

EXPERIMENT-1

Aim: Write SQL queries to create for various databases using DDL commands (i.e. CREATE, ALTER, DROP, TRUNCATE).

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]);
```

```
SQL> CREATE TABLE student(
  2  s_name VARCHAR2(10),
  3  s_id VARCHAR(5),
  4  s_branch VARCHAR(5),
  5  s_block VARCHAR(5),
  6  PRIMARY KEY(s_id)
  7 )
  8 ;

Table created.

SQL>
SQL> |
```

```
SQL> DESC student;
Name          Null?    Type
-----        -----
S_NAME        *        VARCHAR2(10)
S_ID          NOT NULL VARCHAR2(5)
S_BRANCH      VARCHAR2(5)
S_BLOCK       VARCHAR2(5)

SQL> |
```

ALTER TABLE:

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

```
SQL> ALTER TABLE student
  2 ADD s_fee NUMBER NOT NULL;

Table altered.

SQL> DESC student;
Name          Null?    Type
-----        -----
S_NAME        *        VARCHAR2(10)
S_ID          NOT NULL VARCHAR2(5)
S_BRANCH      VARCHAR2(5)
S_BLOCK       VARCHAR2(5)
S_FEE         NOT NULL NUMBER

SQL>
```

DROP TABLE:

Deletes the specified table.

Syntax:

`DROP TABLE table_name;`

```
SQL> CREATE TABLE instructor2(
  2 i_name VARCHAR2(10),
  3 i_subject VARCHAR(10),
  4 i_salary NUMBER(10),
  5 i_branch VARCHAR(5)
  6 );
```

Table created.

```
SQL> DROP TABLE instructor2;
```

Table dropped.

RENAME TABLE:

To rename table_name, column_name.

Syntax:

```
RENAME new_table_name TO old_table_name;
```

```
SQL> RENAME student TO students;
Table renamed.

SQL> DESC students;
Name          Null?    Type
-----          -----
S_NAME        VARCHAR2(10)
S_ID          NOT NULL VARCHAR2(5)
S_BRANCH      VARCHAR2(5)
S_BLOCK       VARCHAR2(5)
S_FEE         NOT NULL NUMBER

SQL> |
```

TRUNCATE TABLE:

To remove all rows in a specified table.

Syntax:

```
TRUNCATE TABLE table_name;
```

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

SQL> |
```

Conclusion:

In this lab, we have practiced CREATE, ALTER, DROP, and TRUNCATE commands for the user created table.

EXPERIMENT-2

AIM: To write SQL queries to MANIPULATE TABLES for various databases using DML commands (i.e. INSERT, SELECT, UPDATE, DELETE).

CREATING TABLE:

```
SQL> CREATE TABLE orders(
  2  o_name VARCHAR(10),
  3  o_id VARCHAR(10) NOT NULL,
  4  o_place VARCHAR2(20),
  5  PRIMARY KEY(o_id)
  6  );
```

```
Table created.
```

```
SQL>
```

INSERT COMMAND:

It is used to add values to a table.

Syntax:

```
INSERT INTO table_name
```

```
VALUES (value1, value2, ...., valueN);
```

```
INSERT INTO table_name (column1, column2, ...., columnN)
```

```
VALUES (value1, value2, ...., valueN);
```

```
SQL> INSERT INTO orders(o_name,o_id,o_place)
  2  VALUES('FRIED RICE',2424,'KLD ROAD');

1 row created.

SQL> INSERT INTO orders(o_name,o_id,o_place)
  2  VALUES('GOBI RICE',2038,'GANESH NAGAR');

1 row created.

SQL> INSERT INTO orders(o_name,o_id,o_place)
  2  VALUES('MEALS',5456,'RAM NAGAR');

1 row created.

SQL>
```

SELECT COMMAND:

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

Syntax:

```
SELECT * FROM table_name;
```

```
SELECT column_name FROM table_name;
```

```
SQL> SELECT * FROM ORDERS;

O_NAME      O_ID      O_PLACE
-----  -----  -----
FRIED RICE  2424      KLD ROAD
GOBI RICE   2038      GANESH NAGAR
MEALS       5456      RAM NAGAR

SQL>
```

```
SQL> SELECT o_id FROM orders;

O_ID
-----
2424
2038
5456

SQL>
```

UPDATE COMMAND:

The UPDATE command is used to modify the contents of specified table.

Syntax:

UPDATE table_name

SET column_name = value [,

Column_name = value]

[WHERE condition_list];

```
SQL> UPDATE orders SET o_place='AZAD NAGAR' WHERE o_id=2038;
1 row updated.

SQL> SELECT*FROM orders;
O_NAME      O_ID      O_PLACE
-----  -----  -----
FRIED RICE  2424      KLD ROAD
GOBI RICE   2038      AZAD NAGAR
MEALS       5456      RAM NAGAR

SQL> |
```

DELETE COMMAND:

To delete all rows or specified rows in a table.

Syntax:

DELETE FROM table_name [WHERE condition_list];

```
SQL> DELETE FROM orders WHERE o_id=2038;
1 row deleted.

SQL> SELECT*FROM orders;
O_NAME      O_ID      O_PLACE
-----  -----  -----
FRIED RICE  2424      KLD ROAD
MEALS       5456      RAM NAGAR

SQL> |
```

CONCLUSION:

In this lab, we have practiced INSERT, SELECT, UPDATE, and DELETE commands for user created table.

EXPERIMENT-3

AIM: To implement view high level design for various views to CREATE VIEW, ALTER VIEW and DELETE VIEW using DDL commands.

CREATING TABLE:

```
SQL> CREATE TABLE student1(
  2  s_id VARCHAR2(10),
  3  s_name VARCHAR2(20),
  4  s_branch VARCHAR2(20)
  5 );
```

```
Table created.
```

```
SQL>
```

INSERTING VALUES INTO TABLE student1:

```
SQL> INSERT INTO student1 VALUES(220,'Alluri','CSE');

1 row created.
```

```
SQL> INSERT INTO student1 VALUES(221,'Sita','CSM');
```

```
1 row created.
```

```
SQL> INSERT INTO student1 VALUES(222,'Siva','CSD');
```

```
1 row created.
```

```
SQL> SELECT * FROM student1;
```

S_ID	S_NAME	S_BRANCH
220	Alluri	CSE
221	Sita	CSM
222	Siva	CSD

```
SQL> |
```

CREATE VIEW: Create a view details with attributes s_id and s_name.

```
SQL> CREATE VIEW details AS SELECT s_id,s_name FROM student1;  
View created.  
SQL> |
```

INSERTING VALUES INTO details view:

```
SQL> INSERT INTO details(s_id,s_name) VALUES(223,'RAJU');  
1 row created.  
SQL> INSERT INTO details(s_id,s_name) VALUES(224,'RAMA');  
1 row created.  
SQL> |
```

```
SQL> SELECT * FROM details;
```

S_ID	S_NAME
220	Alluri
221	Sita
222	Siva
223	RAJU
224	RAMA

```
SQL> |
```

ALTER VIEW: Add an attribute branch to the view.

```
SQL> CREATE VIEW details2 AS SELECT s_id,s_name,s_branch FROM student1 WHERE s_id=222;  
View created.  
SQL> SELECT * FROM details2 WHERE s_id=222;  


| S_ID | S_NAME | S_BRANCH |
|------|--------|----------|
| 222  | Siva   | CSD      |

  
SQL> |
```

UPDATE VIEW: Update the existing name with the new name using UPDATE command.

```
SQL> UPDATE details SET s_name='Ram' WHERE s_id=224;  
1 row updated.  
  
SQL> SELECT * FROM details;  
  
S_ID      S_NAME  
-----  
220       Alluri  
221       Sita  
222       Siva  
223       RAJU  
224       Ram  
  
SQL> |
```

DROP VIEW: Drop a view using DROP command.

```
SQL> DROP VIEW details2;  
View dropped.  
  
SQL> |
```

Conclusion:

In this lab, we have practiced how to CREATE VIEW, ALTER VIEW, UPDATE VIEW, and DELETE VIEW for user created table.

EXPERIMENT-4

AIM: To implement SQL queries for set operations like UNION, UNION ALL, INTERSECT, INTERSECT ALL, MINUS, CROSS JOIN, NATURAL JOIN.

CREATING A TABLE **instructor2**:

```
SQL> CREATE TABLE instructor2(
  2  i_id VARCHAR2(10),
  3  i_name VARCHAR2(20),
  4  i_branch VARCHAR2(20)
  5 );
```

```
Table created.
```

INSERTING VALUES INTO TABLE **instructor2**:

```
SQL> INSERT INTO instructor2 VALUES(441,'NEHA','CSE');
1 row created.

SQL> INSERT INTO instructor2 VALUES(442,'NANI','CSD');
1 row created.

SQL> INSERT INTO instructor2 VALUES(443,'MEENA','CSM');
1 row created.

SQL> SELECT * FROM instructor2;
I_ID      I_NAME        I_BRANCH
-----  -----
441       NEHA          CSE
442       NANI          CSD
443       MEENA         CSM
```

CREATING TABLE **department**:

```
SQL> CREATE TABLE department(
 2  d_id VARCHAR2(5),
 3  d_name VARCHAR2(20),
 4  d_budget NUMBER(10,2)
 5 );
```

```
Table created.
```

INSERTING VALUES INTO department:

```
SQL> INSERT INTO department VALUES(451,'CSE',75000);
1 row created.

SQL> INSERT INTO department VALUES(452,'CIVIL',85000);
1 row created.

SQL> INSERT INTO department VALUES(453,'MECH',80000);
1 row created.

SQL> SELECT * FROM department;
```

D_ID	D_NAME	D_BUDGET
451	CSE	75000
452	CIVIL	85000
453	MECH	80000

```
SQL> |
```

UNION: The attributes i_branch from instructor2 and d_name from department are joined using UNION command.

```
SQL> SELECT i_branch FROM instructor2
 2 UNION
 3 SELECT d_NAME FROM department;
```

I_BRANCH
CSE
CSD
CSM
CIVIL
MECH

```
SQL> |
```

UNION ALL: The attributes i_branch from instructor2 and d_name from department are joined along with duplicates using UNION ALL command.

```
SQL> SELECT i_branch FROM instructor2
  2 UNION ALL
  3 SELECT d_NAME FROM department;

I_BRANCH
-----
CSE
CSD
CSM
CSE
CIVIL
MECH

6 rows selected.

SQL> |
```

INTERSECT: Displays similar values in two or more attributes from department and instructor4 using INTERSECT command.

```
SQL> SELECT i_branch FROM instructor2
  2 INTERSECT
  3 SELECT d_NAME FROM department;

I_BRANCH
-----
CSE

SQL> |
```

MINUS: It eliminates the same values of second column from the first column and represents the remaining values using command MINUS.

```
SQL> SELECT i_branch FROM instructor2
  2 MINUS
  3 SELECT d_NAME FROM department;

I_BRANCH
-----
CSD
CSM

SQL> |
```

CROSS JOIN: Its cross products the all the attributes using CROSS JOIN command.

```
SQL> SELECT i_id,i_name,i_branch  
2  FROM instructor2 i,department d;
```

I_ID	I_NAME	I_BRANCH
441	NEHA	CSE
441	NEHA	CSE
441	NEHA	CSE
442	NANI	CSD
442	NANI	CSD
442	NANI	CSD
443	MEENA	CSM
443	MEENA	CSM
443	MEENA	CSM

```
9 rows selected.
```

```
SQL> |
```

Conclusion:

In this lab, we have practiced the set operations like UNION, UNION ALL, INTERSECT, MINUS, CROSS JOIN on user created tables.

EXPERIMENT-5

AIM:SQL queries to perform SPECIAL OPERATIONS (IS NULL, BETWEEN, LIKE, IN, EXISTS).

CREATING TABLE-1:

```
SQL> CREATE TABLE instructors(
 2  id VARCHAR2(10),
 3  name VARCHAR2(20) NOT NULL,
 4  dept_name VARCHAR2(10),
 5  salary NUMERIC(8,2)
 6  );
```

```
Table created.
```

```
SQL>
```

INSERTING VALUES INTO THE TABLE:

```
SQL> INSERT INTO instructors VALUES(801,'GOPI','CSE',45000);
1 row created.

SQL> INSERT INTO instructors VALUES(802,'SIVA','CSM',64000);
1 row created.

SQL> INSERT INTO instructors VALUES(803,'KESI','CSD',87000);
1 row created.

SQL> INSERT INTO instructors VALUES(804,'TEJU','MECH',77000);
1 row created.

SQL> INSERT INTO instructors VALUES(805,'JOTISH','CIVIL',67000);
1 row created.

SQL> SELECT * FROM instructors;

ID      NAME        DEPT_NAME    SALARY
-----  -----       -----       -----
801     GOPI        CSE          45000
802     SIVA         CSM          64000
803     KESI         CSD          87000
804     TEJU         MECH         77000
805     JOTISH       CIVIL        67000
806     LUCKY        NULL         NULL

6 rows selected.

SQL>
```

IS NULL: It is used to check null values and display null attributes. It displays attributes that have null values.

```
SQL> SELECT * FROM instructors WHERE salary IS NULL;

ID      NAME        DEPT_NAME    SALARY
-----  -----       -----       -----
806     LUCKY        NULL         NULL

SQL> |
```

This command displays the salary that are not equal to 64000.

```
SQL> SELECT * FROM instructors WHERE salary <>64000;
```

ID	NAME	DEPT_NAME	SALARY
801	GOPI	CSE	45000
803	KESI	CSD	87000
804	TEJU	MECH	77000
805	JOTISH	CIVIL	67000

```
SQL> |
```

IS NOT NULL: It displays attributes that don't have null values.

```
SQL> SELECT * FROM instructors WHERE salary IS NOT NULL;
```

ID	NAME	DEPT_NAME	SALARY
801	GOPI	CSE	45000
802	SIVA	CSM	64000
803	KESI	CSD	87000
804	TEJU	MECH	77000
805	JOTISH	CIVIL	67000

```
SQL> |
```

BETWEEN: This is used to check range of values.

By following command, it displays all the attributes between 45000 and 77000.

```
SQL> SELECT * FROM instructors WHERE salary BETWEEN 45000 AND 77000;
```

ID	NAME	DEPT_NAME	SALARY
801	GOPI	CSE	45000
802	SIVA	CSM	64000
804	TEJU	MECH	77000
805	JOTISH	CIVIL	67000

```
SQL> |
```

The following command displays salary that are not in between 45000 and 77000.

```
SQL> SELECT * FROM instructors WHERE salary NOT BETWEEN 45000 AND 77000;
```

ID	NAME	DEPT_NAME	SALARY
803	KESI	CSD	87000

```
SQL>
```

IN: This is used to check a member is in a set or not. It displays if the id's are present in the table.

```
SQL> SELECT * FROM instructors WHERE ID IN ('803','802','804');
ID          NAME        DEPT_NAME    SALARY
-----      -----
802         SIVA        CSM          64000
803         KESI        CSD          87000
804         TEJU        MECH         77000
SQL>
```

The following command displays all the attributes with id's except the given id's.

```
SQL> SELECT * FROM instructors WHERE ID NOT IN ('803','802','804');
ID          NAME        DEPT_NAME    SALARY
-----      -----
801         GOPI       CSE          45000
805         JOTISH     CIVIL        67000
806         LUCKY      MECH         67000
SQL>
```

EXISTS: This is used to check whether given set is empty or not. It displays null attributes that are null according to the given condition.

```
SQL> SELECT * FROM instructors WHERE EXISTS
  2 (SELECT * FROM instructors WHERE dept_name IS NULL);
ID          NAME        DEPT_NAME    SALARY
-----      -----
801         GOPI       CSE          45000
802         SIVA        CSM          64000
803         KESI        CSD          87000
804         TEJU        MECH         77000
805         JOTISH     CIVIL        67000
806         LUCKY      MECH         67000
6 rows selected.
SQL>
```

LIKE: This is used to check given string is present or not. It displays all the attributes that start with character 'c'.

```
SQL> SELECT * FROM instructors WHERE dept_name LIKE 'C%';
ID          NAME        DEPT_NAME    SALARY
-----      -----
801         GOPI       CSE          45000
802         SIVA        CSM          64000
803         KESI        CSD          87000
805         JOTISH     CIVIL        67000
SQL> |
```

Conclusion: In this lab, we have practiced SPECIAL OPERATIONS IS NULL, BETWEEN, LIKE, IN, EXISTS on user created table.

EXPERIMENT-6

AIM: To implement SQL queries to perform JOIN OPERATIONS (CONDITIONAL JOIN, EQUI JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN).

CREATING TABLE-1:

```
SQL> CREATE TABLE student(
 2  id NUMBER,
 3  name VARCHAR2(20),
 4  branch VARCHAR2(20)
 5  );
```

```
Table created.
```

```
SQL> |
```

INSERTING VALUES INTO THE TABLE:

```
SQL> INSERT INTO student VALUES(81,'RAJU','CSE');
1 row created.

SQL> INSERT INTO student VALUES(82,'RANI','CSM');
1 row created.

SQL> INSERT INTO student VALUES(83,'MANI','CSD');
1 row created.

SQL> INSERT INTO student VALUES(84,'NANI','CIVIL');
1 row created.

SQL>
```

```
SQL> SELECT * FROM student;
```

ID	NAME	BRANCH
81	RAJU	CSE
82	RANI	CSM
83	MANI	CSD
84	NANI	CIVIL

```
SQL> |
```

CREATING TABLE-2:

```
SQL> CREATE TABLE library(
  2  id NUMBER,
  3  book_name VARCHAR2(20)
  4  );
```

```
Table created.
```

```
SQL>
```

INSERTING VALUES INTO TABLE-2:

```
SQL> INSERT INTO library VALUES(81,'JAVA');
1 row created.

SQL> INSERT INTO library VALUES(82,'DBMS');
1 row created.

SQL> INSERT INTO library VALUES(83,'SE');
1 row created.

SQL> SELECT * FROM library;
  ID BOOK_NAME
----- 
  81 JAVA
  82 DBMS
  83 SE
```

```
SQL> |
```

CONDITIONAL JOIN: It helps in retrieving the desired data and performing complex queries.

```
SQL> SELECT * FROM student JOIN library USING(id);
  ID NAME          BRANCH      BOOK_NAME
----- 
  81 RAJU          CSE         JAVA
  82 RANI          CSM         DBMS
  83 MANI          CSD         SE
```

```
SQL>
```

EQUI JOIN: It helps in retrieving related information from different tables by matching corresponding values.

```
SQL> SELECT * FROM student JOIN library USING(id);
```

ID	NAME	BRANCH	BOOK_NAME
81	RAJU	CSE	JAVA
82	RANI	CSM	DBMS
83	MANI	CSD	SE

```
SQL>
```

LEFT OUTER JOIN: It combines data from two or more tables based on the matching values in specified columns, but it also includes unmatched rows from the left table.

```
SQL> SELECT * FROM student NATURAL LEFT OUTER JOIN library;
```

ID	NAME	BRANCH	BOOK_NAME
81	RAJU	CSE	JAVA
82	RANI	CSM	DBMS
83	MANI	CSD	SE
84	NANI	CIVIL	

```
SQL> |
```

RIGHT OUTER JOIN: It combines data from two or more tables based on the matching values in specified columns, but also includes unmatched rows from the right table.

```
SQL> SELECT * FROM student NATURAL RIGHT OUTER JOIN library;
```

ID	NAME	BRANCH	BOOK_NAME
81	RAJU	CSE	JAVA
82	RANI	CSM	DBMS
83	MANI	CSD	SE

```
SQL>
```

FULL OUTER JOIN: It includes all the rows from both the left and right tables, even if there is no match.

```
SQL> SELECT * FROM student NATURAL FULL OUTER JOIN library;
```

ID	NAME	BRANCH	BOOK_NAME
81	RAJU	CSE	JAVA
82	RANI	CSM	DBMS
83	MANI	CSD	SE
84	NANI	CIVIL	

```
SQL> |
```

Conclusion: In this lab, we have practiced JOIN OPERATIONS CONDITIONAL JOIN, EQUI JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN on user created tables.

EXPERIMENT-7

AIM: To implement SQL queries to perform AGGREGATE OPERATIONS (SUM, COUNT, AVG, MIN, MAX).

CREATING A TABLE:

```
SQL> CREATE TABLE employee(
 2  ID VARCHAR2(5),
 3  NAME VARCHAR2(20),
 4  FIELD VARCHAR2(20),
 5  SALARY NUMERIC(8,2)
 6  );
```

```
Table created.
```

```
SQL> |
```

INSERTING VALUES INTO TABLE:

```
SQL> INSERT INTO employee VALUES(661,'NEHA','DEVELOPER',60000);
1 row created.

SQL> INSERT INTO employee VALUES(662,'NANI','TESTER',62000);
1 row created.

SQL> INSERT INTO employee VALUES(663,'DEVI','MANAGER',70000);
1 row created.

SQL> INSERT INTO employee VALUES(664,'TEJU','REVERSE ENGINEER',75000);
1 row created.

SQL> INSERT INTO employee VALUES(665,'JOTISH','TESTER',66000);
1 row created.

SQL>
```

```
SQL> SELECT * FROM employee;
ID      NAME          FIELD          SALARY
-----  -----          -----          -----
661     NEHA          DEVELOPER      60000
662     NANI          TESTER        62000
663     DEVI          MANAGER       70000
664     TEJU          REVERSE ENGINEER 75000
665     JOTISH        TESTER        66000
SQL>
```

COUNT: It displays the count on members present in employee.

```
SQL> SELECT COUNT(*) FROM employee;
COUNT(*)
-----
5
SQL> |
```

AVERAGE(AVG): It displays average salary of each employee.

```
SQL> SELECT FIELD, AVG(SALARY) AS AVG_SALARY
  2  FROM employee
  3  GROUP BY FIELD;
FIELD          AVG_SALARY
-----          -----
DEVELOPER      60000
TESTER        64000
MANAGER       70000
REVERSE ENGINEER 75000
SQL> |
```

SUM: It displays sum of all the salaries from the table.

```
SQL> SELECT SUM(SALARY) FROM employee;
SUM(SALARY)
-----
333000
SQL>
```

MIN: It displays the minimum salary from the table.

```
SQL> SELECT MIN(SALARY) FROM employee;  
  
MIN(SALARY)  
-----  
       60000  
  
SQL>
```

MAX: It displays the maximum salary from the table.

```
SQL> SELECT MAX(SALARY) FROM employee;  
  
MAX(SALARY)  
-----  
      75000  
  
SQL>
```

Conclusion: In this lab, we have practiced AGGREGATE OPERATIONS like SUM, COUNT, AVERAGE, MIN, MAX on user created table.

EXPERIMENT-8

AIM: To implement SQL queries to perform BUILT-IN FUNCTIONS (DATE, TIME).

CASE CONVERSION:

LOWER (): It converts a string into lowercase.

```
SQL> SELECT LOWER('HELLO WORLD') FROM DUAL;  
  
LOWER('HELL  
-----  
hello world  
  
SQL>
```

UPPER (): It converts a string into uppercase.

```
SQL> SELECT UPPER('Hello world') FROM DUAL;  
  
UPPER('HELL  
-----  
HELLO WORLD  
  
SQL>
```

INITCAP (): It converts a string into camel case.

```
SQL> SELECT INITCAP('hello world') FROM DUAL;  
INITCAP('HE  
-----  
Hello World  
  
SQL>
```

CONCAT (): It adds two or more expressions together.

```
SQL> SELECT CONCAT('Hello', 'World') FROM DUAL;  
CONCAT('HE  
-----  
HelloWorld  
  
SQL> |
```

SUBSTR (): It extracts a substring from a string.

```
SQL> SELECT SUBSTR('Hello World', 1, 5) FROM DUAL;  
SUBST  
-----  
Hello  
  
SQL> |
```

LENGTH (): It returns the length of the given string.

```
SQL> SELECT LENGTH('Hello World') FROM DUAL;  
LENGTH('HELLOWORLD')  
-----  
           11  
  
SQL> |
```

INSTR (): It returns the position or the first occurrence of a string in another string.

```
SQL> SELECT INSTR('Hello World', 'Hello') FROM DUAL;  
INSTR('HELLOWORLD', 'HELLO')  
-----  
           1  
  
SQL> |
```

TRIM (): It removes the selected one from string.

```
SQL> SELECT TRIM('H' FROM 'Hello World') FROM DUAL;  
-----  
ello World  
SQL> |
```

NUMBER FUNCTIONS:

ROUND (): It returns the specified values.

```
SQL> SELECT ROUND(87.9865) FROM DUAL;  
-----  
ROUND(87.9865)  
-----  
88  
SQL> SELECT ROUND(87.9865,2) FROM DUAL;  
-----  
ROUND(87.9865,2)  
-----  
87.99  
SQL> |
```

TRUNCATE (): It removes the decimal values which are specified.

```
SQL> SELECT TRUNC(87.9865,0) FROM DUAL;  
-----  
TRUNC(87.9865,0)  
-----  
87  
SQL> |
```

MOD (): It returns the remainder.

```
SQL> SELECT MOD(1600,33) FROM DUAL;  
-----  
MOD(1600,33)  
-----  
16  
SQL>
```

DATE FUNCTIONS:

SYSDATE ():

```
SQL> SELECT SYSDATE FROM DUAL;
```

```
SYSDATE
```

```
-----  
12-JAN-24
```

```
SQL> |
```

MONTHS_BETWEEN ():

```
SQL> SELECT MONTHS_BETWEEN(SYSDATE, '1-JAN-24') FROM DUAL;
```

```
MONTHS_BETWEEN(SYSDATE, '1-JAN-24')
```

```
-----  
.372958109
```

```
SQL> |
```

ADD_MONTHS ():

```
SQL> SELECT ADD_MONTHS(SYSDATE, 5) FROM DUAL;
```

```
ADD_MONTH
```

```
-----  
12-JUN-24
```

```
SQL> |
```

NEXT_DAY ():

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

```
NEXT_DAY
```

```
-----
```

```
19-JAN-24
```

```
SQL> |
```

LAST_DAY ():

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

```
LAST_DAY
```

```
-----
```

```
31-JAN-24
```

```
SQL> |
```

TRUNC ():

```
SQL> SELECT TRUNC(SYSDATE, 'DAY') FROM DUAL;  
TRUNC(SYS  
-----  
07-JAN-24  
SQL> |
```

Conclusion: In this lab, we have practiced BUILT-IN FUNCTIONS like DATE AND TIME .

EXPERIMENT-9

AIM: To implement SQL queries to perform KEY CONSTRAINTS (PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, CHECK, DEFAULT).

PRIMARY KEY: A primary key is a field which can uniquely identify each row in table and this constraint is used to specify a field as primary key.

```
SQL> CREATE TABLE student5(  
 2 ID NUMBER,  
 3 NAME VARCHAR2(20),  
 4 ADDRESS VARCHAR2(20)  
 5 );  
  
Table created.  
  
SQL>
```

FOREIGN KEY: A foreign key is a field which can uniquely identify each row in another table.

```
SQL> CREATE TABLE orders5(  
 2 o_id NUMBER NOT NULL,  
 3 c_id NUMBER,  
 4 PRIMARY KEY(o_id),  
 5 FOREIGN KEY(c_id)REFERENCES customer(c_id)  
 6 );  
  
Table created.  
  
SQL>
```

UNIQUE: This constraint when specified with a column, tells that the values in the column must be unique i.e., the values in any row of a column must not be repeated.

```
SQL> CREATE TABLE student3(
  2  id NUMBER UNIQUE,
  3  name VARCHAR2(20),
  4  address VARCHAR2(20)
  5  );
```

Table created.

```
SQL>
```

NOT NULL: This constraint tells that we cannot store a null value in a column.

```
SQL> CREATE TABLE student3(
  2  ID NUMBER,
  3  NAME VARCHAR2(20) NOT NULL,
  4  ADDRESS VARCHAR2(20)
  5  );
```

Table created.

```
SQL> |
```

DEFAULT: This constraint specifies a default value for the column when no value is specified by the user.

```
SQL> CREATE TABLE student6(
  2  ID NUMBER,
  3  NAME VARCHAR2(20) NOT NULL,
  4  AGE NUMBER DEFAULT 18
  5  );
```

Table created.

```
SQL>
```

CHECK: This constraint helps to validate the value for the column to meet a particular condition i.e. it helps to ensure that the value stored in a column meets a specific condition.

```
SQL> CREATE TABLE student8(
  2  id NUMBER NOT NULL,
  3  NAME VARCHAR2(20) NOT NULL,
  4  AGE NUMBER NOT NULL CHECK(AGE>=18)
  5  );
```

Table created.

```
SQL> |
```

Conclusion: In this lab, we have practiced KEY CONSTRAINTS PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, CHECK, DEFAULT on user created tables.

EXPERIMENT-10

AIM: To write a PL/SQL program for calculating the factorial of a given number.

SOURCE CODE & OUTPUT:

```
SQL> DECLARE
 2  FACT NUMBER:=1;
 3  N NUMBER;
 4  N1 NUMBER;
 5  BEGIN
 6  N:=&N;
 7  N1:=N;
 8  WHILE N>0 LOOP
 9  FACT:=N*FACT;
10  N:=N-1;
11  END LOOP;
12  DBMS_OUTPUT.PUT_LINE('The Factorial of'||n1||' is'||FACT);
13  END;
14 /
Enter value for n: 5
old   6: N:=&N;
new   6: N:=5;
The Factorial of 5 is 120

PL/SQL procedure successfully completed.

SQL> SET VERIFY OFF
SQL> /
Enter value for n: 4
The Factorial of 4 is 24

PL/SQL procedure successfully completed.

SQL> |
```

- ❑ To run the program ‘/’ is used.
- ❑ To display the output, we use “SET SERVEROUT ON”.
- ❑ To eliminate debugging message “SET VERIFY OFF” should be used.

Conclusion: In this lab, we have practiced a PL/SQL program to calculate factorial of given number.

EXPERIMENT-11

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

SOURCE CODE & OUTPUT:

```
SQL> DECLARE
  2  N NUMBER;
  3  N1 NUMBER;
  4  I NUMBER;
  5  TEMP NUMBER;
  6  BEGIN
  7  N:=&N;
  8  N1:=N;
  9  I:=2;
10  TEMP:=1;
11  FOR I IN 2..N/2
12  LOOP
13  IF MOD(N,I)=0
14  THEN
15  TEMP:=0;
16  EXIT;
17  END IF;
18  END LOOP;
19  IF TEMP=1
20  THEN
21  DBMS_OUTPUT.PUT_LINE(N||' is a prime number');
22  ELSE
23  DBMS_OUTPUT.PUT_LINE(N||' is not a prime number');
24  END IF;
25  END;
26 /
Enter value for n: 8
8 is not a prime number

PL/SQL procedure successfully completed.

SQL> /
Enter value for n: 11
11 is a prime number

PL/SQL procedure successfully completed.

SQL> |
```

- ⓧ To run the program '/' is used.
- ⓧ To display the output, we use "SET SERVEROUT ON".
- ⓧ To eliminate debugging message "SET VERIFY OFF" should be used.

Conclusion: In this lab, we have practiced a PL/SQL program for finding a given number is prime or not.

EXPERIMENT-12

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

SOURCE CODE & OUTPUT:

```
SQL> DECLARE
  2  FIRST NUMBER:=0;
  3  SECOND NUMBER:=1;
  4  TEMP NUMBER;
  5  N NUMBER;
  6  N1 NUMBER;
  7  I NUMBER;
  8  BEGIN
  9  N:=&N;
 10  N1:=N;
 11  DBMS_OUTPUT.PUT_LINE('SERIES:');
 12  DBMS_OUTPUT.PUT_LINE(FIRST);
 13  DBMS_OUTPUT.PUT_LINE(SECOND);
 14  FOR I IN 2..N
 15  LOOP
 16  TEMP:=FIRST+SECOND;
 17  FIRST:=SECOND;
 18  SECOND:=TEMP;
 19  DBMS_OUTPUT.PUT_LINE(TEMP);
 20  END LOOP;
 21  END;
 22 /
Enter value for n: 6
SERIES:
0
1
1
2
3
5
8
PL/SQL procedure successfully completed.
```

- ⓧ To run the program ‘/’ is used.
- ⓧ To display the output, we use “SET SERVEROUT ON”.
- ⓧ To eliminate debugging message “SET VERIFY OFF” should be used.

Conclusion: In this lab, we have practiced a PL/SQL program for displaying Fibonacci series up to an integer.

EXPERIMENT-13

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

PL/SQL Procedure:

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.

The procedure contains a header and a body.

EXAMPLE-1:

```
SQL> CREATE TABLE SAILOR(ID NUMBER(10) PRIMARY KEY,NAME VARCHAR2(100));  
Table created.  
  
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> DECLARE  
  2  CNT NUMBER;  
  3  BEGIN  
  4  INSERTUSER(202,'CHINNU');  
  5  SELECT COUNT(*) INTO CNT FROM SAILOR;  
  6  DBMS_OUTPUT.PUT_LINE(CNT||' RECORD IS INSERTED SUCCESSFULLY');  
  7  END;  
  8 /  
  
PL/SQL procedure successfully completed.  
  
SQL> |
```

DROP PROCEDURE:

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

Conclusion: In this lab, we have practiced a PL/SQL program to implement Stored Procedure on table.

EXPERIMENT-14

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

PL/SQL Function:

The PL/SQL Function is very similar to PL/SQL Procedure. 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. Expect this, all other things of PL/SQL procedure are true for PL/SQL function too.

EXAMPLE-1:

```
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(22,44);
  5  DBMS_OUTPUT.PUT_LINE('ADDITION IS: '||N3);
  6  END;
  7 /

PL/SQL procedure successfully completed.

SQL> SET SERVEROUT ON
SQL> /
ADDITION IS: 66

PL/SQL procedure successfully completed.

SQL>
```

```
SQL> DROP FUNCTION ADDER;  
Function dropped.  
SQL> |
```

EXAMPLE-2:

```
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.  
SQL> |
```

EXECUTION PROCEDURE:

```
SQL> DECLARE  
 2  NUM NUMBER;  
 3  FACTORIAL NUMBER;  
 4  BEGIN  
 5  NUM:=4;  
 6  FACTORIAL:=FACT(NUM);  
 7  DBMS_OUTPUT.PUT_LINE(' FACTORIAL'||NUM||' IS '|| FACTORIAL);  
 8  END;  
 9 /  
FACTORIAL 4 IS 24  
  
PL/SQL procedure successfully completed.  
SQL>
```

```
SQL> DROP FUNCTION FACT;  
Function dropped.  
SQL> |
```

EXPERIMENT-15

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

Trigger:

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.

A database manipulation (DML) statement (DELETE, INSERT, UPDATE).

A database definition (DDL) statement (CREATE, ALTER, DROP).

A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, SHUTDOWN).

CREATING A TABLE:

```
SQL> CREATE TABLE INSTRUCTOR(
  2  ID NUMBER PRIMARY KEY,
  3  NAME VARCHAR2(50) NOT NULL,
  4  DEPT_NAME VARCHAR2(20) NOT NULL,
  5  SALARY NUMBER(10,2) CHECK(SALARY>45000)
  6  );
```

```
Table created.
```

```
SQL> |
```

INSERTING VALUES INTO THE TABLE:

```
SQL> INSERT INTO INSTRUCTOR VALUES(41,'ALLURI','CSE',55000);
1 row created.
```

```
SQL> INSERT INTO INSTRUCTOR VALUES(43,'RAMA','CSM',50000);
1 row created.

SQL> INSERT INTO INSTRUCTOR VALUES(44,'RAJU','CSE',60000);
1 row created.

SQL> SELECT * FROM INSTRUCTOR;
      ID NAME
DEPT_NAME          SALARY
-----  -----
        41 ALLURI      55000
CSE
        43 RAMA       50000
CSM
        44 RAJU       60000
CSE
```

AN EXAMPLE TO CREATE TRIGGER:

```
SQL> CREATE OR REPLACE TRIGGER display_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.

```
SQL> |
```

A PL/SQL Procedure to execute a trigger:

```
SQL> DECLARE
  2  tot_rows NUMBER;
  3  BEGIN
  4    UPDATE instructor
  5    SET SALARY=SALARY*1.5;
  6    IF sql%notfound THEN
  7      DBMS_OUTPUT.PUT_LINE(' NO INSTRUCTORS UPDATED');
  8    ELSIF sql%found THEN
  9      tot_rows:=sql%rowcount;
 10    DBMS_OUTPUT.PUT_LINE(tot_rows||' INSTRUCTORS UPDATED');
 11  END IF;
 12  END;
 13 /
OLD SALARY: 55000
NEW SALARY: 82500
SALARY DIFFERENCE: 27500
OLD SALARY: 50000
NEW SALARY: 75000
SALARY DIFFERENCE: 25000
OLD SALARY: 60000
NEW SALARY: 90000
SALARY DIFFERENCE: 30000
3 INSTRUCTORS UPDATED

PL/SQL procedure successfully completed.

SQL> |
```

Conclusion: In this lab, we have practiced a PL/SQL program to implement Trigger on table.

EXPERIMENT-16

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

CREATING A TABLE:

```
SQL> CREATE TABLE people(
 2 ID NUMBER PRIMARY KEY,
 3 NAME VARCHAR2(20) NOT NULL,
 4 AGE NUMBER(5) NOT NULL,
 5 SALARY NUMBER(10,2) NOT NULL
 6 );
```

```
Table created.
```

```
SQL> |
```

INSERTING VALUES INTO TABLE:

```
SQL> INSERT INTO people VALUES(61,'SREE',24,60000);
1 row created.

SQL> INSERT INTO people VALUES(62,'LAKSHMI',35,66000);
1 row created.

SQL> INSERT INTO people VALUES(63,'DEEPU',28,78000);
1 row created.

SQL> INSERT INTO people VALUES(64,'YUVAN',30,55000);
1 row created.
```

```
SQL> SELECT* FROM people;
```

ID	NAME	AGE	SALARY
61	SREE	24	60000
62	LAKSHMI	35	66000
63	DEEPU	28	78000
64	YUVAN	30	55000

```
SQL> |
```

CREATE UPDATE PROCEDURE:**CREATE PROCEDURE:**

```
SQL> DECLARE
  2  total_rows NUMBER(2);
  3  BEGIN
  4  UPDATE people
  5  SET SALARY=SALARY+6000;
  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 /
4 CUSTOMERS UPDATED

PL/SQL procedure successfully completed.

SQL> |
```

PL/SQL Program using Explicit Cursors:

```
SQL> DECLARE
  2  p_id people.id%type;
  3  p_name people.name%type;
  4  p_age people.age%type;
  5  CURSOR p_people IS
  6  SELECT id,name,age FROM people;
  7  BEGIN
  8  OPEN p_people;
  9  LOOP
 10  FETCH p_people INTO p_id,p_name,p_age;
 11  EXIT WHEN p_people%notfound;
 12  DBMS_OUTPUT.PUT_LINE(p_id||' '||p_name||' '||p_age);
 13  END LOOP;
 14  CLOSE p_people;
 15  END;
 16 /
61 SREE 24
62 LAKSHMI 35
63 DEEPU 28
64 YUVAN 30

PL/SQL procedure successfully completed.

SQL> |
```

Conclusion: In this lab, we have practiced a PL/SQL program to implement Cursor on table.