

# EXPERIMENT – 1

Write SQL queries to CREATE TABLES for various databases using DDL commands like CREATE, ALTER, DROP, TRUNCATE).

## AIM:

To write SQL queries to CREATE TABLES for various databases using DDL commands like CREATE, ALTER, DROP, TRUNCATE).

## PROGRAM DESCRIPTION :

In this lab, we successfully practice how to write SQL queries to CREATE TABLES for various databases using DDL commands like CREATE, ALTER, DROP, TRUNCATE).

In SQL DDL commands are used to define the structure of the table. The DDL Commands as shown in the following Tables.

## CREATE TABLE:

Creates a table with specified constraints.

## SYNTAX:

```
CREATE TABLE tablename (
    column1 data_type [constraint] [, 
    column2 data_type [constraint] ] [, 
    PRIMARY KEY (column1 [, column2]) ] [, 
    FOREIGN KEY (column1 [, column2]) REFERENCES tablename]
    [,CONSTRAINT constraint]);
```

## PERSON TABLE :

```
SQL> CREATE TABLE Person1
  2  (
  3  PERSONID NUMBER(38) NOT NULL,
  4  LASTNAME VARCHAR2(255) NOT NULL,
  5  FIRSTNAME VARCHAR2(255),
  6  CITY VARCHAR2(255)
  7  );

Table created.
```

## INSERT INTO:

```
SQL> INSERT INTO Person1(PERSONID,LASTNAME,FIRSTNAME,CITY)
  2  VALUES(401,'Gowri','Jinka','Dharmavaram');

1 row created.

SQL> INSERT INTO Person1(PERSONID,LASTNAME,FIRSTNAME,CITY)
  2  VALUES(101,'Pallavi','Chittaboyani','Anantapuram');

1 row created.

SQL> INSERT INTO Person1(PERSONID,LASTNAME,FIRSTNAME,CITY)
  2  VALUES(201,'Pavithra','Chowdary','Kalyanadurgam');

1 row created.

SQL> INSERT INTO Person1(PERSONID,LASTNAME,FIRSTNAME,CITY)
  2  VALUES(301,'Pavani','Chanda','Darmapuri');

1 row created.
```

## DESC PERSON:

Name	Null?	Type
PERSONID	NOT NULL	NUMBER(38)
LASTNAME	NOT NULL	VARCHAR2(255)
FIRSTNAME		VARCHAR2(255)
CITY		VARCHAR2(255)

## SELECT :

```
SQL> SELECT * FROM Person1;
```

```
PERSONID
```

```
-----
```

```
LASTNAME
```

```
-----
```

```
FIRSTNAME
```

```
-----
```

```
CITY
```

```
-----
```

```
401
```

```
Gowri
```

```
Jinka
```

```
Dharmavaram
```

```
PERSONID
```

```
-----
```

```
LASTNAME
```

```
-----
```

```
FIRSTNAME
```

```
-----
```

```
CITY
```

```
-----
```

```
101
```

```
Pallavi
```

```
Chittaboyani
```

```
Anantapuram
```

```
PERSONID
```

```
-----
```

```
LASTNAME
```

```
-----
```

```
FIRSTNAME
```

```
-----
```

```
CITY
```

```
-----
```

```
201
```

```
Pavithra
```

```
Chowdary
```

```
Kalyanadurgam
```

```
PERSONID
```

```
-----
```

```
LASTNAME
```

```
-----
```

```
FIRSTNAME
```

```
-----
```

```
CITY
```

```
-----
```

```
301
```

```
Pavani
```

```
Chanda
```

```
Darmapuri
```

## ALTER TABLE:

Used to add or modify table details like column names and data types, column constraints.

### SYNTAX:

ALTER TABLE tablename

```
{ADD | MODIFY} (column_name data_type [ { ADD|MODIFY }  
Column_name data_type]);
```

### EXAMPLE:

```
SQL> ALTER TABLE Person1  
2 ADD AGE INT;  
  
Table altered.
```

### DESC Person1:

```
SQL> DESC Person1  
Name Null? Type  
-----  
-----  
PERSONID NOT NULL NUMBER(38)  
LASTNAME NOT NULL VARCHAR2(255)  
FIRSTNAME VARCHAR2(255)  
CITY VARCHAR2(255)  
AGE NUMBER(38)
```

## DROP TABLE:

Deletes the specified table.

### SYNTAX:

DROP TABLE table\_name;

### EXAMPLE:

```
SQL> DROP TABLE Person1;  
  
Table dropped.
```

## TRUNCATE TABLE:

To remove all rows in a specified table.

### SYNTAX:

TRUNCATE TABLE table\_name;

### EXAMPLE:

```
SQL> TRUNCATE TABLE Persons;  
Table truncated.
```

### CONCLUSION:

In this lab, we successfully practice how to write SQL queries to CREATE TABLES for various databases using DDL commands like CREATE, ALTER, DROP, TRUNCATE).

## EXPERIMENT-2

---

### TO MANIPULATE TABLES for various databases using DML commands

#### AIM:

To write the SQL queries to MANIPULATE TABLES for various databases using DML commands like INSERT, SELECT, UPDATE, DELETE).

#### PROGRAM DESCRIPTION :

In this lab, we learnt how to write the SQL queries to MANIPULATE TABLES for various databases using DML commands like INSERT, SELECT, UPDATE, DELETE).

#### INSERT COMMAND:

It is used to add values to a table.

#### SYNTAX:

```
INSERT INTO tablename  
VALUES (value1,value2,...,valuen);  
INSERT INTO tablename (column1, column2,...,column)  
VALUES (value1, value2,...,valuen);
```

#### SELECT COMMAND:

The SELECT command used to list the contents of a table.

#### SYNTAX:

```
SELECT columnlist  
FROM tablelist  
[WHERE conditionlist]  
[GROUP BY columnlist]  
[HAVING conditionlist]  
[ORDER BY columnlist [ASC|DESC]];
```

## UPDATE COMMAND:

The update command used to modify the contents of specified table.

## SYNTAX:

```
UPDATE tablename  
SET column_name = value[,  
Column_name = value ]  
[ WHERE condition_lsit ];
```

## DELETE COMMAND:

To delete all rows or specified rows in a table.

## SYNTAX:

```
DELETE FROM tablename [ WHERE condition_list];
```

## PARTS TABLE:

```
SQL> CREATE TABLE parts (  
 2  part_id NUMBER,  
 3  part_name VARCHAR(50) NOT NULL,  
 4  lead_time NUMBER(2, 0) NOT NULL,  
 5  cost NUMBER(9, 2) NOT NULL,  
 6  status NUMBER(1,0) NOT NULL,  
 7  PRIMARY KEY(part_id)  
 8  );
```

```
Table created.
```

## EXAMPLE ON INSERT:

```
SQL> INSERT INTO parts(part_id,part_name,lead_time,cost,status)
  2  VALUES(1,'sed dictum',5,134,0);

1 row created.

SQL> INSERT INTO parts(part_id,part_name,lead_time,cost,status)
  2  VALUES(2,'tristique neque',3,62,1);

1 row created.
```

```
SQL> INSERT INTO parts(part_id,part_name,lead_time,cost,status)
  2  VALUES(3,'dolor quam',16,82,1);

1 row created.

SQL> INSERT INTO parts(part_id,part_name,lead_time,cost,status)
  2  VALUES(4,'nec, diam.',41,10,1);

1 row created.

SQL> INSERT INTO parts(part_id,part_name,lead_time,cost,status)
  2  VALUES(5,'vitae erat',22,116,0);

1 row created.
```

## Oracle UPDATE – update multiple columns of a single row

### EXAMPLE :

```
SQL> UPDATE parts
  2  SET cost = 130
  3  WHERE part_id = 1;

1 row updated.
```

## Oracle UPDATE – update multiple rows example

### EXAMPLE:

```
SQL> UPDATE parts
  2  SET cost = cost * 1.05;
5 rows updated.
```

### EXAMPLE ON SELECT:

```
SQL> SELECT * FROM parts
  2  WHERE part_id = 1;

PART_ID PART_NAME                                     LEAD_TIME
----- -----
COST      STATUS
----- -----
1 sed dictum                                5
136.5          0
```

```
SQL> SELECT * FROM parts;

PART_ID PART_NAME                                     LEAD_TIME
----- -----
      1 sed dictum                                         5
    136.5          0

      2 tristique neque                                 3
   65.1           1

      3 dolor quam                                       16
  86.1           1

PART_ID PART_NAME                                     LEAD_TIME
----- -----
      4 nec, diam.                                      41
   10.5           1

      5 vitae erat                                     22
 121.8           0
```

## EXAMPLE ON DELETE:

```
SQL> DELETE FROM parts;
5 rows deleted.
```

## SELECT:

```
SQL> SELECT * FROM parts;
no rows selected
```

## CONCLUSION:

In this lab, we successfully practice how to write sql queries MANIPULATE TABLES for various databases using DML commands like INSERT, SELECT, UPDATE, DELETE).

## EXPERIMENT -3

### AIM:

To Implement view-level design using CREATE VIEW, ALTER VIEW and DELETE VIEW DDL commands.

### Program Description:

In this lab, we study the high-level design implementation of databases by using various view commands like create view, alter view and delete view.

A relation that is not part of a logical model, but it is made visible to a user as a virtual relation, is called a view. Therefore, view is referred as virtual relation.

### CREATE TABLE syntax:

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

### EXAMPLE-1:

```
SQL> CREATE TABLE persons(
 2 PersonName varchar2(255) NOT NULL,
 3 PersonCity varchar2(25) NOT NULL,
 4 PersonAddress varchar2(255)
 5 );

Table created.
```

### EXAMPLE -2:

```
SQL> CREATE TABLE Employee(
 2 EmployeeName varchar2(255) NOT NULL,
 3 EmployeeSalary NUMBER NOT NULL,
 4 EmployeeAddress varchar2(25)
 5 );

Table created.
```

**EXAMPLE-3**

```
SQL> CREATE TABLE INSTRUCTOR (ID VARCHAR2(5),
  2 NAME VARCHAR2(20) NOT NULL, DEPT_NAME VARCHAR2(20),
  3 SALARY NUMERIC(8,2) CHECK (SALARY > 29000), PRIMARY KEY (ID),
  4 FOREIGN KEY (DEPT_NAME) REFERENCES DEPARTMENT(DEPT_NAME) ON DELETE
  5 SET NULL
  6 );
Table created.
```

**CREATE VIEW Syntax:****View syntax:**

```
CREATE VIEW VIEW_NAME AS <QUERY EXPRESSION>
```

**EXAMPLE:**

```
SQL> CREATE VIEW FACULTY AS SELECT ID,NAME,DEPT_NAME FROM INSTRUCTOR;
View created.
```

```
SQL> CREATE VIEW HISTORY_INSTRUCTORS AS SELECT *
  2   FROM INSTRUCTOR
  3 WHERE DEPT_NAME= 'HISTORY';
View created.
```

**UPDATE :****EXAMPLE:**

```
SQL> Update History_instructors SET name='james robert' where Id='58584';
0 rows updated.
```

**DELETE:**

```
SQL> Delete FROM History_instructors where id='58584';
0 rows deleted.
```

## CONCLUSION:

In this Lab, We practice how to create view ,alter view , delete view

## EXPERIMENT - 4

WRITE SQL QUERIES TO PERFORM RELATIONAL OPERATIONS [i.e., UNION, UNION ALL, MINUS, INTERSECTION, CROSS JOIN, NATURAL JOIN].

### **AIM :**

To write SQL queries to perform relational operations like UNION, UNION ALL, MINUS,INTERSECTION, CROSS JOIN,NATURAL JOIN.

### **PROGRAM DESCRIPTION:**

In this lab, we study the implementation of RELATIONAL OPERATIONS.

RELATIONAL OPERAIONS are:

- 1.UNION
- 2.UNION ALL
- 3\_MINUS
- 4.INTERSECTION
- 5.CROSS JOIN
- 6.NATURAL JOIN

## DEPARTMENT TABLE :

```
SQL> CREATE TABLE DEPARTMENT (
 2  DEPT_NAME VARCHAR2(20),
 3  BUILDING VARCHAR2(15),
 4  BUDGET NUMERIC(12,2) CHECK(BUDGET>0),
 5  PRIMARY KEY(DEPT_NAME)
 6  );
```

```
Table created.
```

## INSERT INTO :

```
SQL> INSERT INTO DEPARTMENT(DEPT_NAME,BUILDING,BUDGET)
 2  VALUES('CSE','%WATSON','34000');
```

```
1 row created.
```

```
SQL> INSERT INTO DEPARTMENT(DEPT_NAME,BUILDING,BUDGET)
 2  VALUES('CSM','%WATSON','48000');
```

```
1 row created.
```

```
SQL> INSERT INTO DEPARTMENT(DEPT_NAME,BUILDING,BUDGET)
 2  VALUES('CSD','%TALYSON','64000');
```

```
1 row created.
```

```
SQL> INSERT INTO DEPARTMENT(DEPT_NAME,BUILDING,BUDGET)
 2  VALUES('ECE','%TALYSON','69000');
```

```
1 row created.
```

```
SQL> INSERT INTO DEPARTMENT(DEPT_NAME,BUILDING,BUDGET)
 2  VALUES('EEE','%PELTIER','69000');
```

```
1 row created.
```

```
SQL>
```

**SELECT :**

```
SQL> SELECT * FROM DEPARTMENT;
```

DEPT_NAME	BUILDING	BUDGET
CSE	%WATSON	34000
CSM	%WATSON	48000
CSD	%TALYSON	64000
ECE	%TALYSON	69000
EEE	%PELTIER	69000

**INSTRUCTOR TABLE :**

```
SQL> CREATE TABLE INSTRUCTOR
  2  (
  3   ID VARCHAR2(20) NOT NULL,
  4   DEPT_NAME VARCHAR2(20),
  5   NAME VARCHAR2(20) NOT NULL,
  6   SALARY NUMERIC(8,2) CHECK(SALARY>29000),
  7   PRIMARY KEY(ID),
  8   FOREIGN KEY(DEPT_NAME) REFERENCES DEPARTMENT(DEPT_NAME) ON DELETE SET NULL
  9 );
```

Table created.

**INSERT INTO :**

```
SQL> INSERT INTO INSTRUCTOR(ID,NAME,DEPT_NAME,SALARY)
  2  VALUES('568','PAVANI','CSE','120000');

1 row created.

SQL> INSERT INTO INSTRUCTOR(ID,NAME,DEPT_NAME,SALARY)
  2  VALUES('564','PALLAVI','CSE','80000');

1 row created.

SQL> INSERT INTO INSTRUCTOR(ID,NAME,DEPT_NAME,SALARY)
  2  VALUES('569','PAVITHRA','CSE','70000');

1 row created.

SQL> INSERT INTO INSTRUCTOR(ID,NAME,DEPT_NAME,SALARY)
  2  VALUES('3349','MANOGNA','CSM','100000');

1 row created.
```

```
SQL> INSERT INTO INSTRUCTOR(ID,NAME,DEPT_NAME,SALARY)
  2  VALUES('549','MANICHANDRIKA','CSD','60000');

1 row created.

SQL>
```

## SELECT :

```
SQL> SELECT * FROM INSTRUCTOR;

ID          DEPT_NAME      NAME        SALARY
--          -----
568          CSE           PAVANI     120000
564          CSE           PALLAVI    80000
569          CSE           PAVITHRA   70000
3349         CSM           MANOGNA   100000
549          CSD           MANICHANDRIKA 60000
```

## UNION :

```
SQL> SELECT DEPT_NAME FROM INSTRUCTOR
  2 UNION
  3 SELECT DEPT_NAME FROM DEPARTMENT;

DEPT_NAME
-----
CSD
CSE
CSM
ECE
EEE
```

```
SQL> SELECT SALARY FROM INSTRUCTOR
  2 UNION
  3 SELECT BUDGET FROM DEPARTMENT;

  SALARY
-----
  34000
  48000
  60000
  64000
  69000
  70000
  80000
 100000
 120000

9 rows selected.

SQL>
```

**UNION ALL :**

```
SQL> SELECT DEPT_NAME FROM INSTRUCTOR
  2 UNION ALL
  3 SELECT DEPT_NAME FROM DEPARTMENT;

DEPT_NAME
-----
CSE
CSE
CSE
CSM
CSD
CSE
CSM
CSD
ECE
EEE

10 rows selected.
```

**MINUS :**

```
SQL> SELECT DEPT_NAME FROM INSTRUCTOR
  2 ---MINUS
  3 ---SELECT DEPT_NAME FROM DEPARTMENT;

DEPT_NAME
-----
CSE
CSE
CSE
CSM
CSD
```

```
SQL> SELECT DEPT_NAME FROM DEPARTMENT
  2 ---MINUS
  3 ---SELECT DEPT_NAME FROM INSTRUCTOR;

DEPT_NAME
-----
CSE
CSM
CSD
ECE
EEE
```

## INTERSECTION :

```
SQL> SELECT DEPT_NAME FROM INSTRUCTOR  
  2  INTERSECT  
  3  SELECT DEPT_NAME FROM DEPARTMENT;  
  
DEPT_NAME  
-----  
CSD  
CSE  
CSM
```

## NATURAL JOIN :

```
SQL> SELECT* FROM INSTRUCTOR NATURAL JOIN DEPARTMENT;  
  
DEPT_NAME      ID      NAME      SALARY  
-----  
BUILDING      BUDGET  
-----  
CSE           569      PAVITHRA    70000  
%WATSON        34000  
  
CSE           564      PALLAVI     80000  
%WATSON        34000  
  
CSE           568      PAVANI      120000  
%WATSON        34000  
  
  
DEPT_NAME      ID      NAME      SALARY  
-----  
BUILDING      BUDGET  
-----  
CSM            3349     MANOGNA    100000  
%WATSON        48000  
  
CSD            549      MANICHANDRIKA 60000  
%TALYSON       64000
```

## CROSS JOIN :

```
SQL> SELECT NAME,D.DEPT_NAME,I.SALARY FROM INSTRUCTOR I, DEPARTMENT D
      WHERE I.DEPARTMENT_ID=D.ID
      ORDER BY NAME,DEPT_NAME,SALARY
```

NAME	DEPT_NAME	SALARY
PAVANI	CSE	120000
PAVANI	CSM	120000
PAVANI	CSD	120000
PAVANI	ECE	120000
PAVANI	EEE	120000
PALLAVI	CSE	80000
PALLAVI	CSM	80000
PALLAVI	CSD	80000
PALLAVI	ECE	80000
PALLAVI	EEE	80000
PAVITHRA	CSE	70000
NAME	DEPT_NAME	SALARY
PAVITHRA	CSM	70000
PAVITHRA	CSD	70000
PAVITHRA	ECE	70000
PAVITHRA	EEE	70000
MANOGNA	CSE	100000
MANOGNA	CSM	100000
MANOGNA	CSD	100000
MANOGNA	ECE	100000
MANOGNA	EEE	100000
MANICHANDRIKA	CSE	60000
MANICHANDRIKA	CSM	60000
NAME	DEPT_NAME	SALARY
MANICHANDRIKA	CSD	60000
MANICHANDRIKA	ECE	60000
MANICHANDRIKA	EEE	60000

25 rows selected.

## CONCLUSION :

In this lab, we practice RELATIONAL SET OPERATIONS like UNION, UNION ALL, MINUS, INTERSECTION, CROSS JOIN, NATURAL JOIN, and we learnt syntaxes of Relational Operations.

# EXPERIMENT – 5

Write SQL queries to perform AGGREGATE OPERATIONS (i.e. SUM, COUNT, AVG, MIN, MAX).

**AIM :** To write SQL queries to perform Aggregate operations like sum,avg,max,min,count.

**Program Description :**

Aggregate functions are functions that take a collection (a set or multiset) of

values as input and return a single value. SQL offers five built-in

**Aggregate Functions:**

- Average: avg
- Minimum: min
- Maximum: max
- Total: sum
- Count: count

The input to sum and avg must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well.

```
Microsoft Windows [Version 10.0.22621.2428]
(c) Microsoft Corporation. All rights reserved.

C:\Users\chand>cd ..

C:\Users>cd ..

C:\>sqlplus srit@localhost:1521/XEPDB1

SQL*Plus: Release 21.0.0.0.0 - Production on Thu Nov 9 11:50:38 2023
Version 21.3.0.0.0

Copyright (c) 1982, 2021, Oracle. All rights reserved.

Enter password:
Last Successful login time: Wed Oct 18 2023 18:54:44 +05:30

Connected to:
Oracle Database 21c Express Edition Release 21.0.0.0.0 - Production
Version 21.3.0.0.0
```

## CREATE TABLE SYNTAX :

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

## CREATE TABLE EXAMPLE :

### INSTRUCTOR TABLE:

```
SQL> CREATE TABLE DEPARTMENT
  2  (DEPT_NAME VARCHAR2(20),
  3  BUILDING VARCHAR2(15),
  4  BUDGET NUMERIC(12,2) CHECK (BUDGET > 0),
  5  PRIMARY KEY (DEPT_NAME)
  6  );
```

```
Table created.
```

### DEPARTMENT TABLE:

```
SQL> CREATE TABLE Instructor
  2  (
  3  ID VARCHAR2(20) NOT NULL,
  4  Name VARCHAR2(15),
  5  dept_name VARCHAR2(25),
  6  Salary NUMERIC(5,2) CHECK(Salary>29000),
  7  PRIMARY KEY(ID),
  8  FOREIGN KEY(dept_name) REFERENCES Department(dept_name) ON DELETE SET NULL
  9  );
```

```
Table created.
```

## INSERT INTO SYNTAX:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

## INSERT INTO EXAMPLE:

```
SQL> INSERT INTO DEPARTMENT(DEPT_NAME,BUILDING,BUDGET)
  2  VALUES('CSE','Watson',29000);

1 row created.
```

**AVERAGE :** The **AVG()** function returns the average value of a numeric column.

### AVERAGE SYNTAX :

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

```
SQL> SELECT avg(budget)
  2  FROM Department
  3  WHERE Budget>0;

AVG(BUDGET)
-----
29000

SQL> |
```

**SUM :** The **SUM()** function returns the total sum of a numeric column.

### SUM SYNTAX :

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

### SUM EXAMPLE :

```
SQL> SELECT avg(budget)
  2  FROM Department
  3  WHERE Budget>0;

AVG(BUDGET)
-----
29000
```

**MAXIMUM :** The **MAX()** function returns the largest value of the selected column.

## MAX SYNTAX :

```
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

### MAX EXAMPLE :

```
SQL> SELECT MAX(budget)
  2  FROM Department
  3  WHERE Budget>0;

MAX(BUDGET)
-----
29000
```

**MINIMUM :** The `MIN()` function returns the smallest value of the selected column.

## MIN SYNTAX :

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

### MIN EXAMPLE:

```
SQL> SELECT MIN(budget)
  2  FROM Department
  3  WHERE Budget>0;

MIN(BUDGET)
-----
29000
```

**COUNT :** The `COUNT()` function returns the number of rows that matches a specified criterion.

## COUNT SYNTAX :

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

### COUNT EXAMPLE :

```
SQL> SELECT count(budget)
  2  FROM Department
  3  WHERE Budget>0;

COUNT(BUDGET)
-----
      1
```

**GROUP BY :** The **GROUP BY** statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The **GROUP BY** statement is often used with aggregate functions (**COUNT()**, **MAX()**, **MIN()**, **SUM()**, **AVG()**) to group the result-set by one or more columns.

## GROUP BY SYNTAX :

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

### GROUP BY EXAMPLE :

```
SQL> SELECT budget
  2  FROM Department
  3  WHERE budget>0
  4  GROUP BY budget;

BUDGET
-----
      29000
```

**HAVING :** The **HAVING** clause was added to SQL because the **WHERE** keyword cannot be used with aggregate functions.

### **HAVING SYNTAX :**

```
SELECT column_name(s)
FROM table_name
GROUP BY column_name(s)
HAVING condition;
```

### **HAVING EXAMPLE :**

```
SQL> SELECT budget
  2  FROM Department
  3  GROUP BY budget
  4  HAVING budget>0;

      BUDGET
      -----
      29000
```

### **CONCLUSION :**

We successfully practice aggregate functions like sum,avg,max,min,count.

# EXPERIMENT - 6

Write SQL queries to perform JOIN OPERATIONS like  
(CONDITIONAL JOIN, EQUI JOIN, LEFT OUTER JOIN, RIGHT  
OUTER JOIN, FULL OUTER JOIN)

## AIM:

To write SQL queries to perform JOIN OPERATIONS like (CONDITIONAL JOIN, EQUI JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN)

## PROGRAM DESCRIPTION:

In this lab, we successfully practice how to write SQL queries to perform JOIN OPERATIONS like CONDITIONAL JOIN, EQUI JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN.

## DEPARTMENT TABLE:

```
SQL> CREATE TABLE DEPARTMENT1
  2  (
  3    DEPT_NAME VARCHAR2(20),
  4    BUILDING VARCHAR2(15),
  5    BUDGET NUMERIC(12,2) CHECK (BUDGET > 0),
  6    PRIMARY KEY (DEPT_NAME)
  7  );
Table created.
```

## INSERT INTO:

```
SQL> INSERT INTO department1 VALUES ('Biology', 'Watson', '90000');
1 row created.
```

## SELECT:

```
SQL> SELECT * FROM DEPARTMENT1;
```

DEPT_NAME	BUILDING	BUDGET
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

7 rows selected.

## INSTRUCTOR TABLE:

```
SQL> CREATE TABLE INSTRUCTOR1
  2  (
  3   ID VARCHAR2(5),
  4   NAME VARCHAR2(20) NOT NULL,
  5   DEPT_NAME VARCHAR2(20),
  6   SALARY NUMERIC(8,2) CHECK (SALARY > 29000),
  7   PRIMARY KEY (ID),
  8   FOREIGN KEY (DEPT_NAME) REFERENCES DEPARTMENT(DEPT_NAME)
  9   ON DELETE SET NULL
 10 );
```

Table created.

## INSERT INTO:

```
SQL> INSERT INTO instructor1 VALUES ('10101', 'Srinivasan', 'Comp. Sci.', '65000');

1 row created.
```

## SELECT:

```
SQL> SELECT * FROM INSTRUCTOR1;
```

ID	NAME	DEPT_NAME	SALARY
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

## CONDITIONAL JOIN:

A conditional column join is a fancy way to let us join to a single column and to two (or more) columns in a single query.

### SYNTAX:

```
SELECT *
FROM table1
JOIN table2 ON table1.column_name = table2.column_name;
WHERE table2.column name IS NULL;
```

### EXAMPLE:

SQL> SELECT * FROM DEPARTMENT1 JOIN INSTRUCTOR1 ON DEPARTMENT1.DEPT_NAME = INSTRUCT OR1.DEPT_NAME;				
DEPT_NAME	BUILDING	BUDGET	ID	NAME
DEPT_NAME	SALARY			
Comp. Sci.	Taylor	100000	10101	Srinivasan
Comp. Sci.	65000			
Comp. Sci.	Taylor	100000	45565	Katz
Comp. Sci.	75000			
Comp. Sci.	Taylor	100000	83821	Brandt
Comp. Sci.	92000			

## EQUI JOIN:

EQUI JOIN creates a JOIN for equality or matching column(s) values of the relative tables. EQUI JOIN also create JOIN by using JOIN with ON and then providing the names of the columns with their relative tables to check equality using equal sign (=).

### SYNTAX:

```
SELECT column_list
```

FROM table1, table2....  
WHERE table1.column\_name =  
table2.column\_name;

### EXAMPLE:

DEPT_NAME	BUILDING	BUDGET_ID	NAME
<hr/>			
DEPT_NAME	SALARY		
Comp. Sci.	Taylor	100000	10101 Srinivasan
Comp. Sci.	65000		
Comp. Sci.	Taylor	100000	45565 Katz
Comp. Sci.	75000		
Comp. Sci.	Taylor	100000	83821 Brandt
Comp. Sci.	92000		

### LEFT OUTER JOIN:

The **LEFT OUTER JOIN** keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records from the right side, if there is no match.

### SYNTAX:

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

*EXAMPLE:*

```
SQL> SELECT * FROM DEPARTMENT1 LEFT OUTER JOIN INSTRUCTOR1 ON  
2 DEPARTMENT1.DEPT_NAME = INSTRUCTOR1.DEPT_NAME;
```

DEPT_NAME	BUILDING	BUDGET_ID	NAME
-----------	----------	-----------	------

DEPT_NAME	SALARY
-----------	--------

Comp. Sci.	Taylor	100000	10101 Srinivasan
Comp. Sci.	65000		

Comp. Sci.	Taylor	100000	45565 Katz
Comp. Sci.	75000		

Comp. Sci.	Taylor	100000	83821 Brandt
Comp. Sci.	92000		

DEPT_NAME	BUILDING	BUDGET_ID	NAME
-----------	----------	-----------	------

DEPT_NAME	SALARY
-----------	--------

Biology	Watson	90000	
---------	--------	-------	--

History	Painter	50000	
---------	---------	-------	--

Elec. Eng.	Taylor	85000	
------------	--------	-------	--

DEPT_NAME	BUILDING	BUDGET_ID	NAME
-----------	----------	-----------	------

DEPT_NAME	SALARY
-----------	--------

Finance	Painter	120000	
---------	---------	--------	--

Music	Packard	80000	
-------	---------	-------	--

Physics	Watson	70000	
---------	--------	-------	--

9 rows selected.

## RIGHT OUTER JOIN:

The RIGHT OUTER JOIN keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match.

### SYNTAX:

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

### EXAMPLE:

```
SQL> SELECT * FROM DEPARTMENT1 RIGHT OUTER JOIN INSTRUCTOR1 ON
  2 DEPARTMENT1.DEPT_NAME = INSTRUCTOR1.DEPT_NAME;

DEPT_NAME          BUILDING      BUDGET_ID    NAME
-----          -----      -----
DEPT_NAME          SALARY
-----          -----
Comp. Sci.          Taylor        100000 10101 Srinivasan
Comp. Sci.          65000
Comp. Sci.          Taylor        100000 45565 Katz
Comp. Sci.          75000
Comp. Sci.          Taylor        100000 83821 Brandt
Comp. Sci.          92000
```

## FULL OUTER JOIN:

1.The **FULL OUTER JOIN** keyword returns all records when there is a match in left (table1) or right (table2) table records.

2.**FULL OUTER JOIN** and **FULL JOIN** are the same.

### SYNTAX:

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name
WHERE condition;
```

## EXAMPLE:

```
SQL> SELECT * FROM DEPARTMENT1 FULL OUTER JOIN INSTRUCTOR1 ON
  2  DEPARTMENT1.DEPT_NAME = INSTRUCTOR1.DEPT_NAME;

DEPT_NAME          BUILDING      BUDGET_ID    NAME
-----          -----        -----
DEPT_NAME          SALARY
-----          -----
Biology            Watson       90000
Comp. Sci.         Taylor      100000 10101 Srinivasan
Comp. Sci.         65000
Comp. Sci.         Taylor      100000 45565 Katz
Comp. Sci.         75000

DEPT_NAME          BUILDING      BUDGET_ID    NAME
-----          -----        -----
DEPT_NAME          SALARY
-----          -----
Comp. Sci.         Taylor      100000 83821 Brandt
Comp. Sci.         92000
Elec. Eng.         Taylor      85000
Finance            Painter     120000

DEPT_NAME          BUILDING      BUDGET_ID    NAME
-----          -----        -----
DEPT_NAME          SALARY
-----          -----
History            Painter     50000
Music              Packard     80000
Physics            Watson      70000

9 rows selected.
```

## CONCLUSION:

In this lab, we successfully practice how to write SQL Queries to perform JOIN OPERATIONS like CONDITIONAL JOIN, EQUI JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN.

# EXPERIMENT -7

Write SQL queries to perform AGGREGATE OPERATIONS (i.e. SUM, COUNT, AVG, MIN, MAX).

## AIM:

To write SQL queries to perform AGGREGATE OPERATIONS like SUM, COUNT, AVG, MIN, MAX.

## PROGRAM DESCRIPTION:

In this lab, we practice how to write SQL queries to perform AGGREGATE OPERATIONS like SUM, COUNT, AVG, MIN, MAX.

An aggregate function performs a calculation on a set of values, and returns a single value.

## DEPARTMENT TABLE:

```
SQL> CREATE TABLE DEPARTMENT1
  2  (
  3  DEPT_NAME VARCHAR2(20),
  4  BUILDING VARCHAR2(15),
  5  BUDGET NUMERIC(12,2) CHECK (BUDGET > 0),
  6  PRIMARY KEY (DEPT_NAME)
  7  );

Table created.
```

## INSERT INTO:

```
SQL> INSERT INTO department1 VALUES ('Biology', 'Watson', '90000');

1 row created.
```

**SELECT:**

```
SQL> SELECT * FROM DEPARTMENT1;

DEPT_NAME          BUILDING      BUDGET
-----          -----
Biology            Watson        90000
Comp. Sci.         Taylor       100000
Elec. Eng.         Taylor       85000
Finance           Painter      120000
History            Painter      50000
Music              Packard     80000
Physics            Watson       70000

7 rows selected.
```

**INSTRUCTOR TABLE:**

```
SQL> CREATE TABLE INSTRUCTOR1
  2  (
  3   ID VARCHAR2(5),
  4   NAME VARCHAR2(20) NOT NULL,
  5   DEPT_NAME VARCHAR2(20),
  6   SALARY NUMERIC(8,2) CHECK (SALARY > 29000),
  7   PRIMARY KEY (ID),
  8   FOREIGN KEY (DEPT_NAME) REFERENCES DEPARTMENT(DEPT_NAME)
  9   ON DELETE SET NULL
 10  );

Table created.
```

**INSERT INTO:**

```
SQL> INSERT INTO instructor1 VALUES ('10101', 'Srinivasan', 'Comp. Sci.', '65000');

1 row created.
```

**SELECT:**

```
SQL> SELECT * FROM INSTRUCTOR1;

ID      NAME          DEPT_NAME      SALARY
-----          -----
10101  Srinivasan    Comp. Sci.    65000
45565  Katz          Comp. Sci.    75000
83821  Brandt       Comp. Sci.    92000
```

## SUM () FUNCTION:

The **Sum()** function returns the total sum of a numeric column.

### Syntax:

```
SELECT SUM(Column_name)
```

```
FROM table_name
```

```
WHERE condition;
```

### EXAMPLE:

```
SQL> SELECT SUM(BUDGET)
  2  FROM DEPARTMENT1
  3  WHERE BUDGET>50000;

SUM(BUDGET)
-----
545000
```

## COUNT () FUNCTION:

The **COUNT()** function returns the number of rows that matches a specified criterion.

### SYNTAX:

```
SELECT COUNT(column_name)
```

```
FROM table_name
```

```
WHERE condition;
```

### EXAMPLE:

```
SQL> SELECT COUNT(BUDGET)
  2  FROM DEPARTMENT1
  3  WHERE BUDGET>50000;

COUNT(BUDGET)
-----
6
```

## AVG FUNCTION:

AVG () computes the average of a set of values by dividing the sum of those values by the count of nonnull values.

### SYNTAX:

```
SELECT AVG(Column_name)
```

```
FROM table_name
```

```
WHERE condition;
```

### EXAMPLE:

```
SQL> SELECT AVG(BUDGET)
  2  FROM DEPARTMENT1
  3  WHERE BUDGET>50000;

AVG(BUDGET)
-----
90833.3333
```

## MAX FUNCTION:

The **MAX()** function returns the largest value of the selected column.

### SYNTAX:

```
SELECT MAX (Column_name)
```

```
FROM table_name
```

```
WHERE condition;
```

### EXAMPLE:

```
SQL> SELECT MAX(BUDGET)
  2  FROM DEPARTMENT1
  3  WHERE BUDGET>50000;

MAX(BUDGET)
-----
120000
```

**MIN FUNCTION:**

The **MIN()** function returns the smallest value of the selected column.

**SYNTAX:**

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

**EXAMPLE:**

```
SQL> SELECT MIN(BUDGET)
  2  FROM DEPARTMENT1
  3  WHERE BUDGET>50000;

MIN(BUDGET)
-----
70000
```

**CONCLUSION:**

In this lab, we successfully practice how to write SQL queries to perform AGGREGATE OPERATIONS like SUM, COUNT, AVG, MIN, MAX.

# EXPERIMENT – 8

Write SQL queries to perform ORACLE BUILT-IN FUNCTIONS (i.e. DATE, TIME).

## AIM:

To write a SQL queries to perform ORACLE BUILT-IN FUNCTIONS like DATE, TIME.

## PROGRAM DESCRIPTION:

In this lab, we practice how to write a SQLqueries to perform ORACLE BUILT-IN FUNCTIONS like DATE, TIME.

### Built-in Functions

#### 1.Character Functions

##### \*Case-conversion functions

- UPPER
- LOWER
- INIT CAP

##### \*Character manipulation functions

- SUBSTR
- LENGTH
- REPLACE
- INSTR
- RPAD
- LPAD
- CONCAT
- TRIM
  - i) LTRIM
  - ii) RTRIM
  - iii)

## 2.Number Functions

- ROUND
- TRUNC
- MOD

## 3.DATE functions

- SYSDATE
- MONTHS\_BETWEEN
- ADD\_MONTHS
- NEXT\_DAY
- LAST\_DAY
- ROUND(DATE)
- TRUNCATE(DATE)

## DEPARTMENT TABLE:

```
SQL> CREATE TABLE DEPARTMENT1
  2  (
  3  DEPT_NAME VARCHAR2(20),
  4  BUILDING VARCHAR2(15),
  5  BUDGET NUMERIC(12,2) CHECK (BUDGET > 0),
  6  PRIMARY KEY (DEPT_NAME)
  7  );
```

```
Table created.
```

## INSERT INTO:

```
SQL> INSERT INTO department1 VALUES ('Biology', 'Watson', '90000');
1 row created.
```

SELECT:

```
SQL> SELECT * FROM DEPARTMENT1;

DEPT_NAME          BUILDING      BUDGET
-----          -----
Biology            Watson        90000
Comp. Sci.         Taylor       100000
Elec. Eng.         Taylor       85000
Finance           Painter      120000
History            Painter      50000
Music              Packard     80000
Physics            Watson       70000

7 rows selected.
```

## CASE CONVERSION FUNCTIONS:

UPPER:

Syntax:

UPPER(*text*)

EXAMPLE:

```
SQL> SELECT UPPER(DEPT_NAME) FROM DEPARTMENT1;

UPPER(DEPT_NAME)
-----
BIOLOGY
COMP. SCI.
ELEC. ENG.
FINANCE
HISTORY
MUSIC
PHYSICS

7 rows selected.
```

LOWER:SL

SYNTAX:

LOWER (*text*)

**EXAMPLE:**

```
SQL> SELECT LOWER(DEPT_NAME) FROM DEPARTMENT1;

LOWER(DEPT_NAME)
-----
biology
comp. sci.
elec. eng.
finance
history
music
physics

7 rows selected.
```

**INIT CAP:****SYNTAX:**

INITCAP (string)

**EXAMPLE:**

```
SQL> SELECT UPPER('hello world'),LOWER('HELLO WORLD'),
  2 INITCAP ('hello world') FROM DUAL;

UPPER('HELL LOWER('HELL INITCAP('HE
-----
HELLO WORLD hello world Hello World
```

**CHARACTER MANIPULATION FUNCTION:****SUBSTR:**

SUBSTRING (string, start, length)

**EXAMPLE:**

```
SQL> SELECT SUBSTR('HELLO WORLD',3,7) FROM DUAL;

SUBSTR(
-----
LLO WOR
```

**LENGTH:****SYNTAX:**LENGTH (*string*)**EXAMPLE:**

```
SQL> SELECT LENGTH('HELLO WORLD') FROM DUAL;  
  
LENGTH('HELLOWORLD')  
-----  
           11
```

**REPLACE:****SYNTAX:**REPLACE (*string, old\_string, new\_string*)**EXAMPLE:**

```
SQL> SELECT REPLACE('HELLO WORLD', 'WORLD', 'INDIA') FROM DUAL;  
  
REPLACE('HE  
-----  
HELLO INDIA
```

**INSTR:**

```
SQL> SELECT INSTR('HELLO WORLD', 'WORLD') FROM DUAL;  
  
INSTR('HELLOWORLD', 'WORLD')  
-----  
           7
```

**LPAD:**

```
SQL> SELECT LPAD('HELLO WORLD', 20, '*') FROM DUAL;  
  
LPAD('HELLOWORLD', 20  
-----  
*****HELLO WORLD
```

**RPAD:**

```
SQL> SELECT RPAD('HELLO WORLD',20,'*') FROM DUAL;
RPAD('HELLOWORLD', 20
-----
HELLO WORLD*****
```

## CONCAT:

### SYNTAX:

CONCAT (*string1, string2, ...., string\_n*)

### EXAMPLE:

```
SQL> SELECT CONCAT(DEPT_NAME,BUDGET) FROM DEPARTMENT1;
CONCAT(DEPT_NAME,BUDGET)
-----
Biology90000
Comp. Sci.100000
Elec. Eng.85000
Finance120000
History50000
Music80000
Physics70000

7 rows selected.
```

## TRIM:

### SYNTAX:

TRIM ([*characters* FROM ] *string*)

### EXAMPLE:

```
SQL> SELECT TRIM('      HELLO WORLD      ') FROM DUAL;
TRIM('HELLO
-----
HELLO WORLD
```

## LTRIM:

### SYNTAX:

LTRIM (*string*)

**EXAMPLE:**

```
SQL> SELECT LTRIM('      HELLO WORLD      ') FROM DUAL;  
  
LTRIM('HELLOWORL  
-----  
HELLO WORLD
```

**RTRIM:****SYNTAX:**

RTRIM (*string*)

**EXAMPLE:**

```
SQL> SELECT RTRIM('      HELLO WORLD      ') FROM DUAL;  
  
RTRIM('HELLOWORL  
-----  
HELLO WORLD
```

**NUMBER FUNCTIONS:****ROUND:****Syntax:**

ROUND (*number, decimals*)

**EXAMPLE:**

```
SQL> SELECT ROUND(176.1256,2)FROM DUAL;  
  
ROUND(176.1256,2)  
-----  
176.13
```

**TRUNC:****SYNTAX:**

TRUNCATE (*number, decimals*)

**EXAMPLE:**

```
SQL> SELECT TRUNC(576.5256,-2)FROM DUAL;  
  
TRUNC(576.5256,-2)  
-----  
      500
```

**MOD:****SYNTAX:**

MOD ( $x$ ,  $y$ )

**EXAMPLE:**

```
SQL> SELECT MOD(900,91)FROM DUAL;  
  
MOD(900,91)  
-----  
      81
```

**DATE FUNCTIONS:****SYSDATE:**

```
SQL> SELECT SYSDATE FROM DUAL;  
  
SYSDATE  
-----  
10-DEC-23
```

**MONTHS\_BETWEEN**

```
SQL> SELECT MONTHS_BETWEEN(SYSDATE,'22-JUL-2004') FROM DUAL;  
  
MONTHS_BETWEEN(SYSDATE,'22-JUL-2004')  
-----  
      232.634638
```

**ADD\_MONTHS**

```
SQL> SELECT ADD_MONTHS(SYSDATE, 5) FROM DUAL;  
  
ADD_MONTH  
-----  
10-MAY-24
```

## NEXT\_DAY

```
SQL> SELECT NEXT_DAY(SYSDATE, 'WEDNESDAY') FROM DUAL;

NEXT_DAY(
-----
13-DEC-23
```

## LAST\_DAY

```
SQL> SELECT LAST_DAY(SYSDATE) FROM DUAL;

LAST_DAY(
-----
31-DEC-23
```

## ROUND(DATE)

```
SQL> SELECT ROUND(SYSDATE, 'YEAR') FROM DUAL;

ROUND(SYS
-----
01-JAN-24
```

## TRUNCATE(DATE)

```
SQL> SELECT TRUNC(SYSDATE, 'DAY') FROM DUAL;

TRUNC(SYS
-----
10-DEC-23
```

## CONCLUSION:

In this lab, we successfully practice how to write a SQLqueries to perform ORACLE BUILT-IN FUNCTIONS like DATE, TIME.

C.PALLAVI

224G1A0564

DATE: 21-11-2023

# EXPERIMENT – 9

Write SQL queries to perform KEY CONSTRAINTS (i.e. PRIMARY KEY, FOREIGN KEY, UNIQUE NOT NULL, CHECK, DEFAULT).

## AIM:

To write SQL queries to perform KEY CONSTRAINTS (i.e. PRIMARY KEY, FOREIGN KEY, UNIQUE NOT NULL, CHECK, DEFAULT).

## PROGRAM DESCRIPTION:

In this lab, we successfully practice how to write SQL queries to perform KEY CONSTRAINTS like PRIMARY KEY, FOREIGN KEY, UNIQUE NOT NULL, CHECK, DEFAULT.

## CONSTRAINTS:

SQL constraints are used to specify rules for the data in a table.

### Types of SQL Constraints:

1. NOT NULL - Ensures that a column cannot have a NULL value
2. UNIQUE - Ensures that all values in a column are different
3. PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
4. FOREIGN KEY - Uniquely identifies a row/record in another table
5. CHECK - Ensures that all values in a column satisfies a specific condition
6. DEFAULT - Sets a default value for a column when no value is specified.

First APPLY CONSTRAINT USING CREATE Statement.

Second, DROP the Constraint

third, verify the constraint is working or not using insert statement;

Fourth, make a note of error occurred in observation

After Verification Drop the table to define other Constraints on the same relation-name

### **NOT NULL Constraint Example:**

#### **SYNTAX:**

```
SELECT column_names  
FROM table_name  
WHERE column_name IS NOT NULL;
```

#### **EXAMPLE :**

```
SQL> CREATE TABLE STUDENT1 (  
  2 ID int NOT NULL,  
  3 LastName varchar(255) NOT NULL,  
  4 FirstName varchar(255) NOT NULL,  
  5 Age int  
  6 );  
  
Table created.
```

#### **ALTER:**

```
SQL> ALTER TABLE STUDENT1  
  2 MODIFY Age int NOT NULL;  
  
Table altered.
```

### **UNIQUE CONSTRAINT EXAMPLE:**

The **UNIQUE** constraint ensures that all values in a column are different.

**EXAMPLE:**

```
SQL> CREATE TABLE Students(
  2  ID int NOT NULL,
  3  LastName varchar(255) NOT NULL,
  4  FirstName varchar(255),
  5  Age int,
  6  CONSTRAINT UC_Person UNIQUE (ID,LastName)
  7 );
```

Table created.

**ALTER:**

```
SQL> ALTER TABLE students
  2  DROP CONSTRAINT UC_Person;
```

Table altered.

**PRIMARY KEY CONSTRAINT Example:**

The PRIMARY KEY constraint uniquely identifies each record in a table.

Primary keys must contain UNIQUE values, and cannot contain NULL values.

**EXAMPLE:**

```
SQL> CREATE TABLE Person1 (
  2  ID int NOT NULL,
  3  LastName varchar(255) NOT NULL,
  4  FirstName varchar(255),
  5  Age int,
  6  PRIMARY KEY (ID,LastName)
  7 );
```

Table created.

**FORIEGN KEY CONSTRAINTS Example:**

The **FOREIGN KEY** constraint is used to prevent actions that would destroy links between tables.

A **FOREIGN KEY** is a field (or collection of fields) in one table, that refers to the [PRIMARY KEY](#) in another table.

**EXAMPLE:**

```
SQL> CREATE TABLE INSTRUCTOR1
  2  (
  3  ID VARCHAR2(5),
  4  NAME VARCHAR2(20) NOT NULL,
  5  DEPT_NAME VARCHAR2(20),
  6  SALARY NUMERIC(8,2) CHECK (SALARY > 29000),
  7  PRIMARY KEY (ID),
  8  FOREIGN KEY (DEPT_NAME) REFERENCES DEPARTMENT(DEPT_NAME)
  9  ON DELETE SET NULL
 10 );

Table created.
```

**CHECK CONSTRAINTS Example:**

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

**EXAMPLE:**

```
SQL> CREATE TABLE Person2 (
  2  ID int NOT NULL,
  3  LastName varchar(255) NOT NULL,
  4  FirstName varchar(255),
  5  Age int,
  6  City varchar(255),
  7  CHECK (Age>=18 AND City='Sandnes')
  8 );

Table created.
```

**ALTER:**

```
SQL> ALTER TABLE Person2
  2  ADD CHECK (Age>=18 AND City='Sandnes');

Table altered.
```

**DEFAULT CONSTRANTS Example:**

The **DEFAULT** constraint is used to set a default value for a column.

The default value will be added to all new records, if no other value is specified

### EXAMPLE:

```
SQL> CREATE TABLE Person3(
  2  ID int NOT NULL,
  3  LastName varchar(255) NOT NULL,
  4  FirstName varchar(255),
  5  Age int,
  6  City varchar(255) DEFAULT 'Sandnes'
  7  );  
  
Table created.
```

### ALTER:

```
SQL> ALTER TABLE Persons
  2  MODIFY City DEFAULT 'Sandnes';  
  
Table altered.
```

**Drop cannot be dropped, So make it null value to remove default**

```
SQL> ALTER TABLE Persons MODIFY city DEFAULT NULL;  
  
Table altered.
```

### CONCLUSION;

In this lab, we successfully practice how to write SQL queries to perform KEY CONSTRAINTS like PRIMARY KEY, FOREIGN KEY, UNIQUE NOT NULL, CHECK, DEFAULT.

# EXPERIMENT-10

## 10. WRITE SQL PROGRAM FOR CALCULATING THE FACTORIAL OF A GIVEN NUMBER.

**AIM :** To write a pl/sql program for calculating the factorial of a given number.

**Program :**

```
SQL> DECLARE
  2  fac NUMBER:=1;
  3  n NUMBER;
  4  BEGIN
  5  n:= &number;
  6  WHILE n > 0 LOOP
  7  fac:=n*fac;
  8  n:=n-1;
  9  END LOOP;
10  DBMS_OUTPUT.PUT_LINE("FAC");
11  END;
12 /
Enter value for number: 5
old   5: n:= &number;
new   5: n:= 5;

PL/SQL procedure successfully completed.

SQL>
```

**OUTPUT:**

```
SQL> SET SERVEROUTPUT ON
SQL> /
Enter value for number: 5
old   5: n:= &number;
new   5: n:= 5;
120

PL/SQL procedure successfully completed.

SQL>
```

```
SQL> SET VERIFY OFF
SQL> /
Enter value for number: 5
120

PL/SQL procedure successfully completed.

SQL>
```

**CONCLUSION :** In this lab, -we successfully practice how to write a PL/SQL program for calculations the factorial of a given number.

# EXPERIMENT -11

**Write PL/SQL program for finding the given number or not.**

*AIM: To write a PL/SQL program for finding the given number is prime or not.*

**Program :**

```
SQL> DECLARE
  2  n NUMBER;
  3  i NUMBER;
  4  temp NUMBER;
  5  BEGIN
  6  n := &number;
  7  i := 2;
  8  temp := 1;
  9  FOR i IN 2..n/2
 10 LOOP
 11 IF MOD(n, i) = 0
 12 THEN
 13 temp := 0;
 14 EXIT;
 15 END IF;
 16 END LOOP;
 17 IF temp = 1
 18 THEN
 19 DBMS_OUTPUT.PUT_LINE(n||' is a prime number');
 20 ELSE
 21 DBMS_OUTPUT.PUT_LINE(n||' is not a prime number');
 22 END IF;
 23 END;
 24 /
Enter value for number: 5
5 is a prime number

PL/SQL procedure successfully completed.
```

**OUTPUT : PL/SQL Procedure successfully completed**

SQL > SET SERVEROUTPUT ON

SQL>/

**ENTER VALUE FOR NUMBER:5**

5 is a prime number.

PL/SQL procedure successfully completed.

**CONCLUSION :**

In this lab, we successfully practiced how to write PL/SQL program for finding the given number is prime number or not.

# EXPERIMENT – 12

**Write a PL/SQL program for displaying the Fibonacci series up to an integer.**

*AIM : To write a PL/SQL program for finding the Fibonacci series up to an integer.*

**Program :**

```
SQL> DECLARE
  2  FIRST NUMBER:=0;
  3  SECOND NUMBER:=1;
  4  N NUMBER:=&NUMBER;
  5  I NUMBER;
  6  TEMP NUMBER;
  7  BEGIN
  8  DBMS_OUTPUT.PUT_LINE('SERIES');
  9  DBMS_OUTPUT.PUT_LINE(FIRST);
 10 DBMS_OUTPUT.PUT_LINE(SECOND);
 11 FOR I IN 2..N
 12 LOOP
 13 TEMP:=FIRST+SECOND;
 14 FIRST:=SECOND;
 15 SECOND:=TEMP;
 16 DBMS_OUTPUT.PUT_LINE(TEMP);
 17 END LOOP;
 18 END;
 19 /
```

**OUTPUT :**

```
Enter value for number: 5
old  4: N NUMBER:=&NUMBER;
new  4: N NUMBER:=5;

PL/SQL procedure successfully completed.

SQL> SET VERIFY OFF
SQL> /
Enter value for number: 5

PL/SQL procedure successfully completed.

SQL> SET SERVEROUTPUT ON
SQL> /
Enter value for number: 5
SERIES
0
1
1
2
3
5

PL/SQL procedure successfully completed.
```

**CONCLUSION :** In this lab, we successfully practice PL/SQL program for displaying the Fibonacci series up to an integer.

# EXPERIMENT – 13

**Write PL/SQL program to implement Stored Procedure on table.**

## AIM:

To write PL/SQL program to implement Stored Procedure on table.

## PROGRAM DESCRIPTION :

1. In this lab, we practice how to write PL/SQL program to implement Stored Procedure on table.
2. The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.
3. The procedure contains a header and a body.
4. Header: The header contains the name of the procedure and the parameters or variable passed to the procedure.
5. Body: The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

## How to pass parameters in procedure:

**IN parameters:** The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.

**OUT parameters:** The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.

**INOUT parameters:** The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

## SYNTAX:

```
CREATE [OR REPLACE] PROCEDURE procedure _name
[ (parameter [,parameter]) ]
(IS | AS)
[declaration_ section]
BEGIN
executable_ section
[EXCEPTION
exception_ section]
END [procedure_ name];
```

## EXAMPLE:

```
SQL> CREATE TABLE SAILOR2(ID NUMBER(10) PRIMARY KEY,NAME VARCHAR2(100));
Table created.

SQL>
```

## REPLACE PROCEDURE:

```
SQL> CREATE OR REPLACE PROCEDURE INSERTUSER
  2  (ID IN NUMBER,
  3  NAME IN VARCHAR2)
  4  IS
  5  BEGIN
  6  INSERT INTO SAILOR VALUES(ID,NAME);
  7  DBMS_OUTPUT.PUT_LINE('RECORD INSERTED SUCCESSFULLY');
  8  END;
  9  /

Procedure created.

SQL>
```

**EXECUTION PROCEDURE:**

```
SQL> EXEC INSERTUSER(&id,'&name')
Enter value for id: 102
Enter value for name: raju

PL/SQL procedure successfully completed.

SQL> set serverout on
SQL> /

Procedure created.

SQL> EXEC INSERTUSER(&id,'&name')
Enter value for id: 201
Enter value for name: ramu
RECORD INSERTED SUCCESSFULLY

PL/SQL procedure successfully completed.

SQL> EXEC INSERTUSER(&id,'&name')
Enter value for id: 301
Enter value for name: rama
RECORD INSERTED SUCCESSFULLY

PL/SQL procedure successfully completed.

SQL> EXEC INSERTUSER(&id,'&name')
Enter value for id: 401
Enter value for name: raja
RECORD INSERTED SUCCESSFULLY

PL/SQL procedure successfully completed.

SQL> EXEC INSERTUSER(&id,'&name')
Enter value for id: 501
Enter value for name: raju
RECORD INSERTED SUCCESSFULLY

PL/SQL procedure successfully completed.
```

**DROP PROCEDURE:**

```
SQL> DROP PROCEDURE insertuser;  
Procedure dropped.  
SQL>
```

**CONCLUSION:**

In this lab, we successfully practice how to write PL/SQL program to implement Stored Procedure on table.

# EXPERIMENT – 14

Write PL/SQL program to implement Stored Function on table.

## AIM :

To write PL/SQL program to implement Stored Function on table.

## PROGRAM DESCRIPTION :

In this lab, we learnt how to write PL/SQL program to implement Stored Function on table.

### PL/SQL Function

1. The PL/SQL Function is very similar to PL/SQL Procedure.
2. The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value.
3. Except this, all the other things of PL/SQL procedure
4. are true for PL/SQL function too.

### SYNTAX:

```
CREATE [OR REPLACE] FUNCTION function_name  
[ (parameter [,parameter]) ]  
RETURN return _ datatype  
(IS | AS)  
[declaration_section]  
BEGIN  
executable_section  
[EXCEPTION  
exception_section]  
END [procedure_name];
```

### IRRECURSIVE FUNCTION:

Irrecusive function is a function which does not calls itself.

### EXAMPLE:

```
SQL> CREATE OR REPLACE FUNCTION ADDER(N1 IN NUMBER, N2 IN NUMBER)
 2  RETURN NUMBER
 3  IS
 4  N3 NUMBER(8);
 5  BEGIN
 6  N3 :=N1+N2;
 7  RETURN N3;
 8  END;
 9 /

Function created.

SQL>
```

### Execution Procedure:

```
SQL> DECLARE
 2  N3 NUMBER(2);
 3  BEGIN
 4  N3 := ADDER(11,22);
 5  DBMS_OUTPUT.PUT_LINE('ADDITION IS: ' || N3);
 6  END;
 7 /
ADDITION IS: 33

PL/SQL procedure successfully completed.
```

### DROP FUNCTION

#### SYNTAX:

```
DROP FUNCTION Func_name;
```

#### EG:

```
SQL> DROP FUNCTION Adder;

Function dropped.

SQL>
```

## Recursive function:

Recursive function is a function which either calls itself

### EXAMPLE

```
SQL> CREATE FUNCTION fact(x number)
  2  RETURN number
  3  IS
  4  f number;
  5  BEGIN
  6  IF x=0 THEN
  7  f := 1;
  8  ELSE
  9  f := x * fact(x-1);
10 END IF;
11 RETURN f;
12 END;
13 /
```

```
Function created.
```

## Execution Procedure:

```
SQL> DECLARE
  2  num number;
  3  factorial number;
  4  BEGIN
  5  num:= 6;
  6  factorial := fact(num);
  7  dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
  8  END;
  9 /
Factorial 6 is 720

PL/SQL procedure successfully completed.
```

## DROP FUNCTION

```
SQL> DROP FUNCTION fact;

Function dropped.
```

## CONCLUSION:

In this lab, we successfully practice how to write PL/SQL program to implement Stored Function on table.

# EXPERIMENT – 15

Write PL/SQL program to implement Trigger on table.

## AIM:

To write PL/SQL program to implement Trigger on table.

## PROGRAM DESCRIPTION:

In this lab, we practice how to write PL/SQL program to implement Trigger on table.

Trigger is invoked by Oracle engine automatically whenever a specified event occurs. Trigger is stored into database and invoked repeatedly, when specific condition match. Triggers are stored programs, which are automatically executed or fired when some event occurs. Triggers are written to be executed in response to any of the following events.

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

## SYNTAX:

```
CREATE [OR REPLACE ] TRIGGER TRIGGER_NAME  
{BEFORE | AFTER | INSTEAD OF }  
{INSERT [OR] | UPDATE [OR] | DELETE}  
[OF COL_NAME]  
ON TABLE_NAME  
[REFERENCING OLD AS O NEW AS N]  
[FOR EACH ROW]  
WHEN (CONDITION)  
DECLARE  
DECLARATION-STATEMENTS  
BEGIN
```

## EXECUTABLE-STATEMENTS

### EXCEPTION

### EXCEPTION-HANDLING-STATEMENTS

END;

**CREATE [OR REPLACE] TRIGGER Trigger\_name:**

It creates or replaces an existing trigger with the trigger\_name.

{BEFORE | AFTER | INSTEAD OF} :

This specifies when the trigger would be executed. The INSTEAD OF clause is used for

creating trigger on a view.

{INSERT [OR] | UPDATE [OR] | DELETE}:

This specifies the DML operation.

[OF col\_name]:

This specifies the column name that would be updated.

[ON table\_name]:

This specifies the name of the table associated with the trigger.

[REFERENCING OLD AS o NEW AS n]:

This allows you to refer new and old values for various DML statements, like INSERT,

UPDATE, and DELETE.

[FOR EACH ROW]:

This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.

**WHEN (condition):**

This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

**PL/SQL Trigger Example:****Instructor Table**

```
SQL> CREATE TABLE INSTRUCTOR1
  2  (
  3  ID VARCHAR2(5),
  4  NAME VARCHAR2(20) NOT NULL,
  5  DEPT_NAME VARCHAR2(20),
  6  SALARY NUMERIC(8,2) CHECK (SALARY > 29000),
  7  PRIMARY KEY (ID),
  8  FOREIGN KEY (DEPT_NAME) REFERENCES DEPARTMENT(DEPT_NAME)
  9  ON DELETE SET NULL
 10 );
```

Table created.

**INSTANCES OF INSTRUCTOR TABLE**

```
SQL> insert into instructor values ('83821', 'Brandt', 'Comp. Sci.', '92000');
1 row created.
```

**SELECT:**

```
SQL> SELECT * FROM INSTRUCTOR1;
-----
```

ID	NAME	DEPT_NAME	SALARY
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

**DEPARTMENT TABLE:**

```
SQL> CREATE TABLE DEPARTMENT1
  2  (
  3  DEPT_NAME VARCHAR2(20),
  4  BUILDING VARCHAR2(15),
  5  BUDGET NUMERIC(12,2) CHECK (BUDGET > 0),
  6  PRIMARY KEY (DEPT_NAME)
  7 );
```

Table created.

**INSTANCES OF DEPARTMENT TABLE:**

```
SQL> INSERT INTO department1 VALUES ('Biology', 'Watson', '90000');  
1 row created.
```

**SELECT:**

```
SQL> SELECT * FROM DEPARTMENT1;  
  
DEPT_NAME          BUILDING      BUDGET  
-----  
Biology            Watson        90000  
Comp. Sci.          Taylor       100000  
Elec. Eng.          Taylor       85000  
Finance            Painter      120000  
History             Painter      50000  
Music               Packard     80000  
Physics             Watson       70000  
  
7 rows selected.
```

**An example to create Trigger:**

```
SQL> CREATE OR REPLACE TRIGGER display_salary_changes  
2 BEFORE UPDATE ON instructor  
3 FOR EACH ROW  
4 WHEN (NEW.ID = OLD.ID)  
5 DECLARE  
6 sal_diff number;  
7 BEGIN  
8 sal_diff := :NEW.salary - :OLD.salary;  
9 dbms_output.put_line('Old salary: ' || :OLD.salary);  
10 dbms_output.put_line('New salary: ' || :NEW.salary);  
11 dbms_output.put_line('Salary difference: ' || sal_diff);  
12 END;  
13 /  
  
Trigger created.
```

### A PL/SQL Procedure to execute a trigger

```
SQL> DECLARE
 2  total_rows number(2);
 3  BEGIN
 4    UPDATE instructor
 5    SET salary = salary + 5000;
 6    IF sql%notfound THEN
 7      dbms_output.put_line('no instructors updated');
 8    ELSIF sql%found THEN
 9      total_rows := sql%rowcount;
10    dbms_output.put_line( total_rows || ' instructors updated ');
11  END IF;
12 END;
13 /
```

PL/SQL procedure successfully completed.

### OUTPUT:

```
SQL> SET SERVEROUT ON
SQL> /
Old salary: 70000
New salary: 75000
Salary difference: 5000
Old salary: 80000
New salary: 85000
Salary difference: 5000
Old salary: 97000
New salary: 102000
Salary difference: 5000
3 instructors updated

PL/SQL procedure successfully completed.
```

SQL%FOUND, SQL%NOTFOUND, and SQL%ROWCOUNT are PL/SQL attributes that can be used to determine the effect of an SQL statement.

The SQL%FOUND attribute has a Boolean value that returns TRUE if at least one row was affected by an INSERT, UPDATE, or DELETE statement, or if a SELECT INTO statement retrieved one row.

The SQL%NOTFOUND attribute has a Boolean value that returns TRUE if no rows were affected by an INSERT, UPDATE, or DELETE statement, or if a SELECT INTO statement did not retrieve a row.

The SQL%ROWCOUNT attribute has an integer value that represents the number of rows that were affected by an INSERT, UPDATE, or DELETE statement.

## **DROP the Trigger**

### **Syntax:**

```
DROP trigger trigger_name;
```

### **EXAMPLE:**

```
SQL> DROP trigger display_salary_changes;  
Trigger dropped.
```

### **CONCLUSION:**

In this lab, we successfully practice how to write PL/SQL program to implement Trigger on table.

# EXPERIMENT – 16

Write PL/SQL program to implement Cursor on table.

## AIM:

To write PL/SQL program to implement Cursor on table.

## PROGRAM DESCRIPTION:

In this lab, we practice how to write PL/SQL program to implement Cursor on table.

When an SQL statement is processed, Oracle creates a memory area known as context area. A cursor is a pointer to this context area. It contains all information needed for processing the statement. In PL/SQL, the context area is controlled by Cursor. A cursor contains information on a select statement and the rows of data accessed by it.

## Types of Cursor

1. Implicit cursor
2. Explicit cursor

The implicit cursors are automatically generated by Oracle while an SQL statement is executed, if you don't use an explicit cursor for the statement.

## Table Creation:

```
SQL> CREATE TABLE customers(
  2  ID NUMBER PRIMARY KEY,
  3  NAME VARCHAR2(20) NOT NULL,
  4  AGE NUMBER,
  5  ADDRESS VARCHAR2(20),
  6  SALARY NUMERIC(20,2)
  7  );
```

```
Table created.
```

## Instances of Customers:

```
SQL> INSERT INTO customers VALUES(6, 'Sunita',20,'delhi',35000);
1 row created.
```

**SELECT:**

```
SQL> SELECT * FROM CUSTOMERS;
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allabad	25000
2	Suresh	22	Kanpur	27000
3	Mahesh	24	Ghaziabad	29000
4	chandhan	25	Noida	31000
5	Alex	21	paris	33000
6	Sunita	20	delhi	35000

6 rows selected.

**Create update procedure****Create procedure:**

```
SQL> DECLARE
 2  total_rows number(2);
 3  BEGIN
 4  UPDATE customers
 5  SET salary = salary + 5000;
 6  IF sql%notfound THEN
 7  dbms_output.put_line('no customers updated');
 8  ELSIF sql%found THEN
 9  total_rows := sql%rowcount;
10  dbms_output.put_line( total_rows || ' customers updated ');
11 END IF;
12 END;
13 /
6 customers updated

PL/SQL procedure successfully completed.
```

**SELECT:**

```
SQL> SELECT * FROM CUSTOMERS;
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allabad	30000
2	Suresh	22	Kanpur	32000
3	Mahesh	24	Ghaziabad	34000
4	chandhan	25	Noida	36000
5	Alex	21	paris	38000
6	Sunita	20	delhi	40000

6 rows selected.

## PL/SQL Explicit Cursors

The Explicit cursors are defined by the programmers to gain more control over the context area. These cursors should be defined in the declaration section of the PL/SQL block. It is created on a SELECT statement which returns more than one row.

### SYNTAX:

CURSOR cursor\_ name IS select\_ statement;

### Steps:

You must follow these steps while working with an explicit cursor.

1. Declare the cursor to initialize in the memory.
2. Open the cursor to allocate memory.
3. Fetch the cursor to retrieve data.
4. Close the cursor to release allocated memory.

#### 1) Declare the cursor:

##### SYNTAX:

CURSOR cursor \_ name IS select\_ statement;

#### 2) Open the cursor:

It is used to allocate memory for the cursor and make it easy to fetch the rows returned by the SQL statements into it.

##### SYNTAX:

OPEN cursor\_ name;

#### 3) Fetch the cursor:

It is used to access one row at a time. You can fetch rows from the above opened cursor as follows:

##### SYNTAX:

FETCH cursor\_ name INTO variable \_ list;

#### 4) Close the cursor:

It is used to release the allocated memory. The following syntax is used to close the above opened cursors.

##### SYNTAX:

Close cursor\_ name;

### PL/SQL Program using Explicit Cursors

```
SQL> DECLARE
 2  c_id customers.id%type;
 3  c_name customers.name%type;
 4  c_addr customers.address%type;
 5  CURSOR c_customers is
 6  SELECT id, name, address FROM customers;
 7  BEGIN
 8  OPEN c_customers;
 9  LOOP
10  FETCH c_customers into c_id, c_name, c_addr;
11  EXIT WHEN c_customers%notfound;
12  dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
13  END LOOP;
14  CLOSE c_customers;
15  END;
16  /
```

### OUTPUT:

```
1 Ramesh Allabad
2 Suresh Kanpur
3 Mahesh Ghaziabad
4 chandhan Noida
5 Alex paris
6 Sunita delhi

PL/SQL procedure successfully completed.
```