KKR & KSR INSTITUTE OF TECHNOLOGY & SCIENCES

(Approved by AICTE, New Delhi & Affiliated to JNTUK, Kakinada, Accredited by NAAC with "A" Grade & Accredited by NBA)

VINJANAMPADU, GUNTUR-500017

Department of Computer Science and Engineering

[II B.Tech IISemester]

LAB MANUAL

FOR

DATABASE MANAGEMENT SYSTEMS

Exercise 1:

Creation, altering and dropping of tables and inserting rows into a table(use constraints while creating tables) examples using SELECT command.

Objective: To understand the basic commands of SQL

Background Theory:

Types of SQL Commands:

1) Data Definition Language (DDL).
2) Data Manipulation Language (DML).
3) Data Control Language (DCL).

Data Definition Language (DDL):

1) It is the subset of the SQL. It is used to support the creation, deletion and altering or modification of definitions for tables and views. In DDL Integrity Constraints (IC) are defined on tables at the time of table creation and after the table is creation.

Data Manipulation Language (DML):

1) DML commands are used to insert, update and delete data from existing tables. It can perform various manipulation operations on the data.

Data Control language(DCL):

1) These commands are also called as TCL commands. These commands are used to Connect and Grant permissions for user and their accounts.

2) Also these DCL or TCL commands are used for implementing of COMMIT, ROLLBACK and SAVEPOINT operations.

Integrity constraints:

In SQL when creating tables we can use several Integrity constraints. All these are used to acquire the data consistency and data integrity.

Some of the Integrity constraints are:

1) Distinct.
2) Not null.
3) Unique.
4) Primary key.
5) Foreign key.

Doman-Types (Data-types) in SQL:

The SQL is having various data types. Those are used to construct the tables with specific domains.

The SQL data types are:

1) **char(size):** It is fixed length character string with specific size.

2) **Varchar2(size):** It is variable length character string with specific size.
3) **Int:** It is an integer data type. We can also write it as "integer".
4) **Small-int:** It is used to specify the small integer.
5) **numeric(p,d):** It is fixed number of digits number with used defined precision (decimal part).
6) **real:** They are floating-point and double-precision floating point numbers.
7) **float(x):**It is a float-point number, with user specified precision (decimal point) of atleast "n" digits.
8) **date:**Date is a data type which is used in the calendar format. Contain year, month, day format.
9) **time:**Time data type is the time of day, in hours, minutes and seconds.

Creation of Table:

Syntax-1:

CREATE TABLE <TABLENAME>
(columnname datatype(size), columnname datatype(size));

Creating a table from a existing table-
Syntax-2:

CREATE TABLE <TABLENAME>[(columnname, columnname, ……)]
AS SELECT columnname, columnname……..**FROM** tablename;

Insertion of data into tables-
Syntax-1

INSERT INTO tablename[(columnname, columnname, ……)]
VALUES (expression, expression,……..);

Inserting data into a table from another table:
Syntax-2

INSERT INTO <tablename>
SELECT columnname, columnname, …….
FROM <tablename>
[**WHERE** <condition>];

Retrieving of all columns from the tables-
Syntax-

SELECT * FROM <tablename>;

The retrieving of specific columns from a table-
Syntax-

SELECT columnname, columnname, ….**FROM** <tablename>;

Elimination of duplicates from the select statement-
Syntax-

SELECT DISTINCT columnname, columnname
FROM <tablename>;

Selecting a specific data set from table data-
Syntax-

SELECT columnname, columnname
FROM <tablename>
WHERE <search-condition>;

Examples:

Creation of Department table is as follows:

```
SQL> CREATE TABLE dept( deptno number(2) primary key,
                        dname varchar2(10) not null,
                        loc varchar2(8));
Table created.
SQL> desc dept
 Name                                      Null?    Type
 ----------------------------------------- --------
 DEPTNO                                    NOT NULL NUMBER(2)
 DNAME                                     NOT NULL VARCHAR2(10)
 LOC                                                VARCHAR2(8)

SQL> insert into dept values(10,'accounting','newyork');
1 row created.
SQL> insert into dept values(20,'research','dallas');
1 row created.
SQL> insert into dept values(30,'sales','chicago');
1 row created.
SQL> insert into dept values(40,'operations','goston');
1 row created.

SQL> select *from dept;
   DEPTNO DNAME      LOC
---------- ---------- --------
         10 accounting newyork
         20 research   dallas
         13 sales      chicago
         40 operations goston
```

Creation of Employee table is as follows:

```
SQL> CREATE TABLE emp (empno number(5) primary key,
                       ename varchar2(10) not null,
                       job varchar2(10),
                       mgr number(4),
                       hiredate date,
                       sal number(7,2) check(sal>=500 and sal<=10000),
                      comm number(7,2),
                      deptno number(2),
                      foreign key(deptno) references dept);

Table created.
SQL> insert into emp(empno,ename,job,mgr,hiredate,sal,deptno)
                values(7369,'smith','clerk',7902,'17-dec-80',800,20);
1 row created.

SQL> insert into emp (empno,ename,job,mgr,hiredate,sal,comm,deptno)
                  values(7499,'allen','salesman',7698,'20-feb-81',1600,300,30);
1 row created.

SQL> insert into emp (empno,ename,job,mgr,hiredate,sal,comm,deptno)
                 values(7521,'ward','salesman',7698,'22-feb-81',1250,500,30);
1 row created.
```

SQL> insert into emp (empno,ename,job,mgr,hiredate,sal,deptno)
values(7566,'jones','manager',7839,'2-apr-81',2975,20);
1 row created.

SQL> insert into emp (empno,ename,job,mgr,hiredate,sal,comm,deptno)
values(7654,'martin','salesman',7698,'28-sep-81',1250,1400,30);
1 row created.

SQL> insert into emp (empno,ename,job,mgr,hiredate,sal,deptno)
values(7698,'blake','manager',7839,'1-may-81',2850,30);
1 row created.

SQL> insert into emp (empno,ename,job,mgr,hiredate,sal,deptno)
values(7782,'clark','manager',7839,'9-jun-81',2450,10);
1 row created.

SQL> insert into emp (empno,ename,job,mgr,hiredate,sal,deptno)
values(7788,'scott','analyst',7566,'19-apr-87',3000,20);
1 row created.

SQL> insert into emp(empno,ename,job,hiredate,sal,deptno)
values(7839,'king','president','17-nov-81',5000,10);
1 row created.

SQL> insert into emp (empno,ename,job,mgr,hiredate,sal,comm,deptno)
values(7844,'turner','salesman',7698,'8-sep-81',1000,0,30);
1 row created.

SQL> insert into emp(empno,ename,job,mgr,hiredate,sal,deptno)
values(7876,'adems','clerk',7788,'23-may-87',1100,20);
1 row created.

SQL> insert into emp(empno,ename,job,mgr,hiredate,sal,deptno)
values(7900,'james','clerk',7698,'3-dec-81',950,30);
1 row created.

SQL> insert into emp(empno,ename,job,mgr,hiredate,sal,deptno)
values(7902,'ford','analyst',7566,'3-dec-81',3000,20);
1 row created.

SQL> insert into emp(empno,ename,job,mgr,hiredate,sal,deptno)
values(7934,'miller','clerk',7782,'23-jan-82',1300,10);
1 row created.

SQL> select * from emp;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	smith	clerk	7902	17-DEC-80	800		20
7499	allen	salesman	7698	20-FEB-81	1600	300	30
7521	ward	salesman	7698	22-FEB-81	1250	500	30
7566	jones	manager	7839	02-APR-81	2975		20
7654	martin	salesman	7698	28-SEP-81	1250	1400	30
7698	blake	manager	7839	01-MAY-81	2850		30
7782	clark	manager	7839	09-JUN-81	2450		10

7788	scott	analyst	7566	19-APR-87	3000	20
7839	king	president		17-NOV-81	5000	10
7844	turner	salesman	7698	08-SEP-81	10000	30
7876	adems	clerk	7788	23-MAY-87	1100	20
7900	james	clerk	7698	03-DEC-81	950	30
7902	ford	analyst	7566	03-DEC-81	3000	20
7934	miller	clerk	7782	23-JAN-82	1300	10

14 rows selected.

Dropping a table:

Syntax: DROP TABLE <tablename>;

Example: DROP TABLE emp;

ALTER TABLE Command:

This Alter Table command can be used to add, delete, modify and rename the columns in existing table. Also this command can be used to add and drop various constraints on existing tables.

Syntax to add column:

ALTER TABLE <table name> **ADD**(column_name specification);

Syntax to drop column:

ALTER TABLE <table name> **DROP COLUMN** columnn_name ;

Syntax to modify column:

ALTER TABLE <table name> **MODIFY**(column_name specification);

Syntax to rename column:

ALTER TABLE <table name> **RENAME COLUMN** old_name TO new_name;

Syntax to add constraints:

ALTER TABLE <table name> **ADD** <constraint type>(column_names);

Experiment 2:

Queries (along with sub Queries) using ANY, ALL, IN ,EXISTS, NOT EXISTS, UNION, INTERSET,

Objective: To understand about different operators in SQL and usage of special operators with sub queries.

Background Theory:

Operators in SQL:

An SQL operator is a special word or character used to perform tasks. These tasks can be anything from complex comparisons, to basic arithmetic operations.

SQL operators are primarily used within the WHERE clause of an SQL statement. This is the part of the statement that is used to filter data by a specific condition or conditions.

Apart from general operators, SQL is having some special operators which include,

Operator	Description
ANY	This will return TRUE if any of the subquery value meet the condition
ALL	This will return TRUE if all of the sub query values meet the condition
IN	This will return TRUE if operand is equal to one of the list of values
EXISTS	This will return TRUE if sub query returns one or more records
BETWEEN-AND	This will return TRUE if operand is within the range of comparisons
LIKE	This will return TRUE if operand matches the pattern
UNION	This will combine the result of two or more SELECT statements
INTERSECT	This will return only common records returned by two or more SELECT statement
MINUS	This will return records from the set which does not exists in another set

Example: Write a query to get employees who are getting the salary 3000.

SQL> select *from emp where sal=3000;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7788	scott	analyst	7566	19-APR-87	3000		20
7902	ford	analyst	7566	03-DEC-81	3000		20

Example: Write a query to get employees who are getting the salary greater than 3000.

SQL> select *from emp where sal>3000;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	king	president		17-NOV-81	5000		10
7844	turner	salesman	7698	08-SEP-81	10000	0	30

Example: Write a query to get employees who are getting the salary less than 3000.

SQL> select *from emp where sal<3000;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	smith	clerk	7902	17-DEC-80	800		20
7499	allen	salesman	7698	20-FEB-81	1600	300	30
7521	ward	salesman	7698	22-FEB-81	1250	500	30
7566	jones	manager	7839	02-APR-81	2975		20
7654	martin	salesman	7698	28-SEP-81	1250	1400	30
7698	blake	manager	7839	01-MAY-81	2850		30
7782	clark	manager	7839	09-JAN-81	2450		10
7876	adems	clerk	7788	23-MAY-87	1100		20
7900	james	clerk	7698	03-DEC-81	950		30
7934	miller	clerk	7782	23-JAN-82	1300		10

Example: Write a query to get employees who are getting the salary less than or equal to 3000.

SQL> select *from emp where sal<=3000;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	smith	clerk	7902	17-DEC-80	800		20
7499	allen	salesman	7698	20-FEB-81	1600	300	30
7521	ward	salesman	7698	22-FEB-81	1250	500	30
7566	jones	manager	7839	02-APR-81	2975		20
7654	martin	salesman	7698	28-SEP-81	1250	1400	30
7698	blake	manager	7839	01-MAY-81	2850		30
7782	clark	manager	7839	09-JAN-81	2450		10
7788	scott	analyst	7566	19-APR-87	3000		20
7876	adems	clerk	7788	23-MAY-87	1100		20
7900	james	clerk	7698	03-DEC-81	950		30
7902	ford	analyst	7566	03-DEC-81	3000		20
7934	miller	clerk	7782	23-JAN-82	1300		10

Example: Write a query to get employees who are getting the salary greater than or equal to 3000.

SQL> select *from emp where sal>=3000;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7788	scott	analyst	7566	19-APR-87	3000		20
7839	king	president		17-NOV-81	5000		10
7844	turner	salesman	7698	08-SEP-81	10000	0	30
7902	ford	analyst	7566	03-DEC-81	3000		20

Example: Write a query to get employees whose job is ‘clerk’ or ‘analyst’.

SQL>select * from emp where job IN ('clerk','analyst');

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	smith	clerk	7902	17-DEC-80	800		20
7788	scott	analyst	7566	19-APR-87	3000		20
7876	adems	clerk	7788	23-MAY-87	1100		20
7900	james	clerk	7698	03-DEC-81	950		30
7902	ford	analyst	7566	03-DEC-81	3000		20
7934	miller	clerk	7782	23-JAN-82	1300		10

Example: Write a query to get employees whose job is not 'clerk' or 'analyst'.

SQL> select *from emp where job NOT IN ('clerk','analyst');

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	allen	salesman	7698	20-FEB-81	1600	300	30
7521	ward	salesman	7698	22-FEB-81	1250	500	30
7566	jones	manager	7839	02-APR-81	2975		20
7654	martin	salesman	7698	28-SEP-81	1250	1400	30
7698	blake	manager	7839	01-MAY-81	2850		30
7782	clark	manager	7839	09-JAN-81	2450		10
7839	king	president		17-NOV-81	5000		10
7844	turner	salesman	7698	08-SEP-81	10000	0	30

Example: Write a query to get empno, ename, job and deptno of managers.

SQL> select empno,ename,job,deptno from emp e
where exists(select empno from emp where emp.mgr=e.empno);

EMPNO	ENAME	JOB	DEPTNO
7698	blake	manager	30
7839	king	president	10
7566	jones	manager	20
7788	scott	analyst	20
7782	clark	manager	10
7902	ford	analyst	20

Example: Write a query to get ename, job who are not managers.

SQL> select ename,job from emp e
where not exists(select mgr from emp where mgr=e.empno);

ENAME	JOB
turner	salesman
ward	salesman
martin	salesman
allen	salesman
miller	clerk
smith	clerk
adems	clerk
james	clerk

Example: Write a query to get employee names who are getting any salary of employees working in the department 20.

SQL> select ename from emp where sal =any(select sal from emp where deptno=20);

ENAME
smith
jones
ford
scott
adems

Example: Write a query to get employee names who are getting less salary of any employees working in the department 20.

SQL> select ename from emp where sal <any(select sal from emp where deptno=20);

```
ENAME
----------
smith
james
adems
ward
martin
miller
allen
clark
blake
jones
```

Example: Write a query to get employee names who are getting more salary of any employees working in the department 20.

SQL> select ename from emp where sal >any(select sal from emp where deptno=20);

```
ENAME
----------
turner
king
ford
scott
jones
blake
clark
allen
miller
martin
ward
adems
james
```

Example: Write a query to get employee names who are getting same salary of all employees working in the department 30.

SQL> select ename from emp where sal =all(select sal from emp where deptno=30)

no rows selected

Example: Write a query to get employee names who are getting less salary of all employees working in the department 30.

SQL> select ename from emp where sal <all(select sal from emp where deptno=30)

```
ENAME
----------
smith
```

Example: Write a query to get employee names who are getting more salary of all employees working in the department 20.

SQL> select ename from emp where sal >all(select sal from emp where deptno=20)

```
ENAME
----------
king
turner
```

Example: Write a query to get employee names who are working in the departments located in 'dallas' and 'newyork'

```
SQL> select e.ename from emp e,dept d
       where e.deptno=d.deptno and d.loc='dallas'
     UNION
       select e.ename from emp e,dept d
       where e.deptno=d.deptno and d.loc='newyork';
ENAME
---------
clark
king
miller
```

Example: Write a query to get department numbers that are common for both emp and dept tables.

SQL> select e.deptno from emp e INTERSECT select d.deptno from dept d;

```
DEPTNO
----------
10
20
30
```

Example: Write a query to get department numbers that are not having employees.

SQL> select deptno from dept MINUS select deptno from emp;

```
   DEPTNO
---------------
   40
```

Example: Write a query to get employee number, name and salary who does not have commission.

SQL> select ename,empno,sal from emp where comm is null;

```
ENAME   EMPNO      SAL
---------- ---------- ----------
smith         7369       800
jones         7566      2975
blake         7698      2850
clark         7782     2450
scott         7788      3000
king         7839      5000
adems         7876     1100
```

```
james          7900       950
ford           7902      3000
miller         7934      1300
```

Example: Write a query to get employee number, name and salary who have commission.

SQL> select ename,empno,sal from emp where comm is not null;

```
ENAME    EMPNO       SAL
---------- ---------- ----------
allen          7499      1600
ward           7521      1250
martin         7654      1250
turner         7844     10000
```

Example: Write a query to get employee number, name and salary whose salary between 1000 and 3000.

SQL> select ename,empno,sal from emp where sal between 1000 and 3000;

```
ENAME         EMPNO       SAL
---------- -   --------- --          --------
allen          7499        1600
ward           7521       1250
jones          7566        2975
martin         7654       1250
blake          7698        2850
clark          7782        2450
scott          7788        3000
adems          7876       1100
ford           7902          3000
miller         7934        1300
```

Example: Write a query to get employee number, name and salary whose salary not in between 1000 and 3000.

SQL> select ename,empno,sal from emp where sal not between 1000 and 3000;

```
ENAME         EMPNO       SAL
----------          ----------     ----------
smith               7369         800
king               7839         5000
turner              7844         10000
james               7900         950
```

Example: Write a query to get employee names and salary whose name starts with letter ‘s’.

SQL> select ename, sal from emp where ename like's%';

```
ENAME          SAL
---------- ----------
smith           800
scott          3000
```

Example: Write a query to get employee names and salary whose name not start with letter ‘s’.

SQL> select ename, sal from emp where ename not like's%';

ENAME	SAL
allen	1600
ward	1250
jones	2975
martin	1250
blake	2850
clark	2450
king	5000
turner	10000
adems	1100
james	950
ford	3000
miller	1300

Example: Write a query to get employee names and salary whose name ends with letter ‘s’.

SQL> select ename, sal from emp where ename like '%s';

ENAME	SAL
jones	2975
adems	1100
james	950

Example: Write a query to get employee names and salary whose name starts with ‘smit’ and end with any single character.

SQL> select ename, sal from emp where ename like 'smit_';

ENAME	SAL
smith	800

Exercise 3:
Queries using Aggregate Functions (COUNT, SUM, AVG, MAX and MIN), GROUP BY, HAVING and Creation and dropping of Views.

Objective: To understand about Aggregate Functions and handling of Views.

Background Theory:
These aggregate functions are also called as Group functions. Actually these are mathematical functions that can operate on sets of rows to give one result per set. The types of these functions are

Function	Description
COUNT	It is used to count the number of rows in a set
SUM	It is used to calculate the sum of values in a specified column
AVG	It is used to calculate the average of values in a specified column
MAX	It is used to find the maximum value in a specified column
MIN	It is used to find the minimum value in a specified column

GROUP BY Clause:
This clause can be used to arrange identical data into groups. Usually this clause is used along the aggregate functions.

HAVING Clause:
This clause can be used to specify a condition with aggregate functions along with GROUP BY clause. The WHERE clause can't be used with aggregate functions.

Example: Write a query to count the records for employee table.
SQL> select count(*) from emp;
```
  COUNT(*)
  ----------
    14
```

Example: Write a query to count the no of distinct jobs of employees.
SQL> select count(distinct job) from emp;
```
COUNT(DISTINCTJOB)
------------------
5
```

Example: Write a query to find the sum of salaries for employees.
SQL> select sum(sal) from emp;
```
  SUM(SAL)
----------
    37525
```

Example: Write a query to find the maximum salary for the job 'salesman'.
SQL> select max(sal) from emp where job='salesman';
```
  MAX(SAL)
----------
    10000
```

Example: Write a query to the minimum salary of employees.

```
SQL> select min(sal) from emp;
  MIN(SAL)
----------
       800
```

Example: Write a query to find the average salaray for employee of department 20.

```
SQL> select avg(sal), count(*) from emp where deptno=20;
  AVG(SAL)   COUNT(*)
---------- ----------
      2175          5
```

Example: Write a query to count the no of employee working in each department.

```
SQL> select deptno, count(*) from emp group by deptno;
    DEPTNO   COUNT(*)
---------- ----------
        30          6
        20          5
        10          3
```

Example: Write a query to find the sum of salaries for each department,

```
SQL> select deptno, sum(sal) from emp  emp group by deptno;
    DEPTNO   SUM(SAL)
---------- ----------
        30      17900
        20      10875
        10       8750
```

Example: Write a query to find the no of employees for each job.

```
SQL> select job, count(*) from emp group by job order by count(*) desc;
JOB       COUNT(*)
---------   ----------
salesman          4
clerk             4
manager           3
analyst           2
president         1
```

Example: Write a query to find sum, average, maximum and minimum salaries of each job.

```
SQL> select job,sum(sal),avg(sal),max(sal),min(sal) from emp group by job;

JOB       SUM(SAL)   AVG(SAL)   MAX(SAL)   MIN(SAL)
---------   ----------   ----------   ----------   ----------
salesman      14100    3525         10000        1250
president      5000    5000         5000         5000
clerk          4150    1037.5       1300         800
manager        8275    2758.33      2975         2450
analyst        6000     3000        3000         3000
```

Example: Write a query to find the job wise average salaries of employees who are not managers.

```
SQL> select job, avg(sal) from emp where job!='manager' group by job;
JOB         AVG(SAL)
---------     ----------
salesman        3525
```

president 5000
clerk 1037.5
analyst 3000

Example: Write a query to find sum, average, maximum and minimum salaries of each job for employees working in department 20.

SQL> select job,sum(sal),avg(sal),max(sal),min(sal) from emp where deptno=20 group by job;

JOB	SUM(SAL)	AVG(SAL)	MAX(SAL)	MIN(SAL)
clerk	1900	950	1100	800
manager	2975	2975	2975	2975
analyst	6000	3000	3000	3000

Example: Write a query to find department number and average salary for that department which is having more than 5 employees.

SQL> select deptno, avg(sal) from emp group by deptno having count(*)>5;

DEPTNO	AVG(SAL)
30	2983.33333

Example: Write a query to find job and maximum salary for the job and the maximum salary is more than 3000.

SQL> select job, max(sal) from emp group by job having max(sal)>=3000;

JOB	MAX(SAL)
salesman	10000
president	5000
analyst	3000

Example: Write a query to find sum, average, maximum and minimum salaries of each job and average salary is more than 1000 and display the information in ascending order of sum of salaries.

SQL> select job,sum(sal),avg(sal),max(sal),min(sal) from emp where deptno=20 group by job having avg(sal)>1000 order by sum(sal);

JOB	SUM(SAL)	AVG(SAL)	MAX(SAL)	MIN(SAL)
manager	2975	2975	2975	2975
analyst	6000	3000	3000	3000

Example: Write a query to find employee name who is getting maximum salary.

SQL> select ename from emp where sal=(select max(sal) from emp);

ENAME
---
turner

Example: Write a query to find job for which average salary is the maximum average salary for that job.

SQL> select job,avg(sal) from emp group by job having avg(sal)=(select max(avg(sal)) from emp group by job);

JOB	AVG(SAL)
president	5000

Experiment 4:
Queries using Conversion Functions(to_char, to_number and to_date), String functions(Concatenation, lpad, rpad, ltrim, rtrim, lower, upper, initcap, length, substr and instr), date functions (Sysdate, next_day, add_months, last_day, months_between, least, greatest, trunk, round)

Objective: To understand about Numeric, String, Date functions and Conversion functions.

Background Theory:

Numeric Functions:

a) **ABS(n):** It returns the absolute value of the x;
Syntax: Select ABS(n) from DUAL;

b) **SQRT(n)** : It returns the square root value of n. if it is negative null values is returned.
Syntax: Select SQRT(n) from DUAL;

c) **CEIL(n)** : It returns the smallest integer greater than or equal to n.
Syntax: Select CEIL(n) from DUAL;

d) **FLOOR(n)**: It returns the largest integer less than or equal to n.
Syntax: Select FLOOR(n) from DUAL;

e) **POWER(m,n):** It returns m raised to the power n.
Syntax: Select POWER(m,n) from DUAL;

f) **EXP(n):** It returns exponential value of n
Syntax: Select EXP(n) from DUAL

g) **MOD(m,n):** It returns the remainder of m divided by n
Syntax: Select MOD(m,n) from DUAL;

h) **ROUND(m,n):** It is used to round the n specify number of digits after the decimal.
Syntax: Select ROUND(M,n) from DUAL;

i) **TRUNC(m,n):** It is used to Truncates the n specify number of digits after the decimal.
Syntax: Select ROUND(M,n) from DUAL;

String Functions:

a) **LENGTH('string')** : It is used to return the number of characters in string.
Syntax: Select LENGTH('STRING') FROM DUAL;

b) **LOWER('string') :** It is used to convert the string to lower case letters.
Syntax: Select LOWER('STRING') FROM DUAL;

c) **UPPER('string') :** It converts a string to uppercase.
Syntax: Select UPPER('STRING') FROM DUAL;

d) **INITCAP('string') :** It is used to convert the first letter into Uppercase letter.
Syntax: Select INITCAP('STRING') FROM DUAL;

e) **REPLACE('String', 'Source string', 'Replace string')** : It is used to replace the search string with Replace string
Syntax: Select REPLACE('String', 'Source string', 'Replace string') from DUAL;

f) **SUBSTR('string', m, n)** : It used to display the searching string from m position to n position
Syntax: Select SUBSTR('string', m, n) from DUAL;

g) INSTR('string','char') : It returns the position of the first occurrence in the string.
Syntax: Select INSTR('string','char') from DUAL;

h) LPAD('string', n, 'wildcard character') : It worked on left side of the given string and fill that area with specified characters.
Syntax: Select LPAD('string', n, 'wildcard character') from DUAL;

i) RPAD('string', n, 'wildcard character') : It worked on right side of the given string and fills that area with specified characters.
Syntax: Select RPAD('string', n, 'wildcard character') from DUAL;

j) **LTRIM('string', 'char')** : It is used to trim the character from the string.
Syntax: Select LTRIM('string', 'char') from DUAL;

k) **RTRIM('string', 'char')** : It is used to trim the character from the string.
Syntax: Select RTRIM('string', 'char') from DUAL;

Miscellaneous Functions:

a) **GREATEST(**list of values): It returns greatest value in given list of values.
Syntax: Select GREATEST(v1,v2,v3,….) from dual;

b) **LEAST**(list of values): It returns smallest value in the given list of values.
Syntax: Select LEAST(v1,v2,v3,……) from dual;

Date functions:

a) **SYSDATE**: it returns the system date.
Syntax: Select SYSDATE from DUAL;

b) **NEXT_DAY('date','day name')** : it returns the date of next specified day of the week after the date.
Syntax: Select NEXT_DAY('date', 'day name') from DUAL;

c) **ADD_MONTHS('date' , n)** : it can add months to given 'date'
Syntax: Select ADD_MONTHS('date' , n) from DUAL

d) **MONTHS_BETWEEN('date1' , 'date2')** : it returns the number of months between date 1 and date 2.
Syntax: Select MONTHS_BETWEEN('date1' , 'date2') from DUAL;

Conversion Functions:

These functions are to convert the one data type to another data type.

a) **TO_CHAR('date', format)** : it is used to convert the date into the specified character format.

Syntax:

Select **TO_CHAR**('Date specification', 'DDTH-MMTH-YYTH') from DUAL;
Select **TO_CHAR**('date specifications', 'ddspth-mmspth-yyspth') from dual;

SPECIAL DATE FORMATS :

DEFAULT FORMAT-DD:MON:YY;

DAY:

D - DAY OF WEEK(1 TO 7)
DD - DAY OF THE MONTH(1 TO 31)
DDD - DAY OF THE YEAR(1 TO 365)
DY - NAME OF THE DAY,THREE LETTER ABBRIVITAION
SUN,MON,THU
DAY - NAME OF THE DAY(MONDAY,SUNDA etc)

MONTH:

MM - MONTH OF NUMBER(1 TO 12)
MON - NAME OF THE MONTH WITH THREE LETTERS(JAN,FEB,MAR)
MONTH - FULL NAME OF THE MONTH.

YEAR :

Y - LAST DIGIT OF THE YEAR(2000-0)
YY - TWO DIGITS OF THE YEAR(2000-00)
YYY - THREE DIGITS OF THE YEAR(2000-000)
YYYY – FULL DIGITS OF THE YEAR
Y,YYY - 2,000

b) **TO_DATE('char' , format)** : it is used to convert the character into specified date format;

Syntax: Select **TO_DATE**('date in character' , 'date') from dual;

Examples:

SQL> select CEIL(77.7) from DUAL;

CEIL(77.7)

78

SQL> select FLOOR(69.2) from DUAL;

FLOOR(69.2)

69

SQL> select ABS(-19) from DUAL;

ABS(-19)

19

SQL> select POWER(7,2) from DUAL;

POWER(7,2)

49

SQL> select MOD(79,10) from DUAL;

MOD(79,10)

9

SQL> select SIGN(-8), SIGN(9) from DUAL;

SIGN(-8) **SIGN(9)**

-1 1

SQL> select ROUND(55.438,1) from DUAL;

ROUND(55.438,1)

55.4

SQL> select EXP(4) from DUAL;

EXP(4)

54.59815

SQL> select SQRT(64) from DUAL;

SQRT(64)

8

SQL> select TRUNC(79.128,2) from DUAL;

TRUNC(79.128,2)

79.12

SQL> select **LENGTH**('second cse') "OutPut" from dual;

OutPut

10

SQL> select **LOWER**('CHEC') "result" from dual;

result

chec

SQL> select **UPPER**('chec') "OutPut" from dual;

OutPut

CHEC

SQL> select **INITCAP**('chec') "output" from dual;

OutPut

Chec

SQL> select **REPLACE**('midia and midia', 'mi', 'in') "repalced" from dual;

Replaced

india and india

SQL> select **SUBSTR**('independence', 3, 8) "substring" from dual;

Substrin

dependen

SQL> select **INSTR**('aeroplane', 'p') "result" from dual;

OutPut

5

SQL> select **LPAD**('cat', 5, '*') "padding OutPut" from dual;

OutPut

**cat

SQL> select **RPAD**('doll',9,'%') "OutPut" from dual;

OutPut

doll%%%%%

SQL> select **LTRIM**('INTERNET', 'IN') "Result" from dual

Result

TERNET

SQL> select **RTRIM**('internet', 'r') "Result" from dual;

Result

internet

SQL> select **RTRIM**('internet', 'i') "Result" from dual;

Result

internet

SQL> select **RTRIM**('internet', 't') "Result" from dual;

Result

Interne

SQL> select **SYSDATE** from dual;

```
SYSDATE
---------
13-AUG-15
```

SQL> select hireddate, **ADD_MONTHS** (hireddate,4),ADD_MONTHS(hireddate,-4)
from emp where deptno=10;

```
HIREDDATE    ADD_MONTHS(HIREDDA    ADD_MONTHS(HIREDDA
---------    ------------------    ------------------
09-JUN-81    09-OCT-81             09-FEB-81
17-NOV-81    17-MAR-82             17-JUL-81
23-JAN-82    23-MAY-82             23-SEP-81
```

SQL> select **ROUND**(TO_DATE('12-apr-71'),'MM') "Nearest month"
from dual;

Nearest month

01-APR-71

SQL> select **MONTHS_BETWEEN**('05-jan-98','05-jan-98') "Same Months",

MONTHS_BETWEEN('05-mar- 98','05-jan-98') "Diff Months" from dual;

Same Months Diff Months
--- ---
0 2

SQL> select **SYSDATE, LAST_DAY(SYSDATE)** from dual;

SYSDATE LAST_DAY(SYSDATE)
--- ---
13-AUG-15 31-AUG-15

SQL> select **SYSDATE, NEXT_DAY(SYSDATE,'WEDNESDAY')** from dual;

SYSDATE NEXT_DAY(SYSDATE,'
--- ---
13-AUG-15 19-AUG-15

SQL> select **SYSDATE, TO_CHAR(SYSDATE,'DAY')** from dual;

SYSDATE TO_CHAR(SYSDATE,'DAY')
--- ---
13-AUG-15 THURSDAY

SQL> select **GREATEST**(10,'7',-1) from dual;

GREATEST(10,'7',-1)

10

SQL> select **LEAST**('abcd','ABCD','a','XYZ') "Least" from dual;

Least

ABCD

SQL> select **LEAST**(9,3,56,89,23,1,0,-2,12,34,7,22) as LOWEST from dual;

LOWEST

-2

Example: Write a query to convert hireddate of employees as DD/MM/YY for department 20.

```
SQL> select ename, TO_CHAR(hireddate,'DD/MM/YY') as hireddate  from emp
        where deptno=20;
ENAME          HIREDDAT
---------      -----------------
smith          17/12/80
jones          02/04/81
scott          19/04/87
adams          23/05/87
ford           03/12/81
```

Example: Write a query to display salary of employees with symbol '$'.

```
SQL> select eno,ename,job, TO_CHAR(sal,'$9999') as salary from emp;

 ENO ENAME      JOB            SALARY
---------- ----------  ----------      ------
7369 smith          clerk          $800
7499 allen          salesman       $1600
7521 ward           salesman       $1250
7566 jones          manager        $2975
7654 martin         salesman       $1250
7698 blake          manager        $2850
7782 clark          manager        $2450
7788 scott          analyst        $3000
7839 king           president      $5000
7844 turner         salesman       $1500
7876 adams          clerk          $1100
7900 james          clerk          $950
7902 ford           analyst        $3000
7984 miller         clerk          $1300
```

Example: Write a query to find the no of employees who joined in the same year.

```
SQL> select TO_CHAR(hireddate,'YY') as YY,count(*)  from emp
            group by TO_CHAR(hireddate,'YY');

YY   COUNT(*)
-- ----------
87          2
81         10
82          1
80          1
```

PL/SQL PROGRAMS

What is PL/SQL?

PL/SQL stands for Procedural Language extension of SQL.

PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.

The PL/SQL Engine:

Oracle uses a PL/SQL engine to processes the PL/SQL statements. A PL/SQL code can be stored in the client system (client-side) or in the database (server-side).

A Simple PL/SQL Block:

Each PL/SQL program consists of SQL and PL/SQL statements which form a PL/SQL block.

A PL/SQL Block consists of three sections:

- **The Declaration section (optional).**
- **The Execution section (mandatory).**
- **The Exception (or Error) Handling section (optional).**

Declaration Section:
The Declaration section of a PL/SQL Block starts with the reserved keyword DECLARE. This section is optional and is used to declare any variables, constants, records and cursors, which are used to manipulate data in the execution section.

Execution Section:
The Execution section of a PL/SQL Block starts with the reserved keyword BEGIN and ends with END. This is a mandatory section and is the section where the program logic is written to perform any task. The programmatic constructs like loops, conditional statement and SQL statements form the part of execution section.

Exception Section:
Exception section starts with **EXCEPTION** keyword. This section is optional which contains statements that are executed when a run-time error occurs. Any exceptions can be handled in this section.

Every statement in the above three sections must end with a semicolon; and PL/SQL blocks can be nested within other PL/SQL blocks. Comments can be used to document code.

This is how a sample PL/SQL Block looks.

```
DECLARE
       declaration statements;-----Optional

BEGIN
       executable statements------Mandatory

EXCEPTIONS
       exception handling statements------Optional

END;
```

Comments:

1) **Single line Comments:** --(double hyphen) is used for single line comments
2) **Multiple line Comments:** /*…*/ symbols are used for multi line comments

PL/SQL Variables

These are placeholders that store the values that can change through the PL/SQL Block.

The General **Syntax** to declare a variable is:

variable_name datatype [NOT NULL := value];

- *variable_name* is the name of the variable.
- *datatype* is a valid PL/SQL datatype.
- NOT NULL is an optional specification on the variable.
- *value* or DEFAULT *value* is also an optional specification, where you can initialize a variable.
- Each variable declaration is a separate statement and must be terminated by a semicolon.

When a variable is specified as NOT NULL, you must initialize the variable when it is declared.

The value of a variable can change in the execution or exception section of the PL/SQL Block.

Anchored Data types:

A variable can be declared as having anchored data type , it means that data type for a variable is determined based on the data type of other object. Its general syntax is given as follows:

Variable_name object%type[NOT NULL]:=initial_value;

Example:

no student.sno%type;

Assigning a value to variable:

There are different ways to assign a value to a given variable, which include

1) **By using Assignment operator**: we can directly assign a value to a variable with '=' as
   Variable_name:= value;

2) **By reading from the keyboard**: we can read a value for a variable from keyboard while the program is in execution with the following syntax
   Variable_name:=&name

3) **By selecting or fetching table data values**: we can assign values to variables by fetching them from a table with following syntax
   Select col1, col2, col3,…. Into var1,var2,var3,…..from <tname> [where <condition>];

Displaying Message:

To display a message or any output of the program on the screen in PL/SQL, the following statement is used.

Syntax: dbms_output.put_line(message);

Note: The environment parameter "serveroutput" must be set to ON to display messages or output of programs on the screen.

Exercise 5:

i) Create a simple PL/SQL program which includes declaration section, executable section and exception handling sections (Example: Student marks can be selected from the table and printed for those who secured first class and an exception can be raised if no records were found).

ii) Insert data into student and use COMMIT, ROLLBACK and SAVEPOINT in PL/SQL block

Objective: To understand about the basic structure of PL/SQL block

Background Work:

```
CREATE TABLE STUDENT(SNO NUMBER(5) PRIMARY KEY,
                     SNAME VARCHAR2(20) NOT NULL,
                     M1  NUMBER(3) ,
                     M2  NUMBER(3) ,
                     M3  NUMBER(3) );
SQL> insert into student values(121,'Kishore',66,74,85);
SQL> insert into student values(122,'Suresh',45,56,67);
SQL> insert into student values(123,'Prasad',75,58,61);
SQL> insert into student values(124,'Krishna',54,60,45);
SQL> insert into student values(125,'Prakash',77,66,88);
SQL> commit;
SQL> select * from student;
```

SNO	SNAME	M1	M2	M3
121	Kishore	66	74	85
122	Suresh	45	56	67
123	Prasad	75	58	61
124	Krishna	54	60	45
125	Prakash	77	66	88

i)
```
declare
   exp exception;
   flag number:=0;

begin
    for item in
      (select sno,sname,m1,m2,m3 from student where m1>=40 and m2>=40 and m3>=40
                                                  and round((m1+m2+m3)/3,0)>=60)
    loop
        flag:=1;
        dbms_output.put_line(item.sno||','||item.sname);
    end loop;
    if flag=0 then
        raise exp;
    end if;
exception
    when exp then
        dbms_output.put_line('No first classes');
end;
```

Output:

```
Statement processed.
SNo SNAME
--- -----
121 Kishore
123 Prasad
125 Prakash
```

SQL> update student set m1=35 where sname='Kishore';
SQL> update student set m2=33 where sname='Prakash';
SQL> update student set m3=39 where sname='Prasad';

After updation the table becomes:

SNO	SNAME	M1	M2	M3
121	Kishore	35	74	85
122	Suresh	45	56	67
123	Prasad	75	58	39
124	Krishna	54	60	45
125	Prakash	77	33	88

After execution of above program for updated table the output becomes

Output:

Statement processed.
No first classes

Exercise 6:

Develop a program that includes the features NESTED IF, CASE and CASE expression. The program can be extended using NULLIF and COALESCE functions.

Objective: To understand about basic control statements in PL/SQL.

Background Theory:

if – then – else Statement:
PL/SQL supports programming language features like conditional statements, iterative statements. These programming constructs are similar to how we use in programming languages like Java and C++. The syntax of conditional statements in PL/SQL programming is

```
if (condition) then
        -- Executes this block if condition is true
else
        -- Executes this block if condition is false
end if;
```

The if statement alone tells us that if a condition is true it will execute a block of statements. We can use the else statement with if statement to execute a block of code when the condition is false.

Nested if statement:
Nested if-then statements mean an if statement inside another if statement. PL/SQL allows us to nest if statements within if-then statements. i.e, we can place an if then statement in another if statement or else part of it.
Syntax-1:-

```
if (condition1) then
        -- Executes when condition1 is true
if (condition2) then
        -- Executes when condition2 is true
end if;
end if;
```

Syntax-2:

```
if condition 1 then
        -- Executes when condition1 is true
elsif condtion2 THEN
        -- Executes when condition2 is true
else
        -- Executes when condition1 and condition2 are false
end if;
```

Example 1: Program to find biggest of two numbers.

```
declare
   a number;
   b number;
begin
  a:=25;
  b:=45;
 if (a>b) then
    dbms_output.put_line('the biggest of a & b is '||a);
 else
    dbms_output.put_line('the biggest of a & b is '||b);
 end if;
end;
```

Output:

```
Statement processed.
the biggest of a & b is 45
```

Example 2: Program to smallest of three numbers.

```
DECLARE
        a NUMBER :=10;
        b NUMBER :=15;
        c NUMBER :=20;
BEGIN
        if (a<b and a<c) then
                dbms_output.put_line('the smallest of a,b and c is '||a);
        elsif (b<c) then
                dbms_output.put_line('the snallest of a,b and c is '||b);
        else
                dbms_output.put_line('the smallest of a,b and c is '||c);
        end if;
END;
```

Output:

```
Statement processed.
the smallest of a,b and c is 10
```

CASE Statement:

Like the **if** statement, the **CASE statement** selects one sequence of statements to execute. However, to select the sequence, the **CASE** statement uses a selector rather than multiple Boolean expressions. A selector is an expression, the value of which is used to select one of several alternatives. The syntax for the case statement in PL/SQL is −

```
CASE selector
        WHEN 'value1' THEN S1;
        WHEN 'value2' THEN S2;
        WHEN 'value3' THEN S3;
        ...
        ELSE Sn;  -- default case
END CASE;
```

Example 1: Program to display the description of grades

```
DECLARE
        grade char(1) := 'A';
BEGIN
        CASE grade
                when 'A' then dbms_output.put_line('Excellent');
                when 'B' then dbms_output.put_line('Very good');
                when 'C' then dbms_output.put_line('Well done');
                when 'D' then dbms_output.put_line('You passed');
                when 'F' then dbms_output.put_line('Better try again');
        else    dbms_output.put_line('No such grade');
        END CASE;
END;
```

Output:
Statement processed.
Excellent

NULLIF Function:

The NULLIF function was introduced in Oracle 9i. It accepts two parameters and returns null if both parameters are equal. If they are not equal, the first parameter value is returned.

Example 1: select **NULLIF(10,10)** from dual;
Output:

NULLIF(10,10)
-

Example 2: select **NULLIF(10,20)** from dual;
Output:

NULLIF(10,20)
10

COALESCE Function:

The COALESCE function was introduced in Oracle 9i. It accepts two or more parameters and returns the first non-null value in a list. If all parameters contain null values, it returns null.

Example 1: select COALESCE(NULL,'A','B') "result" from dual;

Output:

RESULT
A

Example 2: select COALESCE(NULL, NULL, NULL, NULL) "result" from dual;

Output:

RESULT
-

Exercise 7:

Program development using WHILE LOOPS, numeric FOR LOOPS, nested loops using ERROR Handling, BUILT-IN Exceptions, USER defined Exceptions, RAISE APPLICATION ERROR.

Objective: To understand about basic Loop statements and Exception Handling in PL/SQL.

Background Theory:
An Loop Control Statements are used when we want to repeat the execution of one or more statements for specified number of times.

There are three types of loops in PL/SQL:

1) **Simple Loop.**
2) **While Loop.**
3) **For Loop**

1) Simple Loop: In this loop structure, sequence of statements is enclosed between the LOOP and the END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop. It requires EXIT statement within the body to terminate the loop.

Syntax:

```
LOOP
        Execute Commands/Statements
END LOOP;
```

Example: **Write a program to display numbers from 1 to 5 along with their square values.**

```
Declare
    n number(3):=1;
begin
    LOOP
        Exit when n>=6;
        Dbms_output.put_line('the square of ||n|| ' is '||n*n);
        n:=n+1;
    END LOOP;
End;
```

Output:

```
Statement processed.
the square of 1 is 1
the square of 2 is 4
the square of 3 is 9
the square of 4 is 16
the square of 5 is 25
```

2) **WHILE LOOP**: Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

Syntax:

```
WHILE <condition>
LOOP
        Execute Commands/Statements
END LOOP;
```

Example: **To check whether the given number is palindrome or not.**

```
Declare
    n  number(5);
    m number;
    rev number:=0;
    r number;
begin
    n=567;
    m:=n;
    WHILE n>=0
    LOOP
        r:=mod(n,10);
        rev:=rev*10+r;
        n=trunc(n,10);
    END LOOP;
    if m=rev then
        dbms_out.put_line('The given number is palindrome');
    else
        dbms_output.put_line('The given number is not palindrome');
  Exception
    When value_error then
        dbms_output.put_line('The value is too large to assign ');
    When zero_divide then
        dbms_output.put_line('Division with zero ');
  End;
```

Output:
Statement processed.
The given number is not palindrome

3) **FOR LOOP**: Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.

Syntax:

```
    FOR variable IN [Reverser] Start…end
    LOOP
        Execute Commands/Statement;
    END LOOP;
```

Important steps to follow when executing a for loop:

1) The counter variable is implicitly declared in the declaration section, so it's not necessary to declare it explicitly.
2) The counter variable is incremented by 1 and does not need to be incremented explicitly.
3) EXIT WHEN statement and EXIT statements can be used in FOR loops but it's not done oftenly.

Example: Program to check whether the given value is prime of not.

```
Declare
    n    number(5);
    nof    number:=0;
    exp   exception;
begin
    n:=23;
    if n<0 then
       raise exp;
     end if;
```

```
        FOR i in 1..n
        LOOP
                if mod(n,i)=0 then
                    nof:=nof+1;
                end if;
        END LOOP;
        if nof=2 then
                dbms_output.put_line('The given number is prime');
        else
                dbms_output.put_line('The given number is not prime');
        end if;
    Exception
        When zero_divide then
                dbms_output.put_line('The value is too large to assign');
        When exp then
                dbms_output.put_line('The given value is negative');
    End;
```

Output:

```
Statement processed.
The given number is prime
```

Nested Loops: PL/SQL allows using one loop inside another loop. Such loops are called as nested loops. We can use any loop in any other loop.

Example: Program to print Prime numbers upto 50.

```
Declare
        nof   number;
        k   number;
begin
        dbms_output.put_line('The Prime numbers are given as  ');
        FOR n in 1..50 LOOP
                nof:=0;
                k:=floor(n/2);
                FOR i in 2..k LOOP
                        if (mod(n,i)=0) then
                                nof:=1;
                        end if;
                END LOOP;
                if(nof=0) then
                        dbms_output.put_line(n);
                End if;
        END LOOP;
End;
```

Output:

```
Statement processed.
The Prime numbers are given as
1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

Types of Exception

There are two types of Exceptions in Pl/SQL.

1. Predefined Exceptions
2. User-defined Exception

Predefined Exceptions:
Oracle has predefined some common exceptions. These exceptions have a unique exception name and error number. These exceptions are already defined in the 'STANDARD' package in Oracle. In code, we can directly use these predefined exception name to handle them.

User-defined Exception
In Oracle, other than the above-predefined exceptions, the programmer can create their own exception and handle them. They can be created at a subprogram level in the declaration part. These exceptions are visible only in that subprogram. The exception that is defined in the package specification is public exception, and it is visible wherever the package is accessible.

```
Syntax: At Subprogram level
       DECLARE
              <exception_name> EXCEPTION;
       BEGIN
       <Execution block>
       EXCEPTION
              WHEN <exception_name> THEN
                     <Handler>
       END;
```

```
Syntax: At Package specification level
       CREATE PACKAGE <package_name>
       IS
       <exception_name> EXCEPTION;
       .
       .
       END <package_name>;
```

All the predefined exceptions are raised implicitly whenever the error occurs. But the user-defined exceptions needs to be raised explicitly. This can be achieved using the keyword 'RAISE'.

Syntax: RAISE <exception name>;

RAISE_APPLICATION_ERROR:

It is a built-in procedure in oracle which is used to display the user-defined error messages along with the error number whose range is in between -20000 and -20999.

Whenever a message is displayed using RAISE_APPLICATION_ERROR, all previous transactions which are not committed within the PL/SQL Block are rolled back automatically (i.e. change due to INSERT, UPDATE, or DELETE statements). RAISE_APPLICATION_ERROR raises an exception but does not handle it.

RAISE_APPLICATION_ERROR is used for the following reasons,

a) to create a unique id for an user-defined exception.
b) to make the user-defined exception look like an Oracle error.

Syntax:

```
raise_application_error(
      error_number, message[, {TRUE | FALSE}]);
```

The final parameter passed to the procedure is a Boolean(true/false) that tells the procedure to add this error to the error stack or replace all errors in the stack with this error. Passing the value of 'True' adds the error to the current stack, while the default is 'False'.

Steps to be folowed to use RAISE_APPLICATION_ERROR procedure:

1. Declare a user-defined exception in the declaration section.
2. Raise the user-defined exception based on a specific business rule in the execution section.
3. Finally, catch the exception and link the exception to a user-defined error number in RAISE_APPLICATION_ERROR

Example: Raise an error when balance of bank account is too low.

```
DECLARE
        bal NUMBER;
        low_bal exception;
        pragma exception_init(low_bal, -20201);
BEGIN
        SELECT balance INTO bal FROM CUSTOMER WHERE empid=125;
        IF bal < 500 THEN
              /* Issue your own error code( ORA-20201) with your own error message.*/
        RAISE_APPLICATION_ERROR(-20201, 'The Balance is too low');
        END IF
EXCEPTION
        WHEN low_bal THEN
                Dbms_output.put_line(SQLCODE);
                Dbms_output.put_line(sqlerrm);
END;
```

Output:

```
-20201
ORA-20201: The Balance is too low
```

Exercise 8:

Program development using creation of procedure, passing parameters IN and OUT of PROCEDURES.

Objective: To understand about stored procedure concept with different types of parameters.

Background Theory:

PROCEDURE: It is a stored subprogram invoked by the user. This procedure takes both input and output parameters and does't return any value. This is mainly used to perform an action.

A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure.

The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block.

This PROCEDURE is created with **CREATE OR REPLACE PROCEDURE** statement and deleted with **DROP PROCEDURE** statement.

Syntax:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
  < procedure_body >
END procedure_name;
```

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Based on their purpose, parameters are classified as

1. IN Parameter
2. OUT Parameter
3. IN OUT Parameter

IN Parameter:

- This parameter is used for giving input to the subprograms.
- It is a read-only variable inside the subprograms. Their values cannot be changed inside the subprogram.
- In the calling statement, these parameters can be a variable or a literal value or an expression, for example, it could be the arithmetic expression like '5*8' or 'a/b' where 'a' and 'b' are variables.
- By default, the parameters are of IN type.

OUT Parameter:

- This parameter is used for getting output from the subprograms.
- It is a read-write variable inside the subprograms. Their values can be changed inside the subprograms.
- In the calling statement, these parameters should always be a variable to hold the value from the current subprograms.

IN OUT Parameter:

- This parameter is used for both giving input and for getting output from the subprograms.
- It is a read-write variable inside the subprograms. Their values can be changed inside the subprograms.
- In the calling statement, these parameters should always be a variable to hold the value from the subprograms.

These parameter type should be mentioned at the time of creating the subprograms.

Executing a Standalone Procedure:

A standalone procedure can be called in two ways −

- Using the **EXECUTE** keyword
- Calling the name of the procedure from a PL/SQL block

Deleting a Standalone Procedure:

A standalone procedure is deleted with the **DROP PROCEDURE** statement. Syntax for deleting a procedure is −

SQL>DROP PROCEDURE procedure-name;

Example 1: Create a Procedure to find minimum of two numbers.

```
CREATE OR REPLACE PROCEDURE findMin(x IN number, y IN number, z OUT    number)
IS
BEGIN
   IF x < y THEN
      z:= x;
   ELSE
      z:= y;
   END IF;
END;
```

```
DECLARE
   a number;
   b number;
   c number;

BEGIN
   a:= 23;
   b:= 45;
   findMin(a, b, c);
   dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
```

Output:

```
Statement processed.
Minimum of (23, 45) : 23
```

Example 2: It is also possible to create a procedure in DECLARE section as follows,

```
DECLARE
  procedure greetings as
  begin
    dbms_output.put_line('Hello World');
  end;
BEGIN
  greetings;
END;
```

Output:

```
Statement processed.
Hello World.
```

Exercise 9:

Program development using creation of stored functions, invoke functions in SQL statements and write complex functions.

Objective: To understand about how to create and access the stored functions in PL/SQL

Background Theory:

FUNCTION: It is a stored subprogram invoked by the user. This Function takes both input and output parameters and return a single value. This Function mainly used to compute and return a value.
So, a function is a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value

This FUNCTION is created with **CREATE OR REPLACE FUNCTION** statement and deleted with **DROP FUNCTION** statement.

Syntax:

CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name **[IN | OUT | IN OUT]** type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
< function_body >
END [function_name];

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Example 1: Create a Function that should compute and return a maximum of two values.

```
CREATE OR REPLACE FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
   z number;
BEGIN
  IF x > y THEN
    z:= x;
  ELSE
    Z:= y;
  END IF;
  RETURN z;
END;
```

```
DECLARE
   a number;
   b number;
   c number;
BEGIN
   a:= 23;
   b:= 45;
   c := findMax(a, b);
   dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
```

Output:

```
Statement processed.
Maximum of (23, 45) : 45
```

Exercise 10:

Develop programs using features parameters in a CURSOR, FOR UPDATE CURSOR, WHERE CURRENT of clause and CURSOR variables

Objective: To understand about how to create and use cursors in the PL/SQL programs

Background Theory:

What are Cursors?
A cursor is a temporary work area created in the system memory when a SQL statement is executed. A cursor contains information on a select statement and the rows of data accessed by it. This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the *active* set.

There are two types of cursors in PL/SQL:

Implicit cursors:
These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed. They are also created when a SELECT statement that returns just one row is executed.

Explicit cursors:
They must be created when you are executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When you fetch a row the current row position moves to next row.

Both implicit and explicit cursors have the same functionality, but they differ in the way they are accessed.

The General Syntax for creating a cursor is as given below:

CURSOR cursor_name IS select_statement;

- *cursor_name – A suitable name for the cursor.*
- *select_statement – A select query which returns multiple rows.*

There are four steps in using an Explicit Cursor.

- DECLARE the cursor in the declaration section.
- OPEN the cursor in the Execution Section.
- FETCH the data from cursor into PL/SQL variables or records in the Execution Section.
- CLOSE the cursor in the Execution Section before you end the PL/SQL Block.

1**) Declaring a Cursor in the Declaration Section:**

```
DECLARE
    CURSOR emp_cur IS
        SELECT *    FROM emp
        WHERE salary > 5000;
```

In the above example we are creating a cursor ‘emp_cur’ on a query which returns the records of all the employees with salary greater than 5000. Here ‘emp’ in the table which contains records of all the employees.

2) **Accessing the records in the cursor:**

Once the cursor is created in the declaration section we can access the cursor in the execution section of the PL/SQL program.

General Syntax to open a cursor is: **OPEN cursor_name;**

General Syntax to **fetch records** from a cursor is:

FETCH cursor_name INTO record_name;

OR

FETCH cursor_name INTO variable_list;

General Syntax to **close a cursor is**: **CLOSE cursor_name;**

WHERE CURRENT OF & FOR UPDATE

The **WHERE CURRENT OF** clause is used in some UPDATE and DELETE statements.

The **WHERE CURRENT OF** clause in an UPDATE or DELETE statement states that the most recent row fetched from the table should be updated or deleted. We must declare the cursor with the **FOR UPDATE** clause to use this feature.

Inside a cursor loop, WHERE CURRENT OF allows the current row to be directly updated.

When the session opens a cursor with the **FOR UPDATE** clause, all rows in the return set will hold row-level exclusive locks. Other sessions can only query the rows, but they cannot update, delete, or select with **FOR UPDATE**.

Oracle provides the **FOR UPDATE** clause in SQL syntax to allow the developer to lock a set of Oracle rows for the duration of a transaction.

The syntax of using the **WHERE CURRENT OF** clause in UPDATE and DELETE statements follows:

WHERE [CURRENT OF cursor_name | search_condition**]**

Example 1: The following example opens a cursor for employees and updates the commission, if there is no commission then assigned based on the salary level.

```
DECLARE
    CURSOR C1 IS select empno, ename, salary from emp
        WHERE comm IS NULL FOR UPDATE OF comm;
    Var_comm  number(10,2);
BEGIN
    FOR R1 IN C1 LOOP
        if R1.salary<=5000 then
            Var_comm := R1.salary*0.25;
        elsif R1.salaray<=10000 then
            Var_comm:=R1.salaray*0.20;
        elsif R1.salary<=30000 then
            Var_comm:=R1.salary*0.15;
```

```
            else
                Var_comm:=R1.salary*0.12;
            end if;

            UPDATE emp SET comm=var_comm
                WHERE CURRENT OF C1;
    END LOOP;
END;
```

Example 2: The following example opens a cursor for students and updates the result column with concern result.

```
DECLARE
    CURSOR C2 is select * from student;
    r student%rowtype;
    av number(3,0);
    res varchar2(10);
BEGIN
    OPEN C2;
    LOOP
        FETCH C2 INTO r;
        EXIT WHEN C2%NOTFOUND;
        av:=round((r.m1 + r.m2 + r.m3)/3,0);
        if(r.m1>=40 and r.m2>=40 and r.m3>=40) then
          if av>=60 then
              res:='First';
          elsif av>=50 then
              res:='Second';
          elsif av>=40 then
              res:='Third';
        else
              res:='Fail';
        end if;

        UPDATE SET result:=res WHERE sno=r.sno;
    END LOOP;
    CLOSE C2;
END;
```

Exercise 11:

Develop Programs using BEFORE and AFTER Triggers, Row and Statement Triggers and INSTEAD OF Triggers.

Objective: To understand about how Triggers are automatically fired in the events

Background Theory:

Trigger is also a stored procedure that initiates an action when an event(insert/delete/update) occurs. They are stored and managed by **DBMS.** The **DBMS** automatically fires the **trigger** as a result of a data modification to the associated table. Its syntax is

Syntax of Triggers

The Syntax for creating a trigger is:

```
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)
BEGIN
        --- sql statements
END;
```

CREATE [OR REPLACE] TRIGGER trigger_name -:

This clause creates a trigger with the given name or overwrites an existing trigger with the same name.

{BEFORE | AFTER | INSTEAD OF } :

This clause indicates at what time should the trigger get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. Before and after cannot be used to create a trigger on a view.

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

This clause determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.

[OF col_name] -:

This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.

[ON table_name] :

- This clause identifies the name of the table or view to which the trigger is associated.

[REFERENCING OLD AS o NEW AS n] :-

This clause is used to reference the old and new values of the data being changed. By default, you reference the values as :**old.column_name or :new.column_name.** The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.

[FOR EACH ROW] :-

This clause is used to determine whether a trigger must fire when each row gets affected (i.e. a Row Level Trigger) or just once when the entire sql statement is executed(i.e. Statement Level Trigger).

WHEN (condition) :-

This clause is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

Types of PL/SQL Triggers

There are two types of triggers based on the which level it is triggered.

1) Row level trigger - An event is triggered for each row upated, inserted or deleted.

2) Statement level trigger - An event is triggered for each sql statement executed.

PL/SQL Trigger Execution Hierarchy

The following hierarchy is followed when a trigger is fired.

1) BEFORE statement trigger fires first.

2) Next BEFORE row level trigger fires, once for each row affected.

3) Then AFTER row level trigger fires once for each affected row. This events will alternates between BEFORE and AFTER row level triggers.

4) Finally the AFTER statement level trigger fires.

Example:

SQL>CREATE TABLE EMPLOYEE1(EMPNO NUMBER(10), ENAME VARCHAR(20), ADDRESS VARCHAR(20));

SQL> **INSERT INTO EMPLOYEE2 VALUES(600,'Rama Rao','GUNTUR');**

SQL> **INSERT INTO EMPLOYEE2 VALUES(700,'Chandra'','VIJAYAWAD');**

SQL> **INSERT INTO EMPLOYEE2 VALUES(800,'Kumar','NARSAPUR');**

SQL> SELECT * FROM EMPLOYEE1;

EMPNO	ENAME	ADDRESS
600	Rama Rao	GUNTUR
700	Chandra	VIJAYAWADA
800	Kumar	NARSAPUR

SQL> **CREATE TABLE EMP1(EMPNO NUMBER(10), ENAME VARCHAR(20), ADDRESS VARCHAR(20));**

TRIGGER CREATION:

```
SQL> CREATE OR REPLACE TRIGGER EMP_TRIGG
  2  AFTER UPDATE OR DELETE ON EMPLOYEE1
  3  FOR EACH ROW
  4  BEGIN
  5     INSERT INTO EMP1 VALUES(:OLD.EMPNO, :OLD.ENAME, :OLD.ADDRESS);
  6  END;
SQL> SELECT * FROM EMP1;
no rows selected

SQL> UPDATE EMPLOYEE1 SET ADDRESS='HYDERABAD' WHERE
```

ENAME='Chandra';

.

SQL> SELECT * FROM EMPLOYEE1;

EMPNO	ENAME	ADDRESS
600	Rama Rao	GUNTUR
700	Chandra	HYDERABAD
800	Kumar	NARSAPUR

SQL> **SELECT * FROM EMP1**;

EMPNO	**ENAME**	**ADDRESS**
700	**Chandra**	**VIJAYAWADA**

SQL> **INSERT INTO EMPLOYEE1 VALUES(500,'Kishore','BHIMAVARAM');**

SQL> **INSERT INTO EMPLOYEE1 VALUES(900,'Ravi','VIZAG');**

SQL> SELECT * FROM EMPLOYEE1;

EMPNO	ENAME	ADDRESS
600	Rama Rao	GUNTUR
700	Chandra	HYDERABAD
800	Kumar	NARSAPUR
500	Kishore	BHIMAVARAM
900	Ravi	VIZAG

SQL> **SELECT * FROM EMP1;**

EMPNO	ENAME	ADDRESS
700	Chandra	VIJAYAWADA

SQL> DELETE FROM EMPLOYEE2 WHERE EMPNO='500';

SQL> SELECT * FROM EMPLOYEE2;

EMPNO	ENAME	ADDRESS
600	Rama Rao	GUNTUR
700	Chandra	HYDERABAD
800	Kumar	NARSAPUR
900	Ravi	VIZAG

SQL> **SELECT * FROM EMP1;**

EMPNO	ENAME	ADDRESS

700	Chandra	VIJAYAWADA
500	Kishore	BHIMAVARAM

STATEMENT LEVEL TRIGGER:

Example: The following Trigger does't allow to update the table after 27th of every month.

```
SQL>CREATE OR REPLACE TRIGGER EMP_TRG
      BEFORE UPDATE OF ENAME
      ON EMPLOYEE1
      DECLARE
            Day_of_month  number;
      BEGIN
            Day_of_month:=TO_CHAR(SYSDATE,'DD');
            if Day_of_month BETWEEN 28 AND 31 then
                  RAISE excep;
            end if;
       EXCEPTION
            WHEN excep THEN
                Dbms_output.put_line('Can not update between 28 and 31 dates of month');
       END;
```