE emp2 SET salary = salary + 500 WHERE age > 35;
3 rows updated.

SQL> DELETE FROM emp2 WHERE deptno = 43;
2 rows deleted.

SQL> DELETE FROM emp2;
4 rows deleted.

SQL> SELECT * FROM emp2;
No rows selected.
```

Result:

Thus all DML query successfully executed.

Ex.No.:1c	Implementation of TCL Command
Date:	

AIM:

To demonstrate the use of COMMIT, ROLLBACK, and SAVEPOINT commands for transaction control in SQL.

ALGORITHM:

1. Insert and update data.
2. Use COMMIT to save changes.
3. Use SAVEPOINT to mark transaction points.
4. Use ROLLBACK TO to revert to a specific savepoint.
5. Use SELECT to verify changes.

SYNTAX:

COMMIT command,

COMMIT command is used to permanently save any transaction into the database.

Following is commit command syntax,

COMMIT;

ROLLBACK command,

This command restores the database to last commited state. It is also used with SAVEPOINT command to jump to a savepoint in an ongoing transaction.

Following is rollback command syntax,

ROLLBACK TO savepoint_name;

SAVEPOINT command,

SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.

Following is savepoint command syntax,

SAVEPOINT savepoint_name;

SELECT * FROM class;

Result:

Thus all TCL query successfully executed.

Ex.No.: 2a	Basic SQL Queries on Student Records
Date:	

AIM:

To practice and demonstrate fundamental SQL query operations including data retrieval, filtering, column selection, and sorting using a students table.

PROCEDURE:

1. Access the Database
   Connect to the SQL environment containing the students table.
2. Perform Operations
   - Retrieve all student records.
   - Display only names and grades.
   - Filter students with grades above 80.
   - Sort records by grade in descending order.
3. Verify Results
   Ensure each operation returns accurate and expected output.

PROGRAM:

i)Retrieve all the records from the student's table.

```
SELECT * FROM students;
```

ii)Display only the name and grade of each student.

```
SELECT name, grade FROM students;
```

iii)Select all students who have scored more than 80.

SELECT * FROM students WHERE grade > 80;

iv)Display all student records sorted by grade in descending

SELECT * FROM students ORDER BY grade DESC;

OUTPUT:

```
+----+--------+-------+
| ID | Name   | Grade |
+----+--------+-------+
| 1  | Arjun  | 78    |
| 2  | Meera  | 85    |
| 3  | Ravi   | 92    |
| 4  | Sneha  | 67    |
| 5  | Kiran  | 88    |
+----+--------+-------+

-- Output for: SELECT name, grade FROM students;
+--------+-------+
| Name   | Grade |
+--------+-------+
| Arjun  | 78    |
| Meera  | 85    |
| Ravi   | 92    |
| Sneha  | 67    |
| Kiran  | 88    |
+--------+-------+

-- Output for: SELECT * FROM students WHERE grade > 80;
+----+--------+-------+
```

```
| ID | Name   | Grade |
+----+--------+-------+
| 2  | Meera  | 85    |
| 3  | Ravi   | 92    |
| 5  | Kiran  | 88    |
+----+--------+-------+

-- Output for: SELECT * FROM students ORDER BY grade DESC;
+----+--------+-------+
| ID | Name   | Grade |
+----+--------+-------+
| 3  | Ravi   | 92    |
| 5  | Kiran  | 88    |
| 2  | Meera  | 85    |
| 1  | Arjun  | 78    |
| 4  | Sneha  | 67    |
+----+--------+-------+
```

RESULT:

Thus SQL queries successfully executed for all student records, show names and grades, filter grades above 80, and sort by grade descending.

Ex.No.:2b	Implementation of Sub-queries and Nested Queries in SQL
Date:	

AIM:

To demonstrate the use of subqueries and nested queries in SQL .

PROCEDURE:

Subquery for Maximum Salary:

Use a subquery to find the highest salary in the company, and then filter employees whose salary matches that value.

Subquery for Average Salary:

Use a subquery to calculate the average salary and filter out employees who earn above this average.

Nested Query for Minimum Salary:

Use a nested query to find the lowest salary in the company, and then use the `HAVING` clause to identify departments with this minimum salary.

Multi-row Subquery for High Earners' Departments:

Use a multi-row subquery to find departments where at least one employee earns more than a specified threshold, and then retrieve all employees working in those departments.

PROGRAM:

i) Retrieve the employee(s) who earn the highest salary in the company using a subquery.

SELECT employee_name, salary FROM employees WHERE salary = (SELECT MAX(salary) FROM employees);

ii) Display the names and salaries of employees whose salary is above the

average salary of all employees.

SELECT employee_name, salary FROM employees WHERE salary > (SELECT AVG(salary) FROM employees);

iii)Find the department(s) where the lowest salary is paid, using a nested query.

SELECT department_name FROM departments WHERE department_id IN (SELECT department_id FROM employees GROUP BY department_id HAVING MIN(salary) = (SELECT MIN(salary) FROM employees));

iv) List all employees whose > department_id is found in a list of departments where

at least one employee earns more than 100000, using a multi-row subquery.

SELECT employee_name, department_id FROM employees WHERE department_id IN (SELECT department_id FROM employees WHERE salary 100000);

OUTPUT:

>>SQL

SELECT employee_name, salary

FROM employees

WHERE salary = (SELECT MAX(salary) FROM employees);

| employee_name | salary |

|---------------|--------------|

| John Doe | 120000 |

| Jane Smith | 120000 |

>>SQL

SELECT employee_name, salary

FROM employees

WHERE salary > (SELECT AVG(salary) FROM employees);

employee_name	salary
John Doe	120000
Jane Smith	95000
Bob Johnson	80000
Alice Brown	135000
Charlie White	110000

>>SQL

```
SELECT department_name
FROM departments
WHERE department_id IN (
   SELECT department_id
   FROM employees
   GROUP BY department_id
   HAVING MIN(salary) = (SELECT MIN(salary) FROM employees)
);
```

department_id	department_name
1	HR
2	Engineering
3	Marketing

```
>>SQL
SELECT employee_name, department_id
FROM employees
WHERE department_id IN (
   SELECT department_id
   FROM employees
   WHERE salary > 100000
);
```

employee_name	department_id
John Doe	1
Jane Smith	1
Bob Johnson	2
Alice Brown	2
Charlie White	3

RESULT:

Thus SQL queries for sub queries and nested queries was successfully executed.

Ex.No.:2c	Implementation of SQL JOIN Operations
Date:	

AIM:

To write SQL queries using various types of JOINs — INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN, and CROSS JOIN — to retrieve data from multiple related tables (employees and departments).

PROCEDURE:

i) Write a SQL query to list the names of all employees along with their department names, only for those employees who have a corresponding department in the departments table. Use an INNER JOIN.

```
SELECT e.employee_name, d.department_name
FROM employees e
INNER JOIN departments d
ON e.department_id = d.department_id;
```

ii) Write a SQL query to retrieve a list of all employees and their department names.If an employee does not belong to any department, show the department name as NULL. LEFT JOIN table names, only for those employees who have a corresponding department in the departments.Use an INNER JOIN.

```
SELECT e.employee_name, d.department_name
FROM employees e
INNER JOIN departments d
ON e.department_id = d.department_id;
```

iii) Write a SQL query to list all departments along with the names of employees working in those departments. If a department has no employees, display NULL for employee names.Use a RIGHT JOIN.

```
SELECT d.department_name, e.employee_name
```

```
FROM employees e
RIGHT JOIN departments d
ON e.department_id = d.department_id;
```

iv) Write a SQL query to list all employees and all departments, even if some employees are not assigned to a department and some departments have no employees. Use a FULL OUTER JOIN.

```
SELECT e.employee_name, d.department_name
FROM employees e
FULL OUTER JOIN departments d
ON e.department_id = d.department_id;
```

v) Write a SQL query to list all possible combinations of employee names and department names from the employees and departments tables. Use a CROSS JOIN

```
SELECT e.employee_name, d.department_name
FROM employees e CROSS JOIN departments d;
```

OUTPUT:

Employees table:

employee_id	employee_name	department_id
1	Alice	101
2	Bob	102
3	Charlie	NULL

Departments table:

department_id	department_name
101	HR
102	Finance
103	Marketing

i) INNER JOIN Output:

| employee_name | department_name |

|---|---|

| Alice | HR |

| Bob | Finance |

ii) LEFT JOIN Output:

| employee_name | department_name |

|---|---|

| Alice | HR |

| Bob | Finance |

| Charlie | NULL |

iii) RIGHT JOIN Output:

| department_name | employee_name |

|---|---|

| HR | Alice |

| Finance | Bob |

| Marketing | NULL |

iv) FULL OUTER JOIN Output:

| employee_name | department_name |

|---|---|

| Alice | HR |

| Bob | Finance |

| Charlie | NULL |

| NULL | Marketing |

V) CROSS JOIN Output:

| employee_name | department_name |

|---|---|

| Alice | HR |

| Alice | Finance |

| Alice | Marketing |

| Bob | HR | | Bob | Finance |

| Bob | Marketing |

| Charlie | HR |

| Charlie | Finance |

| Charlie | Marketing |

RESULT:

Thus SQL Queries for various types of join was successfully executed.

Ex.No.: 3a	Write a SQL program for adding two numbers and displaying the sum
Date:	

Aim

To write a SQL/PLSQL program that accepts two numbers, adds them, and displays the sum.

Procedure

1. Start the PL/SQL block with DECLARE section.
2. Declare three variables:
   Two variables to store the input numbers (a, b).
   One variable to store the result (c).
3. In the BEGIN section, assign values to the variables and perform addition (c := a + b).
4. Use DBMS_OUTPUT.PUT_LINE to display the result.
5. End the block with END;.
6. Execute the program in Oracle SQL*Plus or any PL/SQL environment.

PROGRAM

```
DECLARE
  a NUMBER := 10;
  b NUMBER := 20;
  c NUMBER;
BEGIN
```

```
  c := a + b;
  DBMS_OUTPUT.PUT_LINE('Sum of ' || a || ' and ' || b || ' is: ' || c);
END;
/
```

RESULT:

Thus SQL program for adding two numbers and displaying the sum was successfully executed.

Ex.No.: 3b	Write a SQL program to print a series of “n” numbers using a for loop.
Date:	

Aim

To write a SQL/PL/SQL program that prints a series of “n” numbers using a `FOR` loop.

Procedure

1. Start the PL/SQL block with the `DECLARE` section.
2. Declare a variable `n` to store the limit of the series.
3. In the `BEGIN` section, use a `FOR` loop with counter `i`.
4. The loop will iterate from `1` to `n`.
5. Inside the loop, use `DBMS_OUTPUT.PUT_LINE` to print the value of `i`.
6. End the block with `END;`.
7. Execute the program in Oracle SQL*Plus or any PL/SQL environment.

PROGRAM

```
DECLARE @n INT = 10;

-- Simulated FOR loop using WHILE

FOR i = 1 TO @n

BEGIN

   PRINT CAST(i AS VARCHAR);

END
```

Result:

Thus the SQL program to print series of ‘n’ numbers using for loop was successfully executed.

Ex.No.: 3c	Write a SQL program to print a series of “n” numbers using a while loop.
Date:	

Aim

To write a SQL/PLSQL program that prints a series of “n” numbers using a `WHILE` loop.

Procedure

1. Start the PL/SQL block with the `DECLARE` section.
2. Declare variables:
   `n` to store the limit of the series.
   `i` as the counter variable initialized to 1.
3. In the `BEGIN` section, write a `WHILE` loop with the condition `i <= n`.
4. Inside the loop, use `DBMS_OUTPUT.PUT_LINE` to print the value of `i`.
5. Increment `i` in each iteration (`i := i + 1`).
6. End the block with `END;`.
7. Execute the program in Oracle SQL*Plus or any PL/SQL environment.

Program

```
DECLARE
  n NUMBER := 10;  -- how many numbers to print
  i NUMBER := 1;
BEGIN
  WHILE i <= n LOOP
    DBMS_OUTPUT.PUT_LINE(i);
    i := i + 1;
```

```
  END LOOP;
END;
/
```

Result:

Thus the SQL program to print series of ‘n’ numbers using while loop was successfully executed.

Ex.No.:4a	SQL program to Debiting an amount from an entry in a table
Date:	

AIM:

To write a SQL program that debits a specified amount from a user's account balance in a table

ALGORITHM:

1. Identify the account using a unique identifier (e.g., account_number).
2. Subtract the debit amount from the current balance.
3. Update the table with the new balance
4. Ensure balance does not go negative

PROCEDURE:

1. Use UPDATE to modify the balance.
2. Apply WHERE clause to target the correct account.
3. Optionally use a condition to prevent overdraft.

PROGRAM

```
UPDATE accounts
SET balance = balance - debit_amount
WHERE account_number = '12345';
```

OUTPUT:

Query OK, 1 row affected

Result:

The SQL query to debit the amount and update the balance was successfully executed.

Ex.No.:4b	SQL Program to Find the Sum of First N Numbers Using Recursion
Date:	

AIM:

To write a SQL program using recursive CTE to calculate the sum of the first N natural numbers.

ALGORITHM:

1. Define a recursive Common Table Expression (CTE) named NumberSeries.
2. Start with base case: n = 1, total_sum = 1.
3. Recursively increment n and add it to total_sum.
4. Continue until n reaches the desired limit.
5. Select the final total_sum using ORDER BY n DESC LIMIT 1.

PROCEDURE:

1. Use WITH RECURSIVE to define the series.
2. Apply UNION ALL to build the recursive logic.
3. Use SELECT to retrieve the final sum.
4. Use ORDER BY and LIMIT to get the last row (final result).

PROGRAM:

```
WITH RECURSIVE NumberSeries (n, total_sum) AS (
  -- Base case
  SELECT 1, 1
  UNION ALL
  -- Recursive step
  SELECT n + 1, total_sum + (n + 1)
  FROM NumberSeries
  WHERE n < 10 -- Change 10 to desired value of N
)
```

```
SELECT total_sum
FROM NumberSeries
ORDER BY n DESC
LIMIT 1;
```

OUTPUT:

```
SQL> WITH RECURSIVE NumberSeries (n, total_sum) AS (
  SELECT 1, 1
  UNION ALL
  SELECT n + 1, total_sum + (n + 1)
  FROM NumberSeries
  WHERE n < 10 -- Change 10 to desired value of N
)
SELECT total_sum
FROM NumberSeries
ORDER BY n DESC
LIMIT 1;
```

TOTAL_SUM

55

RESULT:

The SQL recursive query successfully calculated the sum of the first 10 natural numbers. The final result is **55**

Ex.No.:4c	SQL Program to Find Factorial Using Recursion
Date:	

AIM:

To write a SQL program using recursive CTE to calculate the factorial of a given number.

ALGORITHM:

5. Define a recursive Common Table Expression (CTE) named FactorialSeries.
6. Start with base case: n = 0, factorial = 1.
7. Recursively increment n and multiply with current factorial.
8. Continue until n reaches the desired number.
9. Select the final factorial using ORDER BY n DESC LIMIT 1.

PROCEDURE:

4. Use WITH RECURSIVE to define the factorial series.
5. Apply UNION ALL to build the recursive logic.
6. Use SELECT to retrieve the final factorial.
7. Use ORDER BY and LIMIT to get the last row (final result)

PROGRAM

```
WITH RECURSIVE FactorialSeries (n, factorial) AS (
 -- Base case
 SELECT 0, 1
 UNION ALL
 -- Recursive step
 SELECT n + 1, factorial * (n + 1)
 FROM FactorialSeries
 WHERE n < 5 -- Change 5 to desired value
```

```
)
SELECT factorial
FROM FactorialSeries
ORDER BY n DESC
LIMIT 1;
```

OUTPUT:

```
SQL> WITH RECURSIVE FactorialSeries (n, factorial) AS (
  -- Base case
  SELECT 0, 1
  UNION ALL
  -- Recursive step
  SELECT n + 1, factorial * (n + 1)
  FROM FactorialSeries
  WHERE n < 5 -- Change 5 to desired value
)
SELECT factorial
FROM FactorialSeries
ORDER BY n DESC
LIMIT 1;
```

FACTORIAL

120

RESULT:

The SQL recursive query successfully calculated the factorial of 5. The final result is **120.**

Ex.No.: 5a	Reversing a String Using SQL Loop
Date:	

AIM:

To develop a SQL program that reverses a given string using a loop.

ALGORITHM:

1. Accept an input string.
2. Initialize an empty variable to hold the reversed string.
3. Use a loop to traverse the input string from the end to the beginning.
4. Concatenate each character to the reversed string.
5. Display the reversed string as the output.

PROGRAM

```
DECLARE
  input_str VARCHAR2(100) := 'SAIRAM';
  reversed_str VARCHAR2(100) := '';
BEGIN
  FOR i IN REVERSE 1..LENGTH(input_str) LOOP
    reversed_str := reversed_str || SUBSTR(input_str, i, 1);
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Original String: ' || input_str);
  DBMS_OUTPUT.PUT_LINE('Reversed String: ' || reversed_str);
END;
```

OUTPUT:

```
SQL> DECLARE
```

```
 input_str VARCHAR2(100) := 'SAIRAM';
  reversed_str VARCHAR2(100) := '';
BEGIN
  FOR i IN REVERSE 1..LENGTH(input_str) LOOP
    reversed_str := reversed_str || SUBSTR(input_str, i, 1);
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Original String: ' || input_str);
  DBMS_OUTPUT.PUT_LINE('Reversed String: ' || reversed_str);
END;
```

Original String: SAIRAM

Reversed String: MARIAS

RESULT:

The SQL program successfully reversed the string "SAIRAM" to "MARIAS" using a loop.

Ex.No.:5b	Fibonacci Series Generation Using SQL
Date:	

AIM:

To write a SQL program that generates the Fibonacci series up to a given number of terms.

ALGORITHM:

1. Initialize the first two numbers as 0 and 1.
2. Print the first two numbers.
3. Use a loop to generate the next terms by adding the previous two.
4. Continue until the required number of terms is reached.

PROGRAM

```
DECLARE
  n1 NUMBER := 0;
  n2 NUMBER := 1;
  n3 NUMBER;
  terms NUMBER := 10; -- number of terms to generate
BEGIN
  DBMS_OUTPUT.PUT_LINE('Fibonacci Series:');
  DBMS_OUTPUT.PUT(n1 || ' ' || n2);
  FOR i IN 3..terms LOOP
    n3 := n1 + n2;
    DBMS_OUTPUT.PUT(' ' || n3);
    n1 := n2;
```

```
   n2 := n3;
  END LOOP;
END;
```

OUTPUT:

```
SQL> DECLARE
  n1 NUMBER := 0;
  n2 NUMBER := 1;
  n3 NUMBER;
  terms NUMBER := 10; -- number of terms to generate
BEGIN
  DBMS_OUTPUT.PUT_LINE('Fibonacci Series:');
  DBMS_OUTPUT.PUT(n1 || ' ' || n2);
  FOR i IN 3..terms LOOP
    n3 := n1 + n2;
    DBMS_OUTPUT.PUT(' ' || n3);
    n1 := n2;
    n2 := n3;
  END LOOP;
END;
```

Fibonacci Series:

0 1 1 2 3 5 8 13 21 34

RESULT:

The SQL program successfully generated the first 10 terms of the Fibonacci series.

Ex.No.: 5c	Displaying a Specific Attribute from a Table
Date:	

AIM:

To display a specific attribute (column) from a database table using SQL queries.

ALGORITHM:

1. Create a sample table with multiple attributes.
2. Insert sample data into the table.
3. Use the SELECT statement to display only the required attribute.
4. Execute the query and view the result.

PROGRAM

```
-- Step 1: Create table
CREATE TABLE Student (
  RollNo INT PRIMARY KEY,
  Name VARCHAR(50),
  Dept VARCHAR(20),
  Marks INT
);

-- Step 2: Insert sample data
INSERT INTO Student VALUES (1, 'Arun', 'CSE', 85);
INSERT INTO Student VALUES (2, 'Priya', 'IT', 90);
INSERT INTO Student VALUES (3, 'Kumar', 'ECE', 88);
```

-- Step 3: Display only one attribute (Name column)

SELECT Name FROM Student;

OUTPUT:

SQL> CREATE TABLE Student (...);

Table created.

SQL> INSERT INTO Student VALUES (...);

1 row created.

(Repeated for all entries)

SQL> SELECT Name FROM Student;

NAME

Arun

Priya

Kumar

RESULT:

The SQL query successfully displayed the Name attribute from the Student table.

Ex.No.: 6	Banking System – Oracle Database Integration with Visual Basic
Date:	

AIM:

To design and implement a simple Banking System using Visual Basic (VB) that connects to an Oracle Database. The system performs basic banking operations such as viewing balance, depositing money, withdrawing money, and updating account details.

ALGORITHM:

1. Set up Oracle Database and create a table for account details.
2. Design a VB Windows Forms application with input fields and action buttons.
3. Establish a connection between VB and Oracle using ODP.NET or ODAC.
4. Implement button click events to perform SQL operations:
   - View balance
   - Deposit money
   - Withdraw money
   - Update account holder name

1. Display results and messages using labels and message boxes.

PROCEDURE:

Step 1: Oracle Database Setup

- Install Oracle XE or full version.
- Create schema BANK_DB.
- Create table:

Query:

```
CREATE TABLE Account (
  AccountNumber INT PRIMARY KEY,
  AccountHolder VARCHAR2(50),
```

```
  Balance DECIMAL(10,2)
);
```

Insert sample data:

```
SQL>CREATE TABLE Account (
  AccountNumber INT PRIMARY KEY,
  AccountHolder VARCHAR2(50),
  Balance DECIMAL(10,2)
);
```

Step 2: Visual Basic Setup

- Create a new Windows Forms Application.
- Add references to Oracle.DataAccess or Oracle.ManagedDataAccess.
- Design form with:
- TextBoxes: AccountNumber, AccountHolder, Amount
- Buttons: View Balance, Deposit, Withdraw, Update Account
- Labels: For displaying results

Step 3: VB Code Snippets

Connection Setup (Visual basic):

```
Dim conn As New OracleConnection("Data Source=YourOracleDB;User
Id=your_username;Password=your_password;")
```

View Balance (Visual basic):

```
Dim cmd As New OracleCommand("UPDATE Account SET AccountHolder = :holder
WHERE AccountNumber = :accNum", conn)
cmd.Parameters.Add("holder", OracleDbType.Varchar2).Value =
txtAccountHolder.Text
cmd.Parameters.Add("accNum", OracleDbType.Int32).Value =
CInt(txtAccountNumber.Text)
conn.Open()
cmd.ExecuteNonQuery()
conn.Close()
```

```
MessageBox.Show("Account holder name updated successfully.")
```

View Balance (Visual basic):

```
Dim cmd As New OracleCommand("SELECT Balance FROM Account WHERE
AccountNumber = :accNum", conn)

cmd.Parameters.Add("accNum", OracleDbType.Int32).Value =
CInt(txtAccountNumber.Text)

conn.Open()

Dim reader As OracleDataReader = cmd.ExecuteReader()

If reader.Read() Then

  lblBalance.Text = "Balance: " & reader("Balance").ToString()

Else

  MessageBox.Show("Account not found.")

End If

conn.Close()
```

Deposit Money:

```
Dim cmd As New OracleCommand("UPDATE Account SET Balance = Balance
+ :amount WHERE AccountNumber = :accNum", conn)

cmd.Parameters.Add("amount", OracleDbType.Decimal).Value =
CDec(txtAmount.Text)

cmd.Parameters.Add("accNum", OracleDbType.Int32).Value =
CInt(txtAccountNumber.Text)

conn.Open()

cmd.ExecuteNonQuery()

conn.Close()

MessageBox.Show("Deposit successful.")
```

Withdraw Money:

```
Dim cmd As New OracleCommand("UPDATE Account SET Balance = Balance -
:amount WHERE AccountNumber = :accNum", conn)

cmd.Parameters.Add("amount", OracleDbType.Decimal).Value =
CDec(txtAmount.Text)
```

```
cmd.Parameters.Add("accNum", OracleDbType.Int32).Value =
CInt(txtAccountNumber.Text)

conn.Open()

cmd.ExecuteNonQuery()

conn.Close()

MessageBox.Show("Withdrawal successful.")
```

Update Account Holder:

```
Dim cmd As New OracleCommand("UPDATE Account SET AccountHolder = :holder
WHERE AccountNumber = :accNum", conn)

cmd.Parameters.Add("holder", OracleDbType.Varchar2).Value =
txtAccountHolder.Text

cmd.Parameters.Add("accNum", OracleDbType.Int32).Value =
CInt(txtAccountNumber.Text)

conn.Open()

cmd.ExecuteNonQuery()

conn.Close()

MessageBox.Show("Account holder name updated successfully.")
```

OUTPUT:

View Balance:

Account Number: 1001

Account Holder: John Doe

Balance: 5000.00

After Deposit of ₹1000:

Deposit Amount: 1000.00

Updated Balance: 6000.00

RESULT:

Thus Banking System was successfully implemented using Visual Basic and Oracle Database.