18CSC303J - DATABASE MANAGEMENT SYSTEMS

SEMESTER – VI

2021 – 2022 (EVEN)

Name : Kolli Krishna Chaitanya

Register No. : RA1911026010044

Branch : CSE-AI ML

Section : K1

DEPARTMENT OF COMPUTATIONAL INTELLIGENCE

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
(Under Section 3 of UGC Act, 1956)

S.R.M. NAGAR, KATTANKULATHUR – 603 203
CHENGALPATTU DISTRICT

DEPARTMENT OF COMPUTATIONAL INTELLIGENCE
COLLEGE OF ENGINEERING & TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)
S.R.M. NAGAR, KATTANKULATHUR

BONAFIDE CERTIFICATE

Register No:RA1911026010033

Certified to be the bonafide record of work done by **Kolli Krishna Chaitanya** of **CSE-AIML, B.Tech.** Degree course in the Practical of **18CSC303J - DATABASE MANAGEMENT SYSTEMS** in **SRM IST**, Kattankulathur during the academic year **2021 - 2022**.

Staff In-Charge **Head of the Department**

Date :

Submitted for University Examination held on _____ at **SRM IST**, Kattankulathur.

Date : **Internal Examiner I** **Internal Examiner II**

CONTENTS

EXERCISE – 1

Aim: Data Definition Language using SQL COMMANDS

Data Definition Language (DDL) statements are used to define the database structure or schema. Some examples:

- CREATE - to create objects in the database
- ALTER - alters the structure of the database
- DROP - delete objects from the database
- TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
- COMMENT - add comments to the data dictionary
- RENAME - rename an object

The Create Table Command

The create table command defines each column of the table uniquely. Each column has minimum of three attributes.

- Name
- Data type
- Size(column width).

Each table column definition is a single clause in the create table syntax. Each table column definition is separated from the other by a comma. Finally, the SQL statement is terminated with a semicolon.

The Structure of Create

Table Command Table

name is Student

Column name	Data type	Size
Reg_no	varchar2	10
Name	char	30
DOB	date	
Address	varchar2	50

The DROP Command

Syntax:

```
DROP TABLE <table_name>
```

The TRUNCATE Command

Syntax:

```
TRUNCATE TABLE <Table_name>
```

The RENAME Command

```
RENAME <OldTableName> TO <NewTableName>
```

Syntax

The ALTER Table Command

By The use of ALTER TABLE Command we can **modify** our exiting table.

Adding New

Columns

```
ALTER TABLE <table_name>
        ADD (<NewColumnName> <Data_Type>(<size>),..... n)
```

Dropping a Column

```
ALTER TABLE <table_name> DROP COLUMN
<column_name>
```

Modifying Existing Table

Syntax:

```
ALTER TABLE <table_name> MODIFY (<column_name> <NewDataType>(<NewSize>))
```

Restriction on the ALTER TABLE
Using the ALTER TABLE clause the following tasks cannot be performed.

- Change the name of the table
- Change the name of the column
- Decrease the size of a column if table data exists

Lab Experiment:

SQL> CREATE TABLE EMPNEW (EMPNO NUMBER(6), ENAME VARCHAR2(20) NOT NULL, JOB VARCHAR2(10) NOT NULL, DEPTNO NUMBER(3), SAL NUMBER(7,2));

Table created.

SQL> ALTER TABLE EMPNEW ADD EXPERIENCE NUMBER;

Table altered.

SQL> ALTER TABLE EMPNEW MODIFY JOB VARCHAR2(20);

Table altered.

SQL> spool dbmsweek1.lst

SQL> DESCRIBE EMPNEW

Name Null? Type

EMPNO NUMBER(6)

ENAME NOT NULL VARCHAR2(20)

JOB NOT NULL VARCHAR2(20)

DEPTNO NUMBER(3)

SAL NUMBER(7,2)

EXPERIENCE NUMBER

SQL> spool off

SQL> CREATE TABLE DEPT (DEPTNO NUMBER(2) PRIMARY KEY, DNAME VARCHAR2(10), LOC VARCHAR2(10));

Table created.

SQL> DESCRIBE DEPT

Name Null? Type

DEPTNO NOT NULL NUMBER(2)

DNAME VARCHAR2(10)

LOC VARCHAR2(10)

SQL> CREATE TABLE EMPNEW1

2 (ENAME VARCHAR2(20), EMPNO NUMBER(6) CONSTRAINT CH CHECK(EMPNO > 100));

Table created.

SQL> DESCRIBE EMPNEW1

Name Null? Type

ENAME VARCHAR2(20)

EMPNO NUMBER(6)

SQL> ALTER TABLE EMPNEW DROP COLUMN EXPERIENCE;

Table altered.

SQL> DESCRIBE EMPNEW

Name Null? Type

EMPNO NUMBER(6)

ENAME NOT NULL VARCHAR2(20)

JOB NOT NULL VARCHAR2(20)

DEPTNO NUMBER(3)

SAL NUMBER(7,2)

SQL> TRUNCATE TABLE EMPNEW;

Table truncated.

SQL> DROP TABLE DEPT;

Table dropped.

SQL> SELECT TABLE_NAME FROM USER_TABLES

2

```
SQL> SELECT TABLE_NAME FROM USER_TABLES;

TABLE_NAME
----------
EMP
EMP1
EMPNEW
EMPNEW1
SQL> DESCRIBE EMPNEW
Name Null? Type
----------
EMPNO NUMBER(6)
ENAME NOT NULL VARCHAR2(20)
JOB NOT NULL VARCHAR2(20)
DEPTNO NUMBER(3)
SAL NUMBER(7,2)
```

Result: Data Definition Language using SQL COMMANDS has been studied and implemented.

EXERCISE – 2

Aim: To study DML (Data Manipulation Language) using SQL COMMANDS

DML statements affect records in a table. These are basic operations we perform on data such as selecting a few records from a table, inserting new records, deleting unnecessary records, and updating/modifying existing records.

DML statements include the following:

SELECT – select records from a table
INSERT – insert new records
UPDATE – update/Modify existing records
DELETE – delete existing records

DML command

Data Manipulation Language (DML) statements are used for managing data in database. DML commands are not auto-committed. It means changes made by DML command are not permanent to database, it can be rolled back.

INSERT COMMAND

Insert command is used to insert data into a table. Following is its general syntax,

INSERT into *table-name* values(data1,data2,..)

UPDATE COMMAND

Update command is used to update a row of a table. Following is its general syntax,

UPDATE *table-name* set column-name = value *where* **condition**;

DELETE COMMAND

Delete command is used to delete data from a table. Delete command can also be used with conditions to delete a particular row. Following is its general syntax,

DELETE from *table-name*;

WHERE clause

Where clause is used to specify condition while retrieving data from table. ***Where*** clause is used mostly with *Select*, *Update* and *Delete* query. If condition specified by ***where*** clause is true then only the result from table is returned.

Syntax for WHERE clause

SELECT column-name1,
column-name2,
column-name3,
column-nameN
from table-name **WHERE [condition]**;

SELECT COMMAND

SELECT Query

Select query is used to retrieve data from a tables. It is the most used SQL query. We can retrieve complete tables, or partial by mentioning conditions using WHERE clause.

Syntax of SELECT Query

SELECT column-name1, column-name2, column-name3, column-nameN from *table-name*;

Like Clause

Like clause is used as condition in SQL query. **Like** clause compares data with an expression using wildcard operators. It is used to find similar data from the table.

Wildcard operators

There are two wildcard operators that are used in like clauses.

- **Percent sign %** : represents zero, one or more than one character.
- **Underscore sign _** : represents only one character.

Order By Clause

Order by clause is used with the **Select** statement for arranging retrieved data in sorted order. The **Order by clause** by default sort data in ascending order. To sort data in descending order **DESC** keyword is used with **Order by** clause.

Syntax of Order By

SELECT column-list|* from table-name **order by** *asc*|*desc*;

Group By Clause

Group by clause is used to group the results of a SELECT query based on one or more columns. It is also used with SQL functions to group the result from one or more tables.

Syntax for using Group by in a statement.

```
SELECT column_name, function(column_name)
FROM table_name
WHERE condition
GROUP BY column_name
```

HAVING Clause

Having clause is used with SQL Queries to give more precise conditions for a statement. It is used to mention conditions in Group based SQL functions, just like WHERE clauses.

Syntax for having will be,

```
select column_name, function(column_name)
FROM table_name
WHERE column_name condition
GROUP BY column_name
HAVING function(column_name) condition
```

Distinct clause

The **distinct** keyword is used with **Select** statement to retrieve unique values from the table. **Distinct** Removes all the duplicate records while retrieving from database.

Syntax for DISTINCT Keyword

SELECT *distinct* column-name from *table-name*;

AND & OR clause

AND and **OR** operators are used with **Where** clause to make more precise conditions for fetching data from database by combining more than one condition together.

AND operator

AND operator is used to set multiple conditions with *Where* clause.

OR operator

OR operator is also used to combine multiple conditions with the Where clause. The only difference between AND and OR is their behavior. When we use AND to combine two or more than two conditions, records satisfying all the conditions will be in the result. But in the case of OR, at least one condition from the conditions specified must be satisfied by any record to be in the result.

Lab Experiment:

```
SQL> CREATE TABLE STUDENT
2 (RegNo NUMBER(9),
3 Name VARCHAR2(20),
4 Gender VARCHAR(1),
5 DOB DATE,
```

6 mobileno NUMBER(10),

7 City VARCHAR(32));

Table created.

SQL> DESC STUDENT

Name Null? Type

REGNO NUMBER(9)

NAME VARCHAR2(20)

GENDER VARCHAR2(1)

DOB DATE

MOBILENO NUMBER(10)

CITY VARCHAR2(32)

SQL> INSERT INTO STUDENT VALUES

2 (312, 'Bala', 'M', DATE'2000-12-20', 8096735597, 'Rajahmundry');

1 row created.

SQL> INSERT INTO STUDENT VALUES

2 (9531, 'Madhav', 'F', DATE '2015-09-15', 9848035597, 'Rajahmundry');

1 row created.

SQL> INSERT INTO STUDENT VALUES

2 (8088, 'Pavan', 'F', DATE '1986-12-31', 9705710159,

3 'Rajahmundry');

1 row created.

SQL> INSERT INTO STUDENT VALUES

2 (2609, 'Sasi Rao', 'M', DATE '1973-09-26', 9949028509,

3 'Rajahmundry');

1 row created.

SQL> INSERT INTO STUDENT VALUES

2 (3001, 'Harsha', 'M', DATE '2002-01-30', 9884792252, 'Rajahmundry');

1 row created.

```
SQL> INSERT INTO STUDENT VALUES
2 (601, 'Kumar', 'F', DATE '1999-01-06', 7674978787,
3 'Rajahmundry');

1 row created.

SQL> SELECT * FROM STUDENT;

REGNO NAME G DOB MOBILENO
---------- ---------- - --------- ----------
CITY
----------
312 Bala M 20-DEC-00 8096735597
Rajahmundry
9531 Madhav F 15-SEP-15 9848035597
Rajahmundry
8088 Pavan F 31-DEC-86 9705710159
Rajahmundry

REGNO NAME G DOB MOBILENO
---------- ---------- - --------- ----------
CITY
----------
2609 Sasi Rao M 26-SEP-73 9949028509
Rajahmundry
3001 Harsha M 30-JAN-02 9884792252
Rajahmundry
601 Kumar F 06-JAN-99 7674978787
Rajahmundry

6 rows selected.
```

```
SQL> UPDATE STUDENT
2 SET NAME='Ntr' WHERE RegNo=312;
1 row updated.
SQL> SELECT * FROM STUDENT;
REGNO NAME G DOB MOBILENO
---------- ---------
CITY
--------------------------
312 SamM 20-DEC-00 8096735597
Rajahmundry
9531 Madhav F 15-SEP-15 9848035597

Rajahmundry
8088 Pavan F 31-DEC-86 9705710159
Rajahmundry

REGNO NAME G DOB MOBILENO
---------- ---------- - ---------
CITY
--------------------------
2609 Sasi Rao M 26-SEP-73 9949028509
Rajahmundry
3001 Harsha M 30-JAN-02 9884792252
Rajahmundry
601 Kumar F 06-JAN-99 7674978787
Rajahmundry

6 rows selected.
```

```
SQL> UPDATE STUDENT
2 SET NAME='RAM' WHERE RegNo=312;
1 row updated.
SQL> SELECT * FROM STUDENT;
REGNO NAME G DOB MOBILENO
---------- - ----------
CITY
--------------------
312 RAM M 20-DEC-00 8096735597
Rajahmundry
9531 Madhav F 15-SEP-15 9848035597
Rajahmundry
8088 Pavan F 31-DEC-86 9705710159
Rajahmundry

REGNO NAME G DOB MOBILENO
---------- - ----------
CITY
--------------------
2609 Sasi Rao M 26-SEP-73 9949028509
Rajahmundry

3001 Harsha M 30-JAN-02 9884792252
Rajahmundry
601 Kumar F 06-JAN-99 7674978787
Rajahmundry

6 rows selected.
SQL> UPDATE STUDENT
```

2 SET DOB=Date'1983-05-01' WHERE NAME='RAM';

1 row updated.

SQL> SELECT * FROM STUDENT;

REGNO NAME G DOB MOBILENO

CITY

---

312 RAM M 01-MAY-83 8096735597

Rajahmundry

9531 Madhav F 15-SEP-15 9848035597

Rajahmundry

8088 Pavan F 31-DEC-86 9705710159

Rajahmundry

REGNO NAME G DOB MOBILENO

CITY

2609 Sasi Rao M 26-SEP-73 9949028509

Rajahmundry

3001 Harsha M 30-JAN-02 9884792252

Rajahmundry

601 Kumar F 06-JAN-99 7674978787

Rajahmundry

6 rows selected.

```
SQL> CREATE TABLE
EMP(2 EMPNO NUMBER(5),
3 ENAME VARCHAR(20),
4 JOB VARCHAR(50),

5 SALARY NUMBER(10));
Table created.
SQL> DESC EMP
Name Null? Type
----------------------------------------- -------- ----------------------------
EMPNO NUMBER(5)
ENAME VARCHAR2(20)
JOB VARCHAR2(50)
SALARY NUMBER(10)
SQL> INSERT INTO EMP
2 VALUES(1,'Ntr','Asst professor',10000);
1 row created.
SQL> INSERT INTO EMP
2 VALUES(2,'SK','Asst professor',10000);
1 row created.
SQL> INSERT INTO EMP
2 VALUES(2,'SAI','HOD',70000);
1 row created.
SQL> INSERT INTO EMP
2 VALUES(2,'dhoni','CHANCELLOR',90000);
1 row created.
```

SQL> SELECT * FROM EMP;

EMPNO ENAME

JOB SALARY

---

1 Ntr

Asst professor 10000

2 SK

Asst professor 10000

2 SAI

HOD 70000

EMPNO ENAME

JOB SALARY

---

2 dhoni

CHANCELLOR 90000

SQL> UPDATE EMP

2 SET SALARY=15000 WHERE JOB='Asst professor';

2 rows updated.

SQL> SELECT * FROM EMP;

EMPNO ENAME

JOB SALARY

---

1 Ntr

Asst professor 15000

2 SK

Asst professor 15000

2 SAI

HOD 70000

EMPNO ENAME

JOB SALARY

---

2 dhoni

CHANCELLOR 90000

SQL> CREATE TABLE employee AS SELECT * FROM EMP;

Table created.

SQL> SELECT * FROM employee;

EMPNO ENAME

JOB SALARY

---

1 Ntr

Asst professor 15000

2 SK

Asst professor 15000

2 SAI

HOD 70000

EMPNO ENAME

JOB SALARY

---

2 dhoni

CHANCELLOR 90000

SQL> SELECT ENAME,JOB FROM EMP;

ENAME JOB

SamAsst professorSK

Asst professor SAI HOD

dhoni CHANCELLOR

SQL> spool off

Result:

DML (Data Manipulation Language) using SQL COMMANDS has been studied and implemented.

EXERCISE-3

AIM: To write SQL queries to execute different DCL and TCL commands.

Explanation: Database created for this exercise is:

customer_id integer	sale_date date	sale_amount numeric	salesperson character varying (255)	store_state character varying (255)	order_id character varying (255)
1001	2020-05-23	1200	Raj K	KA	1001
1001	2020-05-22	1200	M K	NULL	1002
1002	2020-05-23	1200	Malika Rakesh	MH	1003
1003	2020-05-22	1500	Malika Rakesh	MH	1004
1004	2020-05-22	1210	M K	NULL	1003
1005	2019-12-12	4200	R K Rakesh	MH	1007
1002	2020-05-21	1200	Molly Samberg	DL	1001

Data Control Language (DCL) Commands:

DCL includes commands such as GRANT and REVOKE which mainly deal with the rights, permissions, and other controls of the database system.

List of DCL commands:

- **GRANT:** This command gives users access privileges to the database.
- **REVOKE:** This command withdraws the user's access privileges given by using the GRANT command.

Transaction Control Language (TCL) Commands:

- **COMMIT:** Commits a Transaction.
- **ROLLBACK:** Rollbacks a transaction in case of any error occurs.
- **SAVEPOINT:**Sets a savepoint within a transaction.

Lab Experiment:

SQL> DESC STUDENT;

```
Name                                      Null?    Type
----------------------------------------- -------- ----------------------------
REGNO                                              NUMBER(9)
NAME                                               VARCHAR2(20)
GENDER                                             VARCHAR2(1)
DOB                                                DATE
MOBILENO                                           NUMBER(10)
CITY                                               VARCHAR2(32)

SQL> SELECT * from STUDENT;

    REGNO NAME                 G DOB         MOBILENO
--------- -------------------- - --------- ----------
CITY
------------------------
       312 RAM                  M 01-MAY-83 8096735597
Rajahmundry

      9531 Madhav               M 15-SEP-15 9848035597
Rajahmundry

      8088 Pavan                M 31-DEC-86 9705710159
Rajahmundry


    REGNO NAME                 G DOB         MOBILENO
--------- -------------------- - --------- ----------
CITY
------------------------
```

2609 Sasi Rao M 26-SEP-73 9949028509
Rajahmundry

3001 Harsha M 30-JAN-02 9884792252
Rajahmundry

601 Kumari F 06-JAN-99 7674978787
Rajahmundry

6 rows selected.

SQL> GRANT SELECT ON STUDENT TO RA1911026010033;

Grant succeeded.

SQL> REVOKE SELECT ON STUDENT FROM RA1911026010033;

Revoke succeeded.

SQL> INSERT INTO STUDENTS
VALUES('225','RAM','M',DATE'2002-10-08','98675756423','RAJAHMUNDRY');
INSERT INTO STUDENTS
VALUES('225','RAM','M',DATE'2002-10-08','98675756423','RAJAHMUNDRY')
*

ERROR at line 1:

ORA-00942: table or view does not exist

```
SQL> INSERT INTO STUDENT
VALUES('225','RAM','M',DATE'2002-10-08','98675756423','RAJAHMUNDRY');
INSERT INTO STUDENT
VALUES('225','RAM','M',DATE'2002-10-08','98675756423','RAJAHMUNDRY')
                                        *
ERROR at line 1:
ORA-01438: value larger than specified precision allowed for this column


SQL> INSERT INTO STUDENT
VALUES('225','RAM','M',DATE'2002-10-08','9867575623','RAJAHMUNDRY');

1 row created.

SQL> COMMIT;

Commit complete.

SQL> SELECT * FROM STUDENT;

    REGNO NAME                G DOB         MOBILENO
--------------------------------------- ----------
CITY
--------------------------
       225 RAM                M 08-OCT-02 9867575623
RAJAHMUNDRY

       312 RAM                M 01-MAY-83 8096735597
Rajahmundry
```

9531 Madhav M 15-SEP-15 9848035597
Rajahmundry

REGNO NAME G DOB MOBILENO
---------- - ---------- ----------
CITY

8088 Pavan M 31-DEC-86 9705710159
Rajahmundry

2609 Sasi Rao M 26-SEP-73 9949028509
Rajahmundry

3001 Harsha M 30-JAN-02 9884792252
Rajahmundry

REGNO NAME G DOB MOBILENO
---------- - ---------- ----------
CITY
----
601 Kumari F 06-JAN-99 7674978787
Rajahmundry

7 rows selected.

SQL> DELETE FROM STUDENT WHERE REGNO='312';

1 row deleted.

SQL> SELECT * FROM STUDENT;

REGNO NAME G DOB MOBILENO

CITY

---

225 RAM M 08-OCT-02 9867575623

RAJAHMUNDRY

9531 Madhav F 15-SEP-15 9848035597

Rajahmundry

8088 Pavan F 31-DEC-86 9705710159

Rajahmundry

REGNO NAME G DOB MOBILENO

CITY

2609 Sasi Rao M 26-SEP-73 9949028509

Rajahmundry

3001 Harsha M 30-JAN-02 9884792252

Rajahmundry

601 Kumari F 06-JAN-99 7674978787
Rajahmundry

6 rows selected.

SQL> ROLLBACK;

Rollback complete.

SQL> SELECT * FROM STUDENT;

REGNO NAME G DOB MOBILENO
---------- - ---------
CITY
---

225 RAM M 08-OCT-02 9867575623
RAJAHMUNDRY

312 RAM M 01-MAY-83 8096735597
Rajahmundry

9531 Madhav F 15-SEP-15 9848035597
Rajahmundry

REGNO NAME G DOB MOBILENO
---------- - ---------
CITY

```
----------
      8088 Pavan                F 31-DEC-86 9705710159
Rajahmundry

      2609 Sasi Rao             M 26-SEP-73 9949028509
Rajahmundry

      3001 Harsha               M 30-JAN-02 9884792252
Rajahmundry


    REGNO NAME                G DOB        MOBILENO
---------- ---------- - --------- ----------
CITY
----------
       601 Kumar              F 06-JAN-99 7674978787
Rajahmundry


7 rows selected.

SQL> SAVEPOINT SP1;

Savepoint created.

SQL> DELETE FROM STUDENT WHERE REGNO='312';

1 row deleted.
```

```
SQL> SAVEPOINT SP2;

Savepoint created.

SQL> ROLLBACK TO S1;
ROLLBACK TO S1
*
ERROR at line 1:
ORA-01086: savepoint 'S1' never established in this session or is invalid


SQL> ROLLBACK TO SP1;

Rollback complete.

SQL> SELECT * FROM STUDENT;

    REGNO NAME                G DOB         MOBILENO
--------------------------------------- ----------
CITY
--------------------------
       225 RAM                M 08-OCT-02 9867575623
RAJAHMUNDRY

       312 RAM                M 01-MAY-83 8096735597
Rajahmundry

      9531 Madhav             F 15-SEP-15 9848035597
Rajahmundry
```

```
    REGNO NAME                G DOB        MOBILENO
------------------------------------- ----------
CITY
------------------------
      8088 Pavan                  F 31-DEC-86 9705710159
Rajahmundry

      2609 Sasi Rao               M 26-SEP-73 9949028509
Rajahmundry

      3001 Harsha             M 30-JAN-02 9884792252
Rajahmundry

    REGNO NAME                G DOB        MOBILENO
------------------------------------- ----------
CITY
------------------------
       601 Kumar              F 06-JAN-99 7674978787
Rajahmundry

7 rows selected.
SQL> spool off
```

Result: Thus the DCL and TCL commands are used to modify or manipulate data records present in the customer database tables.

EXERCISE – 4

Aim: Built-In functions in SQL

Functions

Functions accept zero or more arguments and both return one or more results. Both are used to manipulate individual data items.Operators differ from functional in that they follow the format of function_name(arg..). Functions can be classified into **single row functions and group functions**.

Single Row functions

The single row function can be broadly classified as,
o Date Function
o Numeric Function
o Character Function
o Conversion Function
o Miscellaneous Function

The example that follows mostly uses the symbol table "**dual**". It is a table, which is automatically created by oracle along with the data dictionary

Date Function

1. Add_month
This function returns a date after adding a specified date with a specified number of months.

Syntax: Add_months(d,n); where d-date n-number of months
Example: *Select add_months(sysdate,2) from dual;*

2. last_day
It displays the last date of that month.

Syntax: last_day (d); where d-date
Example: *Select last_day ('1-jun-2009') from dual;*

3. Months_between
It gives the difference in the number of months between d1 & d2.

Syntax: month_between (d1,d2); where d1 & d2 –dates
Example: *Select month_between ('1-jun-2009','1-aug-2009') from dual;*

4. next_day

It returns a day followed the specified date.

Syntax: next_day (d,day);
Example: *Select next_day (sysdate,'wednesday') from dual*

5. round

This function returns the date, which is rounded to the unit specified by the format model.
Syntax : round (d,[fmt]);
where d- date, [fmt] – optional. By default date will be rounded to the nearest day
Example: *Select round (to_date('1-jun-2009','dd-mm-yy'),'year') from dual;*
Select round ('1-jun-2009','year') from dual;

Numerical Functions

Command	Query	Output
Abs(n)	Select abs(-15) from dual;	15
Ceil(n)	Select ceil(55.67) from dual;	56
Exp(n)	Select exp(4) from dual;	54.59
Floor(n)	Select floor(100.2) from dual;	100
Power(m,n)	Select power(4,2) from dual;	16
Mod(m,n)	Select mod(10,3) from dual;	1
Round(m,n)	Select round(100.256,2) from dual;	100.26
Trunc(m,n)	Select trunc(100.256,2) from dual;	100.23
Sqrt(m,n)	Select sqrt(16) from dual;	4

Character Functions

Command	Query	Output
initcap(char);	*select initcap("hello") from dual;*	Hello

lower (char); upper (char);	*select lower ('HELLO') from dual;* *select upper ('hello') from dual;*	hello HELLO
ltrim (char,[set]);	*select ltrim ('cseit', 'cse') from dual;*	it
rtrim (char,[set]);	*select rtrim ('cseit', 'it') from dual;*	cse
replace (char,search string, replace string);	*select replace ('jack and jue', 'j', 'bl') from dual;*	black and blue
substr (char,m,n);	*select substr ('information', 3, 4) from dual;*	form

Conversion Function

1. to_char()

Syntax: to_char(d,[format]);
This function converts date to a value of varchar type in a form specified by date format. If format is negelected then it converts date to varchar2 in the default date format.
Example: *select to_char (sysdate, 'dd-mm-yy') from dual;*

2. to_date()

Syntax: to_date(d,[format]);
This function converts character to date data format specified in the form character.
Example: *select to_date('aug 15 2009','mm-dd-yy') from dual;*

Miscellaneous Functions

1. uid – This function returns the integer value (id) corresponding to the user currently logged in.
Example: *select uid from dual;*

2. user – This function returns the logins user name.
Example: *select user from dual;*

3. nvl – The null value function is mainly used in the case where we want to consider null values as zero.
Syntax; nvl(exp1, exp2)
If exp1 is null, return exp2. If exp1 is not null, return exp1.

Example: *select custid, shipdate, nvl(total,0) from order;*

4. vsize: It returns the number of bytes in expression.

Example: *select vsize('tech') from dual;*

Group Functions

A group function returns a result based on group of rows.

1. avg

Example: *select avg (total) from student;*

2.max

Example: *select max (percentagel) from student;*

3.min

Example: *select min (marksl) from student;*

4. sum

Example: *select sum(price) from product;*

Count Function

In order to count the number of rows, count function is used.

1. count(*) – It counts all, inclusive of duplicates and nulls.
Example: *select count(*) from student;*

2. count(col_name)– It avoids null value.
Example: *select count(total) from order;*

3. count(distinct col_name) – It avoids the repeated and null values.
Example: *select count(distinct ordid) from order;*

Group by clause

This allows us to use simultaneous column name and group functions.
Example: *Select max(percentage), deptname from student group by deptname;*

Having clause

This is used to specify conditions on rows retrieved by using group by clause.
Example: *Select max(percentage), deptname from student group by deptname having*

count(*)>=50;

Special Operators:

In / not in – used to select a equi from a specific set of values
Any - used to compare with a specific set of values
Between / not between – used to find between the ranges
Like / not like – used to do the pattern matching

Lab Experiment:

SQL> desc employee;
SQL> Select add_months(sysdate,2) from dual;

ADD_MONTH

14-APR-22

SQL> Select last_day ('1-jun-2009') from dual;

LAST_DAY(

30-JUN-09

SQL> Select month_between ('1-jun-2009','1-aug-2009') from dual;
Select month_between ('1-jun-2009','1-aug-2009') from dual
*
ERROR at line 1:
ORA-00904: "MONTH_BETWEEN": invalid identifier

SQL> Select months_between ('1-jun-2009','1-aug-2009') from dual;

MONTHS_BETWEEN('1-JUN-2009','1-AUG-2009')

-2

SQL> Select next_day (sysdate,'wednesday') from dual
2 ;

NEXT_DAY(

16-FEB-22

SQL> Select round (to_date(`1-jun-2009','dd-mm-yy'),'year') from dual;
ERROR:
ORA-01756: quoted string not properly terminated

SQL> Select round (`1-jun-2009','year') from dual;

ERROR:
ORA-01756: quoted string not properly terminated

SQL> Select round (to_date('1-jun-2009','dd-mm-yy'),'year') from dual;

ROUND(TO_

01-JAN-09

SQL> Select round ('1-jun-2009','year') from dual;
Select round ('1-jun-2009','year') from dual

*
ERROR at line 1:
ORA-01722: invalid number

SQL> SELECT ABS(-15) FROM DUAL;

ABS(-15)

15

SQL> SELECT CEIL(55.67) FROM DUAL;

CEIL(55.67)

56

SQL> SELECT EXP(4) FROM DUAL;

EXP(4)

54.59815

SQL> SELECT FLOOR(100.2) FROM DUAL;

FLOOR(100.2)

100

SQL> SELECT POWER(4,2) FROM DUAL;

POWER(4,2)

16

SQL> SELECT MOD(10,3) FROM DUAL;

MOD(10,3)

1

```
SQL> SELECT ROUND(100.256,2) FROM DUAL;

ROUND(100.256,2)
----------------
100.26

SQL> SELECT TRUNC(100.256,2) FROM DUAL;

TRUNC(100.256,2)
----------------
100.25

SQL> SELECT SQRT(100) FROM DUAL;

SQRT(100)
----------

10

SQL> SELECT initcap(&#39;hello&#39;) FROM DUAL;

INITC
-----
Hello

SQL> SELECT upper(&#39;hello&#39;) FROM DUAL;

UPPER
-----
HELLO

SQL> SELECT lower(&#39;HellLLo&#39;) FROM DUAL;

LOWER(&#39;
-------
hellllo

SQL> SELECT LTRIM(&#39;hello&#39;,&#39;hola&#39;) FROM DUAL;

LTRI
----
ello

SQL> SELECT RTRIM(&#39;hello&#39;,&#39;hola&#39;) FROM DUAL;

RT

--
he

SQL> SELECT REPLACE(&#39;hello hi&#39;,&#39;h&#39;,&#39;jj&#39;) FROM DUAL;

REPLACE(&#39;H
```

---
jjello jji

SQL> SELECT REPLACE('hello hi',2,7) FROM DUAL;

REPLACE(
---
hello hi

SQL> SELECT SUBSTR('hello hi',2,7) FROM DUAL;

SUBSTR(
---
ello hi

SQL> SELECT SUBSTR('hello hi',2,6) FROM DUAL;

SUBSTR
---
ello h

SQL> select to_char (sysdate, 'dd-mm-yy') from dual;

TO_CHAR(

14-02-22

SQL> select to_date('aug 15 2009','mm-dd-yy') from dual;

TO_DATE('

15-AUG-09

SQL> select uid from dual;

UID

107

SQL> select user from dual;

USER

RA1911026010033

SQL> select vsize('tech') from dual;

VSIZE('TECH')

4

SQL> SELECT * FROM EMP;

```
EMPNO ENAME JOB DEPTNO SAL
---------- -------------------- --------------- ---------- ----------
1 Mike Asst professor 1 10000
2 Sam Asst professor 2 10000
3 BAGS Asst professor 3 10000
4 BRAD CHANCELLOR 4 90000
5 VP HOD 5 99999

SQL> SELECT AVG(SAL) FROM EMP;

AVG(SAL)
----------
43999.8

SQL> SELECT MAX(SAL) FROM EMP;

MAX(SAL)
----------
99999

SQL> SELECT MIN(SAL) FROM EMP;

MIN(SAL)
----------
10000

SQL> SELECT SUM(SAL) FROM EMP;

SUM(SAL)
----------
219999

SQL> SELECT COUNT(*) FROM EMP;

COUNT(*)
----------
5

SQL> SELECT COUNT(DISTINCT SAL) FROM EMP;

COUNT(DISTINCTSAL)
------------------
3

SQL> SELECT * FROM EMP;

EMPNO ENAME JOB DEPTNO SAL
---------- -------------------- --------------- ---------- ----------
1 Mike Asst professor 1 10000
2 Sam Asst professor 2 10000
3 BAGS Asst professor 3 10000
4 BRAD CHANCELLOR 4 90000
5 VP HOD 5 99999
```

```
SQL> SELECT * FROM EMP
2 WHERE ENAME=&#39;s%&#39;;

no rows selected

SQL> SELECT * FROM EMP
2 WHERE ENAME LIKE &#39;S%&#39;;

EMPNO ENAME JOB DEPTNO SAL
---------- -------------------- --------------- ---------- ----------
2 Sam Asst professor 2 10000

SQL> SELECT * FROM EMP
2 WHERE ENAME NOT LIKE &#39;S%&#39;;

EMPNO ENAME JOB DEPTNO SAL
---------- -------------------- --------------- ---------- ----------
1 Mike Asst professor 1 10000
3 BAGS Asst professor 3 10000
4 BRAD CHANCELLOR 4 90000
5 VP HOD 5 99999

SQL> SELECT * FROM EMP
2 WHERE SAL BETWEEN 5000 AND 15000;

EMPNO ENAME JOB DEPTNO SAL
---------- -------------------- --------------- ---------- ----------
1 Mike Asst professor 1 10000
2 Sam Asst professor 2 10000
3 BAGS Asst professor 3 10000

SQL> SELECT SUM(SAL) FROM EMP;

SUM(SAL)
----------
219999

SQL> SELECT AVG(SAL) FROM EMP;

AVG(SAL)
----------
43999.8

SQL> SELECT MIN(SAL) FROM EMP;

MIN(SAL)
----------
10000

SQL> SELECT MAX(SAL) FROM EMP;

MAX(SAL)
```

```
----------
99999

SQL> SELECT MIN(SAL), ENAME FROM EMP GROUP BY JOB;
SELECT MIN(SAL), ENAME FROM EMP GROUP BY JOB
*
ERROR at line 1:
ORA-00979: not a GROUP BY expression

SQL> SELECT MIN(SAL) FROM EMP GROUP BY JOB;

MIN(SAL)
----------
99999
10000
90000

SQL> SELECT MIN(SAL),JOB FROM EMP GROUP BY JOB;

MIN(SAL) JOB
---------- ---------------
99999 HOD
10000 Asst professor
90000 CHANCELLOR


  SQL> spool off
```

Result:

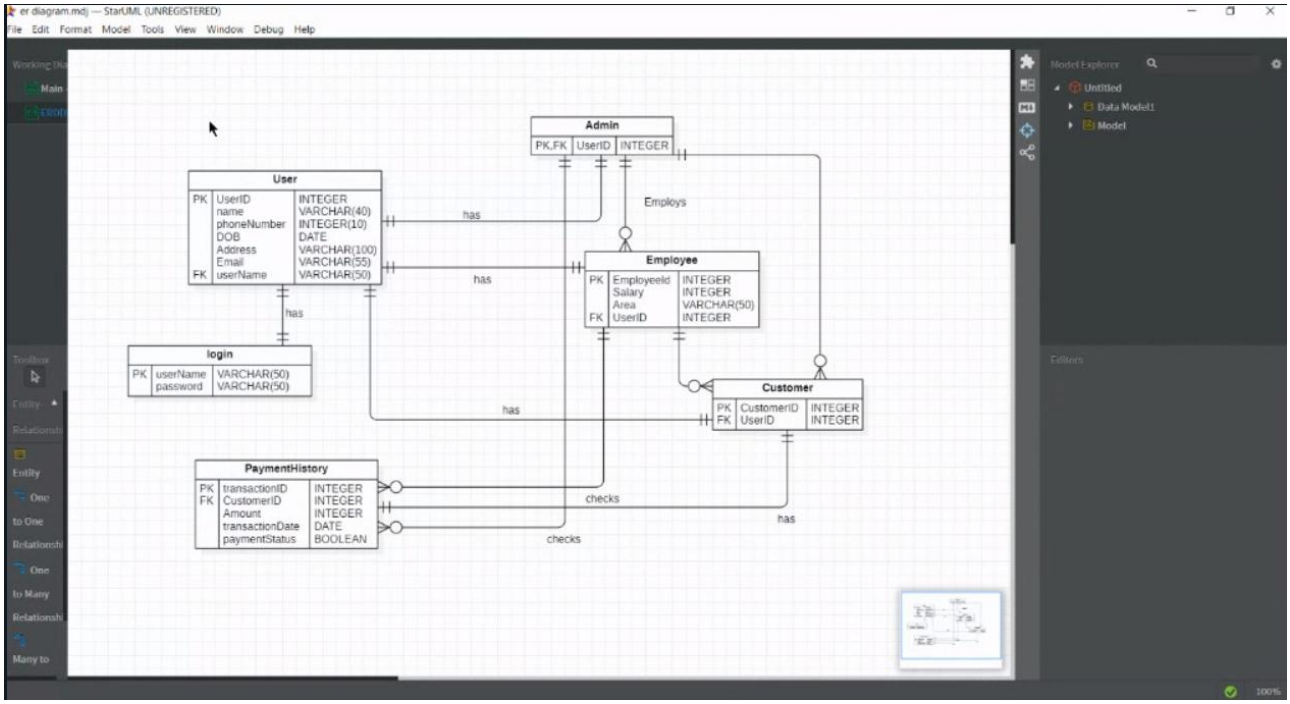
The Built-in Functions in SQL have been implemented.

EXERCISE - 5

Aim: To make an ER model for a Library Management System ER model

EXPLANATION

ER Model stands for Entity Relationship Model is a high-level conceptual data model diagram. ER model helps to systematically analyze data requirements to produce a well-designed database. The ER Model represents real-world entities and the relationships between them.



RESULT – The Extended ER Diagram for Library Management System has been studied.

EXERCISE – 6

AIM - To study JOIN QUERIES in SQL.

EXPLANATION - SQL Join is used to fetch data from two or more tables, which is joined to appear as single set of data. SQL Join is used for combining column from two or more tables by using values common to both tables. **Join** Keyword is used in SQL queries for joining two or more tables. Minimum required condition for joining table, is **(n-1)** where **n**, is number of tables. A table can also join to itself known as, **Self Join**.

Types of Join
The following are the types of JOIN that we can use in SQL.

- Inner
- Outer
- Left
- Right

Cross JOIN or Cartesian Product
This type of JOIN returns the cartesian product of rows from the tables in Join. It will return a table which consists of records which combines each row from the first table with each row of the second table.

Cross JOIN Syntax is,
SELECT column-name-list from table-name1

INNER Join or EQUI Join
This is a simple JOIN in which the result is based on matched data as per the equality condition specified in the query.

Inner Join Syntax is,
SELECT column-name-list from table-name1

Natural JOIN
Natural Join is a type of Inner join which is based on column having same name and same datatype present in both the tables to be joined.

Natural Join Syntax is,
SELECT * from table-name1

NATURAL JOIN
Natural join query will be,
SELECT * from class NATURAL JOIN class_info;

Outer JOIN
Outer Join is based on both matched and unmatched data. Outer Joins subdivide further into,

- Left Outer Join
- Right Outer Join
- Full Outer Join

Left Outer Join
The left outer join returns a result table with the **matched data** of two tables then remaining rows of the **left** table and null for the **right** table's column.

Left Outer Join syntax is,
SELECT column-name-list from table-name1

LEFT OUTER JOIN
Left outer Join Syntax for **Oracle** is,
select column-name-list from table-name1, table-name2 on table-name1.column-name = table-name2.column-name(+);

Left Outer Join query will be,
SELECT * FROM class LEFT OUTER JOIN class_info ON (class.id=class_info.id);

Right Outer Join
The right outer join returns a result table with the **matched data** of two tables then remaining rows of the **right table** and null for the **left** table's columns.

Right Outer Join Syntax is,
select column-name-list from table-name1

RIGHT OUTER JOIN

Right Outer Join query will be,
SELECT * FROM class RIGHT OUTER JOIN class_info on (class.id=class_info.id);
The result table will look like,

Full Outer Join
The full outer join returns a result table with the **matched data** of two table then remaining rows of both **left** table and then the **right** table.
Full Outer Join Syntax is,
select column-name-list from table-name1

FULL OUTER JOIN
Full Outer Join query will be like,
SELECT * FROM class FULL OUTER JOIN class_info on (class.id=class_info.id);

Lab Experiment:

1. CREATE A TABLE STUDENT HAVING COLUMNS LIKE FACULTY_ID, FACULTY_NAME, DEPT_ID,

DEPT_NAME.

SQL> DESC STUDENT1

Name Null? Type

FACULTY_ID NUMBER(6)

FACULTY_NAME VARCHAR2(10)

DEPT_ID NUMBER(6)

DEPT_NAME VARCHAR2(10)

SQL> SELECT * FROM STUDENT1;

FACULTY_ID FACULTY_NA DEPT_ID DEPT_NAME

---

101 Sam3001 CSE

102 Mike 3002 CIVIL

103 sree3003 MECHANICAL

104 dhoni 3004 CHEMICAL

105 prasa 3005 CSE

2. CREATE ANOTHER TABLE COURSE HAVING COLUMNS LIKE FACULTY_ID, STU_NAME AND

COURSE_ID.

SQL> DESC COURSE

Name Null? Type

FACULTY_ID NUMBER(6)

STU_NAME VARCHAR2(10)

COURSE_ID NUMBER(6)

SQL> SELECT * FROM COURSE;

FACULTY_ID STU_NAME COURSE_ID

---

101 Sam1

102 Mike 2

103 sree3

104 dhoni 4

105 prasa 1

4. Use various JOIN OPERATION on Tables

1) SQL> SELECT * FROM STUDENT1 CROSS JOIN COURSE;

FACULTY_ID FACULTY_NA DEPT_ID DEPT_NAME FACULTY_ID STU_NAME COURSE_ID

---

101 Sam3001 CSE 101 Sam1

101 Sam3001 CSE 102 Mike 2

101 Sam3001 CSE 103 sree3

101 Sam3001 CSE 104 dhoni 4

101 Sam3001 CSE 105 prasa 1

102 Mike 3002 CIVIL 101 Sam1

102 Mike 3002 CIVIL 102 Mike 2

102 Mike 3002 CIVIL 103 sree3

102 Mike 3002 CIVIL 104 dhoni 4

102 Mike 3001 CSE 105 prasa 1

103 sree3003 MECHANICAL 101 Sam1

103 sree3003 MECHANICAL 102 Mike 2

103 sree3003 MECHANICAL 103 sree3

FACULTY_ID FACULTY_NA DEPT_ID DEPT_NAME FACULTY_ID STU_NAME COURSE_ID

---

103 sree3003 MECHANICAL 104 dhoni 4

103 sree3001 CSE 105 prasa 1

104 dhoni 3004 CHEMICAL 101 Sam1

104 dhoni 3004 CHEMICAL 102 Mike 2

104 dhoni 3004 CHEMICAL 103 sree3

104 dhoni 3004 CHEMICAL 104 dhoni 4

104 dhoni 3001 CSE 105 prasa 1

104 prasa 3001 CSE 101 Sam1

104 prasa 3001 CSE 102 Mike 2

104 prasa 3001 CSE 103 sree3

104 prasa 3001 CSE 104 dhoni 4

25 rows selected.

2) SQL> SELECT * FROM STUDENT1, COURSE WHERE STUDENT1.FACULTY_ID=COURSE.FACULTY_ID;

FACULTY_ID FACULTY_NA DEPT_ID DEPT_NAME FACULTY_ID STU_NAME COURSE_ID

---

101 Sam3001 CSE 101 Sam1

102 Mike 3002 CIVIL 102 Mike 2

103 sree3003 MECHANICAL 103 sree3

104 dhoni 3004 CHEMICAL 104 dhoni 4

105 prasa 3001 CSE 105 prasa 1

3) SQL> SELECT * FROM STUDENT1 NATURAL JOIN COURSE;

FACULTY_ID FACULTY_NA DEPT_ID DEPT_NAME STU_NAME COURSE_ID

---

101 Sam3001 CSE Sam1

102 Mike 3002 CIVIL Mike 2

103 sree3003 MECHANICAL sree3

104 dhoni 3004 CHEMICAL dhoni 4

105 prasa 3001 CSE prasa 1

4) SQL> SELECT * FROM STUDENT1 LEFT OUTER JOIN COURSE ON

(STUDENT1.FACULTY_ID=COURSE.FACULTY_ID);

FACULTY_ID FACULTY_NA DEPT_ID DEPT_NAME FACULTY_ID STU_NAME COURSE_ID

---

101 Sam3001 CSE 101 Sam1

102 Mike 3002 CIVIL 102 Mike 2

103 sree3003 MECHANICAL 103 sree3

104 dhoni 3004 CHEMICAL 104 dhoni 4

105 prasa 3001 CSE 105 prasa 1

5)SQL> SELECT * FROM STUDENT1 RIGHT OUTER JOIN COURSE ON

(STUDENT1.FACULTY_ID=COURSE.FACULTY_ID);

FACULTY_ID FACULTY_NA DEPT_ID DEPT_NAME FACULTY_ID STU_NAME COURSE_ID

---

101 Sam3001 CSE 101 Sam1

102 Mike 3002 CIVIL 102 Mike 2

103 sree3003 MECHANICAL 103 sree3

104 dhoni 3004 CHEMICAL 104 dhoni 4

105 prasa 3001 CSE 105 prasa 1

6) SQL> SELECT * FROM STUDENT1 FULL OUTER JOIN COURSE ON

(STUDENT1.FACULTY_ID=COURSE.FACULTY_ID);

FACULTY_ID FACULTY_NA DEPT_ID DEPT_NAME FACULTY_ID STU_NAME COURSE_ID

---

101 Sam3001 CSE 101 Sam1

102 Mike 3002 CIVIL 102 Mike 2

103 sree3003 MECHANICAL 103 sree3

104 dhoni 3004 CHEMICAL 104 dhoni 4

105 prasa 3001 CSE 105 prasa 1

7) SQL> SELECT * FROM STUDENT1 LEFT OUTER JOIN COURSE ON

(STUDENT1.DEPT_ID=COURSE.COURSE_ID);

FACULTY_ID FACULTY_NA DEPT_ID DEPT_NAME FACULTY_ID STU_NAME
COURSE_ID

---

102 Mike 3002 CIVIL

103 sree3003 MECHANICAL

104 dhoni 3004 CHEMICAL

101 Sam3001 CSE

105 prasa 3001 CSE

8) SQL> SELECT * FROM STUDENT1 RIGHT OUTER JOIN COURSE ON

(STUDENT1.DEPT_ID=COURSE.COURSE_ID);

FACULTY_ID FACULTY_NA DEPT_ID DEPT_NAME FACULTY_ID STU_NAME
COURSE_ID

---

101 Sam1

102 Mike 2

104 dhoni 4

103 sree3

105 prasa 1

9) SQL> SELECT * FROM STUDENT1 FULL OUTER JOIN COURSE ON

(STUDENT1.DEPT_ID=COURSE.COURSE_ID);

FACULTY_ID FACULTY_NA DEPT_ID DEPT_NAME FACULTY_ID STU_NAME
COURSE_ID

---

101 Sam1

102 Mike 2

103 sree3

104 dhoni 4

105 prasa 1

102 Mike 3002 CIVIL

103 sree3003 MECHANICAL

104 dhoni 3004 CHEMICAL

101 Sam3001 CSE

105 prasa 3001 CSE

10 rows selected.

SQL> SPOOL OFF

Result: JOIN QUERIES in SQL have been successfully implemented.

EXERCISE- 7

Aim: To study SQL SUBQUERIES

Subquery or **Inner query** or **Nested query** is a query in a query. SQL subquery is usually added in the WHERE Clause of the SQL statement. Most of the time, a subquery is used when you know how to search for a value using a SELECT statement, but do not know the exact value in the database.

Subqueries are an alternate way of returning data from multiple tables.

Subqueries can be used with the following SQL statements along with the comparision operators like =, <, >, >=, <= etc.

- SELECT
- INSERT
- UPDATE
- DELETE

2) Let's consider the student_details table which we have used earlier. If you know the name of the students who are studying science subject, you can get their id's by using this query below,

```
SELECT id, first_name
FROM student_details
WHERE first_name IN ('Rahul', 'Stephen');
```

but, if you do not know their names, then to get their id's you need to write the query in this manner,

```
SELECT id, first_name
FROM student_details
WHERE first_name IN (SELECT first_name
FROM student_details
WHERE subject= 'Science');
```

Subquery Output:

id	first_name
100	Rahul
102	Stephen

In the above sql statement, first the inner query is processed first and then the outer query is processed.

SQL Subquery; INSERT Statement

3) Subquery can be used with INSERT statement to add rows of data from one or more tables to another table. Lets try to group all the students who study Maths in a table 'maths_group'.

```
INSERT INTO maths_group(id, name)
SELECT id, first_name || ' ' || last_name
FROM student_details WHERE subject= 'Maths'
```

SQL Subquery; SELECT Statement

4) A subquery can be used in the SELECT statement as follows. Lets use the product and order_items table defined in the sql_joins section.

```
select p.product_name, p.supplier_name, (select order_id from order_items where
product_id = 101) as order_id from product p where p.product_id = 101
```

product_name	supplier_name	order_id
- - - - - - - - - -	- - - - - - - - - -	- - - - - -
Television	Onida	5103

Correlated Subquery

A query is called correlated subquery when both the inner query and the outer query are interdependent. For every row processed by the inner query, the outer query is processed as well. The inner query depends on the outer query before it can be processed.

```
SELECT p.product_name FROM product p
WHERE p.product_id = (SELECT o.product_id FROM order_items o
WHERE o.product_id = p.product_id);
```

Subquery

Notes

Nested

Subquery

1) You can nest as many queries you want but it is recommended not to nest more than 16 subqueries in oracle

Non-Correlated Subquery

2) If a subquery is not dependent on the outer query it is called a non-correlated subquery

Subquery Errors

3) Minimize subquery errors: Use drag and drop, copy and paste to avoid running subqueries with spelling and database typos. Watch your multiple field SELECT comma use, extra or to few getting SQL error message "Incorrect syntax".

SQL Subquery Comments

Adding SQL Subquery comments are good habit (/* your command comment */) which can save you time, clarify your previous work .. results in less SQL headaches

Nested Queries and Performance Issues in SQL

Nested Queries are queries that contain another complete SELECT statement nested within it, that is, in the WHERE clause. The nested SELECT statement is called an "inner query" or an "inner SELECT." The main query is called "outer SELECT" or "outer query." Many nested queries are equivalent to a simple query using JOIN operation. The use of nested query in this case is to avoid explicit coding of JOIN which is a very expensive database operation and to improve query performance. However, in many cases, the use of nested queries is necessary and cannot be replaced by a JOIN operation.

I. Nested queries that can be expressed using JOIN operations:

Example 1: (Library DB Query A) How many copies of the book titled the lost tribe are owned by the library branch whose name is "Sharptown"?

Single Block Query Using Join:

```
SELECT  No_Of_Copies
    FROM    BOOK_COPIES, BOOK, LIBRARY_BRANCH
    WHERE  BOOK_COPIES.BranchId = LIBRARY_BRANCH.BranchId  AND
               BOOK_COPIES.BookId = BOOK.BookId        AND
               BOOK.Title = "The Lost Tribe"  AND
               LIBRARY_BRANCH.BranchName = "Sharpstown";
```

Using Nested Queries:

```
SELECT No_Of_Copies
FROM      BOOK_COPIES
 WHERE       BranchID  IN
                    (SELECT  BranchID  from LIBRARY_BRANCH   WHERE
                              LIBRARY_BRANCH.BranchName = "Sharpstown")
```

```
AND          BookID IN
                  (SELECT  BookID from BOOK  WHERE
                  BOOK.Title = "The Lost Tribe" ) ;
```

Performance considerations: The nested queries in this example involves simpler and faster operations. Each subquery will be executed once and then a simple select operation will be performed. On the other hands, the operations using join require Cartesian products of three tables and have to evaluate 2 join conditions and 2 selection conditions. Nested queries in this example also save internal temporary memory space for holding Cartesian join results.

==========

==========

=

Rule of thumb:

1) ***Correlated queries*** *where the inner query references some attribute of a relation declared in the outer query and use the" =" or IN operators.*
2) *Conversely, if the attributes in the projection operation of a single block query that joins several tables are from only one table, this query can always be translated into a nested query.*

==========

Example 2: see Query 12 and Query 12A

Retrieve the name of each employee who has a dependent with the same first name and same sex as the employee.

Single Block query using JOIN operation

```
select A.fname, A.lname
from employee A, dependent B
where A.ssn = B.essn and
        A.sex = B.sex and A.fname = B.dependent_name;
```

Correlated Query:

```
select  A.fname, A.lname
from employee A
```

```
where A.ssn IN (SELECT essn
                FROM dependent
                WHERE essn = A.ssn  and  dependent_name = A.fname and sex = A.sex);
```

Computer Procedures:

Conceptually, think of this query as stepping through the EMPLOYEE table one row at a time, and then executing the inner query each time. The first row has A.fname = "John" and A.sex = "M" so that the inner query becomes **SELECT Essn FROM dependent where essn = 12345678, dependent_name = "John" and sex = "M"**; The first run of the subquery returns nothing so it continues to proceed to the next tuple and executes the inner query again with the values of A.SSN, A.fname and A.sex for the second row, and so on for all rows of EMPLOYEE.

The term *correlated subquery* is used because its value depends on a variable (or variables) that receives its value from an outer query (e.g., A.SSN, A.fname, A.sex in this example; they are called **correlation variables**.). A correlated subquery thus cannot be evaluated once and for all. It must be evaluated repeatedly -- once for each value of the variable received from the outer query. This is different from non-correlated subqueries explained below.

Non-correlated Subquery:

A non-correlated subquery needs to be evaluated only once. For example:

Query EMP-NQ2: find an employee that has the highest salary of the company.

SELECT fname, lname, bdate

FROM EMPLOYEE

WHERE salary = **(SELECT max (salary) FROM Employee)**;

Here the inner query returns a value: 55000. The inner query will be executed first and only *once* and then the entire query becomes

SELECT fname, lname, bdate

FROM EMPLOYEE WHERE **salary = 55000**;

II. Nested Queries that cannot be directly translated into Join Operations

Rule of thumb:

1) Unknown selection criteria: WHERE clause examines unknown value.

For example shown above (Query EMP-NQ2): find everybody in a department which has an employee that has the highest salary of the company.

Another example in section 7.2.5. finds employees who has salary higher than the highest salary in Department 5.

SELECT ssn, salary, dno from Employee where salary > (SELECT max (salary) from employee where dno = 5);

2) Relational **set** operations such as Division or other comparison that involves EXISTS, NOT EXISTS, > , etc. (This may involve using paradox SET operation operators, such as NO, ONLY, EXACTLY and EVERY.)

3) Outer Join that involves Null value operations. This is the equivalent of using NOT EXISTS. (See *SQL solution for queries on Library DB*: query C and C').

III. General Discussion on SQL query formulation:

There are many ways to specify the same query in SQL. This flexibility in specifying queries has advantage and disadvantages.

- Advantage: You can choose a way to express the query that you prefer. **It is general preferable to write a query with as little nesting and implied ordering as possible.**
- Disadvantages:
  1. the user may be confused

  2. users may have the burden to figure out which way is more efficient due to different DBMS query optimization strategies. (Performance issues.)

Sample Correlated and Non-correlated Subqueries

Write SQL statements for the following queries on the Company Database and determine whether it's a correlated or non-correlated query. (Please translate your SQL single-block join, if applicable, to subqueries.)

Tip: the term *correlated subquery* is used because its value depends on a variable (or variables) that receives its value from an outer query (e.g., A.SSN, A.fname, A.sex in the example shown in the previous handout; they are called **correlation variables**.). A correlated subquery thus cannot be evaluated once and for all. It must be evaluated repeatedly -- once for each value of the variable received from the outer query. A non-correlated subquery needs to be evaluated only once.

Lab Experiment

Create an
employee table and
do the following:
a. Retrieve all
employees who has
a salary greater
than the average
salaries in
department 80.
b. Write a query to
retrieve the
employee number
and last name of all
employees
who work in a
department with
any employee
whose last name
contains the letter
“u.”
c. Retrieve the last
name, department
number, and job ID
of all employees
whose
department
location ID is 1700.

1.

SQL> CREATE

TABLE

EMPLOYEE2

2

SQL> SELECT *

FROM EMP

2 ;

EMPNO ENAME

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

JOB SALARY

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

1 Mike

Asst professor

15000

2 KP

Asst professor

15000

2 bharat

HOD 70000

EMPNO ENAME

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

JOB SALARY

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

2 BRAD

CHANCELLOR

90000

SQL> CREATE

TABLE EMP2

2 (EMPNO

NUMBER(9),

EMPNAME

VARCHAR(30),

DEPTNO

NUMBER(2),

SALARY

NUMBER(8),

DEPTLOCATION

ID NUMBER(4));

Table created.

SQL> DESC

EMP2

Name Null? Type

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

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

EMPNO

NUMBER(9)

EMPNAME

VARCHAR2(30)

DEPTNO

NUMBER(2)

SALARY

NUMBER(8)

DEPTLOCATION

ID NUMBER(4)

SQL> INSERT

INTO EMP2

VALUES (1,

"Mike",

20, 75000, 1500);

INSERT INTO

EMP2 VALUES (1,

"Mike",

20, 75000, 1500)

*

ERROR at line 1:

ORA-00984:

column not

allowed here

```
SQL> INSERT
INTO EMP2
VALUES (1,
&#39;Mike&#39;,
20, 75000, 1500);

1 row created.

SQL> SELECT *
FROM EMP2;

EMPNO
EMPNAME
DEPTNO
SALARY
DEPTLOCATION
ID
---------- ------------
---------------- ------
---- ---------- --------
------
1 Mike 20 75000
1500

SQL> INSERT
INTO EMP2
VALUES (2,
&#39;AJAY&#39;,
40,85000, 7500);
```

1 row created.

SQL> INSERT
INTO EMP2
VALUES (3,
'Mike',
80,95000, 1900);

1 row created.

SQL> INSERT
INTO EMP2
VALUES (4,
'VIKRANTH
', 50, 65000,
1700);

1 row created.

SQL> SELECT *
FROM EMP2;

EMPNO
EMPNAME
DEPTNO
SALARY
DEPTLOCATION
ID

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

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

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

------

1 Mike 20 75000

1500

2 AJAY 40 85000

7500

3 Mike 80 95000

1900

4 VIKRANTH 50

65000 1700

SQL> SELECT *

FROM EMP2

WHERE

SALARY >

(SELECT

AVG(SALARY)

FROM EMP2)

ORDER BY

SALARY;

EMPNO

EMPNAME

DEPTNO

SALARY

DEPTLOCATION

ID

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

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

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

------

2 AJAY 40 85000

7500

3 Mike 80 95000

1900

SQL> SELECT

EMPNO,

EMPNAME

FROM EMP2

WHERE DEPTNO

IN

(SELECT

DEPTNO FROM

EMP2 WHERE

EMPNAME LIKE

'%U%');

no rows selected

SQL> INSERT

INTO EMP2

VALUES (5,

'KP', 10,

6000, 2700);

1 row created.

```
SQL> SELECT
EMPNO,
EMPNAME
FROM EMP2
WHERE DEPTNO
IN
(SELECT
DEPTNO FROM
EMP2 WHERE
EMPNAME LIKE
&#39;%U%&#39;);

EMPNO
EMPNAME
---------- ------------
----------
5 KP

SQL> SELECT
EMPNAME,
DEPTNO FROM
EMP2 WHERE
DEPTLOCATION
ID
IN (SELECT
DEPTLOCATION
ID FROM EMP2
```

```
WHERE
DEPTLOCATION
NO = 1700);
SELECT
EMPNAME,
DEPTNO FROM
EMP2 WHERE
DEPTLOCATION
ID IN
(SELECT
DEPTLOCATION
ID FROM EMP2
WHERE
DEPTLOCATION
NO = 1700)
*
ERROR at line 1:
ORA-00904:
&quot;DEPTLOC
ATIONNO&quot;:
invalid identifier

SQL> SELECT
EMPNAME,
DEPTNO FROM
EMP2 WHERE
DEPTLOCATION
ID
IN (SELECT
```

DEPTLOCATION
ID FROM EMP2
WHERE
DEPTLOCATION
ID = 1700);

EMPNAME
DEPTNO

------- ----------
VIKRANTH 50

2. Create a product
table and do the
following:
a. Retrieve
the products whose
category number is
the same as that of
product
64.
b. Retrieve all
products that cost
more than the
average price in
category no. 60.
c. Retrieve
all products whose
price is equal to

one of the prices of
products in
category 80.
d. Retrieve all
products whose
price is greater than
the prices of all
products in
category 80.
2.

```
SQL> CREATE
TABLE
PRODUCT
2 (PRODUCTID
NUMBER(2),
PRODUCTNAME
VARCHAR(30),
PRICE
NUMBER(6),
CATAGORYNO
NUMBER(2));

Table created.

SQL> INSERT
INTO PRODUCT
VALUES (64,
&#39;BISCUIT&#
39;, 100, 80);
```

1 row created.

SQL> INSERT
INTO PRODUCT
VALUES (65,
'CHOCOLA
TE', 200, 70);

1 row created.

SQL> INSERT
INTO PRODUCT
VALUES (66,
'PEN',
150, 80);

1 row created.

SQL> INSERT
INTO PRODUCT
VALUES (66,
'PENCIL&#3
9;, 175, 60);

1 row created.

SQL> INSERT
INTO PRODUCT

VALUES (67,

'OIL',

125, 60);

1 row created.

SQL> INSERT

INTO PRODUCT

VALUES (68,

'SOAP',

150, 50);

1 row created.

SQL> SELECT *

FROM PRODUCT

2 ;

PRODUCTID

PRODUCTNAME

PRICE

CATAGORYNO

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

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

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

64 BISCUIT 100

80

65 CHOCOLATE

200 70

66 PEN 150 80

66 PENCIL 175 60

67 OIL 125 60

68 SOAP 150 50

6 rows selected.

```
SQL> SELECT
PRODUCTNAME
FROM PRODUCT
WHERE
CATAGORYNO =
(SELECT
CATAGORYNO
FROM PRODUCT
WHERE
PRODUCTID =
64);

PRODUCTNAME
----------
-------
BISCUIT
PEN

SQL> SELECT
PRODUCTNAME
FROM PRODUCT
WHERE PRICE >
```

```
(SELECT
AVG(PRICE)
FROM PRODUCT
WHERE
CATAGORYNO =
60);

PRODUCTNAME
----------
-------
CHOCOLATE
PENCIL

SQL> SELECT
PRODUCTNAME
FROM PRODUCT
WHERE PRICE
IN (SELECT
PRICE FROM
PRODUCT
WHERE
CATAGORYNO =
80);

PRODUCTNAME
----------
-------
BISCUIT
PEN
```

SOAP

```
SQL> SELECT
PRODUCTNAME
FROM PRODUCT
WHERE PRICE =
(SELECT
MAX(PRICE)
FROM PRODUCT
WHERE
CATAGORYNO =
80);

PRODUCTNAME
----------
-------
PEN
SOAP

SQL> SPOOL
OFF
SQL> SPOOL OFF
```

Result - SQL Subqueries have been studied and implemented.

EXERCISE – 8

Aim: To study SET OPERATIONS and VIEWS in SQL

The Set operator combines the result of 2 queries into a single result.The following are the operators:

- **Union**
- Union all
- **Intersect**
- Minus

Rule:

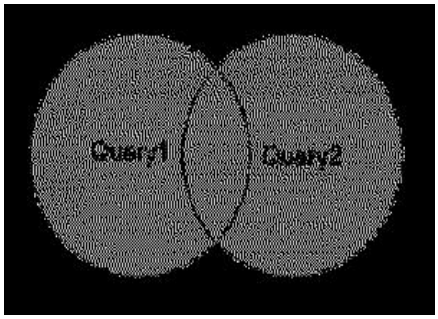
The queries which are related by the set operators should have a same number of column and column definition.

Union:

Returns all distinct rows selected by both the queries

Syntax:

Query1 Union Query2;



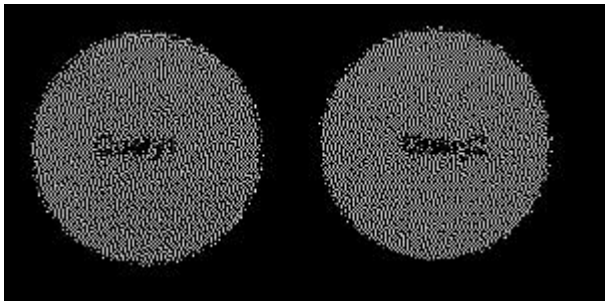
Exp: SELECT * FROM table1 UNION SELECT * FROM table2;

Union all:

Returns all rows selected by either query including the duplicates.

Syntax:

Query1 Union all Query2;



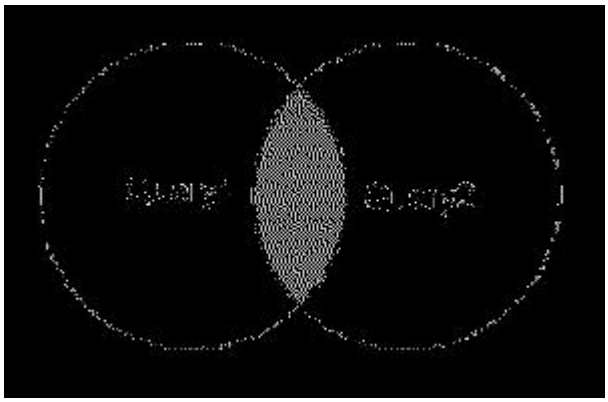
Exp: SELECT * FROM table1 UNION ALL SELECT * FROM table2;

Intersect

Returns rows selected that are common to both queries.

Syntax:

Query1 Intersect Query2;



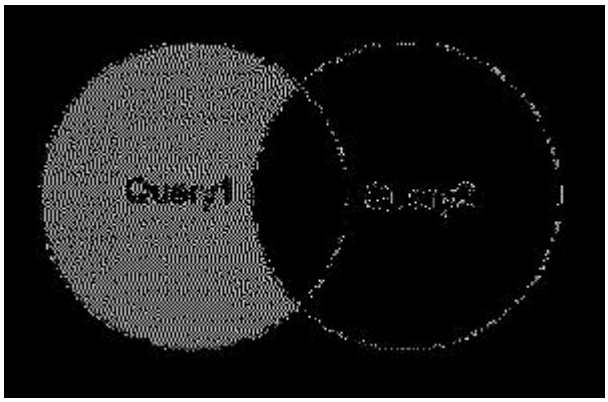
Exp: SELECT * FROM table1 INTERSECT SELECT * FROM table2;

Minus

Returns all distinct rows selected by the first query and are not by the second

Syntax:

Query1 minus Query2;



Exp: SELECT * FROM table1 MINUS SELECT * FROM table2;

VIEWS

A view is the tailored presentation of data contained in one or more table and can also be said as restricted view to the data's in the tables. A view is a "virtual table" or a "stored query" which takes the output of a query and treats it as a table. The table upon which a view is created is called as base table.

Advantages of a view:

a. Additional level of table security.
b. Hides data complexity.

c. Simplifies the usage by combining multiple tables into a single table.
d. Provides data‟s in different perspective.

Types of view:

Horizontal -> enforced by where cause
Vertical -> enforced by selecting the required columns

SQL Commands for Creating and dropping view:

Syntax:
Create [or replace] view <view name> [column alias names] as <query> [with <options> conditions];
Drop view <view name>;

Lab Experiment:

SET OPERATORS:

Create an employee table and perform all set operations (UNION, INTERSECTION, UNIONALL AND MINUS).
1. UNION OR UNIONALL – Which is faster? Justify ur answer once u get the output.

```
SQL> SELECT * FROM EMP
2 UNION
3 SELECT * FROM DEPT;
EID ENAME DEPTNO SAL
---------- ------------------------------ ---------- ----------
1 JOHN 10 59000
2 RAMESH 20 70000
11 Sam10 1000
12 CATCHER 20 5000
SQL> SELECT * FROM EMP
2 UNION ALL
3 SELECT * FROM DEPT;
EID ENAME DEPTNO SAL
---------- ------------------------------ ---------- ----------
1 JOHN 10 59000

2 RAMESH 20 70000
11 Sam10 1000
12 CATCHER 20 5000
1 JOHN 10 59000
```

Both UNION and UNION ALL operators combine rows from result sets into a single result set. The UNION

operator removes eliminate duplicate rows, whereas the UNION ALL operator does not. Because the UNION ALL operator does not remove duplicate rows, it runs faster than the UNION operator.

2. Which Clause will u use along with Union/Unionall Operators to sort the result returned from the query?

```
SQL> SELECT ENAME, DEPTNO FROM EMP UNION
SELECT DNAME, DEPTNO FROM DEPT ORDER BY
DEPTNO;

ENAME DEPTNO
------------------------------ ----------
Sam NAWAS
JOHN 10
RAMESH 20
CATCHER 20

SQL> SELECT ENAME, DEPTNO FROM EMP UNION ALL
SELECT DNAME, DEPTNO FROM DEPT ORDER BY
DEPTNO;

ENAME DEPTNO

------------------------------ ----------
JOHN 10
Sam NAWAS
JOHN 10
CATCHER 20
RAMESH 20
```

3. What is the difference between INTERSECT and INNER JOIN in Oracle? Justify ur answer with an example.

```
SQL> SELECT * FROM EMP
2 INTERSECT
3 SELECT * FROM DEPT;

EID ENAME DEPTNO SAL
---------- ------------------------------ ---------- ----------
1 JOHN 10 59000

SQL> SELECT * FROM EMP
2 INNER JOIN
3 DEPT
4 ON EMP.DEPTNO = DEPT.DEPTNO;

EID ENAME DEPTNO SAL DID
---------- ------------------------------ ---------- ---------- ----------

DNAME DEPTNO TEMP
------------------------------ ---------- ----------
1 JOHN 10 59000 11
```

Sam10 1000

2 RAMESH 20 70000 12
CATCHER 20 5000

1 JOHN 10 59000 1
JOHN 10 59000

The INNER JOIN will return duplicates, if id is duplicated in
either table. INTERSECT removes duplicates. The INNER
JOIN will never return NULL , but INTERSECT will return NULL
.

4. What is the difference between MINUS and NOT IN
operators in Oracle? Justify ur answer with an example.

SQL> SELECT * FROM EMP
2 MINUS
3 SELECT * FROM DEPT;

EID ENAME DEPTNO SAL
---------- ---------- ---------- ----------
2 RAMESH 20 70000

SQL> SELECT * FROM EMP WHERE NOT IN (SELECT *
FROM DEPT);
SELECT * FROM EMP WHERE NOT IN (SELECT * FROM
DEPT)
*
ERROR at line 1:
ORA-00936: missing expression

SQL> SELECT * FROM EMP WHERE ENAME NOT IN
(SELECT DNAME FROM DEPT);

EID ENAME DEPTNO SAL
---------- ---------- ---------- ----------
2 RAMESH 20 70000
Sql> spool off
The fundamental distinction is that MINUS operates at the set
level, while your NOT condition only works on one row at a time
(the same row with the "required" values in the other columns).
Now: You CAN make your query a bit more efficient (although
you can't avoid reading the base table twice).

Result - SET and View Operations have been studied and successfully implemented.

EXP-9:

PL/SQL

Aim : To perform PL/SQL Programs

In addition to SQL commands, PL/SQL can also process data using flow of statements. The flow of control statements are classified into the following categories.

- Conditional control -Branching
- Iterative control – looping
- Sequential control

BRANCHING in PL/SQL:

Sequences of statements can be executed on satisfying certain condition . If statements are being used and differeforms of if are:

1) Simple IF
2) If-Else
3) Nested IF

SIMPLE IF:

Syntax
IF condition THEN statement1; statement2;

END IF;

IF-THEN-ELSE STATEMENT:

Syntax:

IF condition THEN statement1;

ELSE statement2;

END IF;

ELSIF STATEMENTS:

Syntax:

```
IF condition1 THEN statement1; ELSIF
condition2 THEN statement2; ELSIF
condition3 THEN statement3; ELSE
statementn;

END IF;
```

NESTED IF :

Syntax:

```
IF condition THEN statement1;

ELSE
  IF condition THEN statement2;

  ELSE statement3;

  END IF;
END IF;
ELSE statement3;

END IF;
```

SELECTION IN PL/SQL(Sequential Controls) SIMPLE CASE

Syntax:

```
CASE SELECTOR
WHEN Expr1 THEN statement1;
WHEN Expr2 THEN statement2;
          :
ELSE Statement n;

END CASE;
```

SEARCHED CASE:
CASE
WHEN searchcondition1 THEN statement1;
WHEN searchcondition2 THEN statement2; :
ELSE statement n;
END CASE;

ITERATIONS IN PL/SQL

Sequence of statements can be executed any number of times using loop construct. It is broadly classified into:

- Simple Loop
- For Loop
- While Loop

SIMPLE LOOP
Syntax:

LOOP statement1;

EXIT [WHEN Condition];
END LOOP;

WHILE LOOP

Syntax

WHILE condition LOOP statement1; statement2;

END LOOP;

FOR LOOP

Syntax:

FOR counter IN [REVERSE] LowerBound..UpperBound

LOOP
statement1; statement2;
END LOOP;

SQL> SET SERVEROUTPUT ON

Q.1 Write a PL/SQL code to set the sales commission to 10%, if the sales revenue is greater than 200,000.

Otherwise, the sales commission is set to 5%.

```
SQL > DECLARE
2 A NUMBER := 300000;
3 B NUMBER( 10, 2 ) := 0;
4 BEGIN
5 IF A > 200000 THEN
6 B := A * 0.1;
7 ELSE
8 B := A * 0.05;
9 END IF;
10 dbms_output.put_line(B);
11 END;
12 /

30000
PL/SQL procedure successfully completed.
```

Q.2 Use PL/SQL CASE statement where if monthly_value is equal to or less than 4000, then income_level will be set to 'Low Income'.If monthly_value is equal to or less than 5000, then income_level will be set to

'Avg Income'. Otherwise, income_levelwill be set to 'High Income'.

```
SQL > DECLARE
2 A NUMBER (5);
3 BEGIN
4 A:=&A;
5 IF A < 4000 THEN
```

```
6 dbms_output.put_line('LOW INCOME);
7 ELSIF A > 4000 AND A < 5000 THEN
8 dbms_output.put_line('AVG INCOME);
9 ELSE
10 dbms_output.put_line('HIGH INCOME);
11 END IF;
12 END;
13 /

ENTER THE VALUE OF A: 4600
old 4: A:=&A new
4: A:=4600
AVG INCOME

PL/SQL procedure successfully completed.
```

Q.3 Write a PL/SQL program to check whether a number is armstrong number or not.

```
SQL> DECLARE
2 N NUMBER :=&N;
3 S NUMBER:=0;
4 R NUMBER;
5 LEN NUMBER;
6 M NUMBER;
7 BEGIN
8 M:=N;
9 LEN:=LENGTH(TO_CHAR(N));
10 WHILE N>0
 11
12 LOOP
13 R:=MOD(N,10);
14 S:=S+POWER(R,LEN);
15 N:=TRUNC(N/10);
16 END LOOP;
17 IF M=S
18 THEN
19 dbms_output.put_line('ARMSTRONG NUMBER');
20 ELSE
21 dbms_output.put_line('NOT ARMSTRONG NUMBER');
```

```
22 END IF;
```

```
23 END;
24 /
Enter value for n: 153 old 2:
N NUMBER :=&N;
new 2: N NUMBER :=153;
ARMSTRONG NUMBER

PL/SQL procedure successfully completed.
```

Q.4 Write a PL/SQL program by using SELECT INTO statement to get the name of a customer based on the customer id,

which is the primary key of the customers table.

```
SQL> create table customer(ID number(10) PRIMARY KEY,
```

SQL> create table customer(ID number(10) PRIMARY KEY, NAME varchar(10));

Table created.

SQL> insert into customer values(1, 'Pavan');

1 row created.

SQL> insert into customer values(2, 'KP');

1 row created.

SQL> insert into customer values(3, 'SID');

1 row created.

SQL> insert into customer values(4, 'Pavan');

1 row created.

SQL> select * from customer;

```
        ID NAME
---------- ----------
         1 Pavan
         2 KP
         3 SID
         4 Pavan
```

```
SQL> DECLARE
  2  CUSTOMER_NAME CUSTOMER.NAME%TYPE;
```

```
  3  BEGIN
  4  SELECT NAME INTO CUSTOMER-NAME FROM CUSTOMER WHERE ID=2;
  5  dbms_output.put_line('Name:'||CUSTOMER_NAME);
  6  END;
  7  /
Name:Pavan

PL/SQL procedure successfully completed.

SQL> SPOOL OFF
```

RESULT: Hence PL/SQL Programs were studied and performed

EXP-10: PROCEDURES IN PL/SQL

Aim : To perform Procedures program in PL/SQL

Subprogram is a program unit that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called a calling program.

A subprogram can be created −

- At the schema level
- Inside a package
- Inside a PL/SQL block

At the schema level, subprogram is a standalone subprogram. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a packaged subprogram. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter 'PL/SQL - Packages'.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters.

PL/SQL provides two kinds of subprograms –

- Functions − These subprograms return a single value; mainly used to compute and

return a value.

- Procedures − These subprograms do not return a value directly; mainly used to perform an action.

Parts of a PL/SQL Subprogram

Each PL/SQL subprogram has a name, and may also have a parameter list. Like anonymous PL/SQL blocks, the named blocks will also have the following three parts –

S.No	Parts & Description
1	Declarative Part It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.
2	Executable Part This is a mandatory part and contains statements that perform the designated action.
3	Exception-handling

Creating a Procedure

A procedure is created with the CREATE OR REPLACE PROCEDUREstatement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows –

```
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.

Deleting a Standalone Procedure

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

DROP PROCEDURE procedure-name;

You can drop the greetings procedure by using the following statement − DROP PROCEDURE greetings;

Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL

subprograms − S.No Parameter Mode & Description

1	IN An IN parameter lets you pass a value to the subprogram. It is a read-only parameter. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. It is the default mode of parameter passing. Parameters are passed by reference.
2	OUT
	An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. The actual parameter must be variable and it is passed by value.

3	IN OUT An IN OUT parameter passes an initial value to a subprogram and returns a updated value to the caller. It can be assigned a value and the value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a

value. Actual parameter is passed by value.

Methods for Passing Parameters

Actual parameters can be passed in three ways −

- Positional notation
- Named notation
- Mixed notation

Positional Notation

In positional notation, you can call the procedure as −

```
findMin(a, b, c, d);
```

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So,
a is substituted for x, b is substituted for y, c is substituted for z and d is substituted for m. Named Notation

In named notation, the actual parameter is associated with the formal parameter using the arrow symbol (=>). The procedure call will be like the following −

```
findMin(x => a, y => b, z => c, m => d);
```

Mixed Notation

In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

The following call is legal −

```
findMin(a, b, c, m => d);
```

However, this is not legal: findMin(x
=> a, b, c, d);

Result :

Programs related to Procedure in PL/SQL has been successfully implemented.

Q.1 Write a PL/SQL block to get the salary of the employee who has empno=7369 and update his salary as specified below

• if his/her salary < 2500, then increase salary by 25%
• otherwise if salary lies between 2500 and 5000, then increase salary by 20% • otherwise increase salary by adding commission amount to the salary.

```
SQL > Create table emp
(Depid number(3),
empno number(4),
salary number(5),
depname varchar(6));

SQL > insert into emp values(71,7369,2600,'Civil');
1 Row(s) inserted
SQL > insert into emp values(73,7379,2800,'HRD');
1 Row(s) inserted
SQL > insert into emp values(71,7379,2200,'HRD');
1 Row(s) inserted
SQL > select * from emp;

  DEPID EMPNO SALARY DEPNAME  71 7369
              2600 Civil
     73 7379 2800 HRD
71 7379 2200  3  HRD
rows selected.


SQL > create or replace procedure updatesalary (salary in
number, Depid in number) as  2 begin
3 if salary < 2500 then
4 update emp set salary = 2500 + 0.25*2500 where Depid = 7369;
```

```
5 elsif salary >= 2500 and salary < 5000 then

6 update emp set salary = 2500 + 0.20*2500 where Depid = 7369;
7 else
8 update emp set salary = 5000 + 0.15*2500 where Depid = 7369;

9 end if;

10 end;
11 /

Procedure created.

SQL > execute updatesalary(2600, 7369)
```

Q.2 Write a PL/SQL Block to modify the department name of the department 71 if it is not 'HRD'.

```
SQL > create or replace procedure updatedepartment (depname in
varchar, Depid in number) as  2 begin
3 update emp set depname = 'HRD' where Depid = 71 and
depname not in ('HRD');  4 end;
5 /

Procedure Created

SQL > execute updatesalary('Civil', 71)
```

RESULT: Hence Procedures program in PL/SQL were studied and performed.

EXP-11:
FUNCTIONS IN PL/SQL

To perform functions in PL/SQL.

A function is the same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

Creating a Function

A standalone function is created using the CREATE FUNCTION statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

```
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. Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

SQL> set serveroutput on

```
Q.1 Write a PL/SQL Function to find factorial of
a given number. SQL> CREATE OR REPLACE FUNCTION
FAC(N NUMBER) 2 RETURN NUMBER IS
3 I NUMBER(10);
4 F NUMBER:=1;
```

```
5 BEGIN
6 FOR I IN 1..N LOOP
7 F:=F*I;
8 END LOOP;
9 RETURN F;
10 END;
11 /
Function created.
SQL> SELECT FAC(5) FROM DUAL;
 FAC(5)
---------
 120
```

Q.2 Write a PL/SQL Function that computes and returns the maximum of two values.

Q.3

```
SQL> DECLARE
2 a number;
3 b number;
4 c number;
5 FUNCTION findMax(x IN number, y IN number)
6 RETURN number
7 IS
8 z number;
9 BEGIN
10 IF x > y THEN
11 z:= x;
12 ELSE
13 Z:= y;
14 END IF;
15 RETURN z;
16 END;
17 BEGIN
18 a:= 23;
19 b:= 45;
20 c := findMax(a, b);
21 dbms_output.put_line(' Maximum of
(23,45): ' || c);  22 END;
23 /
Maximum of (23,45): 45
PL/SQL procedure successfully completed.
```

Result:

Programs related to Function in PL/SQL have been successfully implemented.

EXP-12: CURSORS IN PL/SQL

AIM:

To implement the concept of CURSORS in PL/SQL

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the active set.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors
–

- Implicit cursors
- Explicit cursors

Implicit Cursors:

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor, which always has attributes such as %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement. The following table provides the description of the most used attributes −

S.No Attribute & Description

1	%FOUND Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	%NOTFOUND The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3	%ISOPEN Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	%ROWCOUNT Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Explicit Cursors:

Explicit cursors are programmer-defined cursors for gaining more control over the context area. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is −

CURSOR cursor_name IS select_statement;

Working with an explicit cursor includes the following steps −

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example −

```
CURSOR c_customers IS
 SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows −

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −

```
FETCH c_customers INTO c_id, c_name, c_addr; Closing
```

the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above opened cursor as follows −

```
CLOSE c_customers;
```

```
SQL> set serveroutput on
```

Q.1 Write a PL/SQL Block, to update salaries of all the employees who work in deptno 20 by 15%. If none of the employee's salary are updated display a message 'None of the salaries were updated'. Otherwise display the totalnumber of employee who got salary updated.

```
SQL> create table emp(
2 dept_id number(10),
3 salary number(10));
Table Created

SQL> desc emp
-----------------------------------------
Column Null? Type
DEPT_ID - NUMBER(10,0) SALARY
    - NUMBER(10,0)

SQL > Insert into emp values(10,1000); 1
row(s) inserted.
SQL >Insert into emp values(20,1500); 1
row(s) inserted.
SQL >Insert into emp values(20,1800); 1
row(s) inserted.
SQL >Insert into emp values(30,1200); 1
row(s) inserted.

SQL > select * from emp;
-------------------------------------
DEPT_ID SALARY
20 1500
20 1800
10 1000
30 1200

4 rows selected.
```

```
Declare num
number(5);
Begin
update emp set salary = salary +
salary*0.15 where dept_id=20;  if
SQL%NOTFOUND then
dbms_output.put_line('none of the salaries were updated');
elsif SQL%FOUND then
num := SQL%ROWCOUNT;
dbms_output.put_line('salaries for '|| num || '
employees are updated'); end  if;
End;
/

PL/SQL procedure successfully completed.
salaries for 2 employees are updated

SQL > select * from emp ;
--------------------------------
DEPT_ID SALARY
20 1725
20 2070
10 1000
30 1200
 4 rows selected.
```

Q.2 Create a table emp_grade with columns empno & grade. Write PL/SQL block to insert values into the table emp_grade by processing emp table with the following constraints. If sal <= 1400 then grade is 'C' Else if sal between 1401 and 2000 then the grade is 'B' Else the grade is 'A'.

```
SQL > create table emp_grade
2 (emp_no number(10),
3 garde varchar(2),
4 sal number(5));
```

```
Table created.
SQL > insert into emp_grade
values  (10,1400);  1  row(s)
inserted.
SQL > insert into emp_grade
values  (20,1600);  1  row(s)
inserted.
SQL > insert into emp_grade
values  (30,2000);  1  row(s)
inserted.
SQL > insert into emp_grade
values  (40,2200);  1  row(s)
inserted.

SQL > select * from emp_grade;
-------------------------------------
EMP_NO SAL
20 1600
30 2000
40 2200
10 1400

4 rows selected.

SQL> spool off
```

Result:

Programs related to concept of cursors in PL/SQL has been successfully implemented.

EXP-13: TRIGGERS IN PL/SQL

AIM: To perform the implementation of the concept TRIGGERS IN PL/SQL

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events –

A database manipulation (DML) statement (DELETE, INSERT, or UPDATE) A database definition (DDL) statement (CREATE, ALTER, or DROP).

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

Triggers can be defined on the table, view, schema, or database with which the event is associated. Benefits of Triggers

Triggers can be written for the following purposes −

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating 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)
DECLARE
 Declaration-statements
BEGIN
 Executable-statements
EXCEPTION
 Exception-handling-statements END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the trigger_name.
- {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be

executed. The INSTEAD OF clause is used for creating trigger on a view.

- {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
- [OF col_name] − This specifies the column name that will be updated.
- [ON table_name] − This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

The following points need to be considered here −

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES
(7, 'Kriti', 22, 'HP', 7500.00 );
```

When a record is created in the CUSTOMERS table, the above trigger, display_salary_changes will be fired and it will display the following result − create

```
Old salary: New
salary: 7500
Salary difference:
```

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The U

PDATE statement will update an existing record in the table −

```
UPDATE customers
SET salary = salary + 500 WHERE
id = 2;
```

When a record is updated in the CUSTOMERS table, the above trigger, display_salary_changes will be fired and it will display the following result − create

```
Old salary: 1500
New salary: 2000
```

1. Create a Trigger to check the entered age is valid or not.

```
CREATE OR REPLACE TRIGGER age_validation
BEFORE INSERT on EMP1
FOR EACH ROW

DECLARE
emp_age number;

BEGIN
   -- Finding employee age by date of birth
   SELECT MONTHS_BETWEEN(TO_DATE(sysdate,'DD-MON-YYYY'), TO_DATE(:new.dob,'DD-MON-YYYY'))/12
  INTO EMP_AGE FROM DUAL;

  -- Check whether employee age is greater than 18 or not
   IF (EMP_AGE < 18) THEN
     RAISE_APPLICATION_ERROR(-20000,'Employee age must be greater than or equal to 18.');
   END IF;

   -- Allow only past date of death
   IF(:new.dob > sysdate) THEN
     RAISE_APPLICATION_ERROR(-20000,'Date of birth can not be Future date.');
   END IF;
END;
/

CREATE TABLE EMP1
```

```
(
    "emp_id" INT,
    "emp_name" VARCHAR2(20),
    "job_id" VARCHAR2(20),
    "mobile" NUMBER(10),
    "salary" NUMBER(10,2),
    "dob" DATE,
    "dept_id" INT,
    PRIMARY KEY ("emp_id")
);

CREATE OR REPLACE TRIGGER age_validation
BEFORE INSERT on EMP1
FOR EACH ROW

DECLARE
emp_age number;

BEGIN
    -- Finding employee age by date of birth
    SELECT MONTHS_BETWEEN(TO_DATE(sysdate,'DD-MON-YYYY'), TO_DATE(:new.dob,'DD-MON-YYYY'))/12
   INTO EMP_AGE FROM DUAL;

   -- Check whether employee age is greater than 18 or not
    IF (EMP_AGE < 18) THEN
      RAISE_APPLICATION_ERROR(-20000,'Employee age must be greater than or equal to 18.');
    END IF;

    -- Allow only past date of death
    IF(:new.dob > sysdate) THEN
      RAISE_APPLICATION_ERROR(-20000,'Date of birth can not be Future date.');
    END IF;
END;
/
```

2. Create a row-level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on that table.

```
CREATE OR REPLACE TRIGGER age_validation
BEFORE INSERT on EMP1
FOR EACH ROW

DECLARE
emp_age number;

BEGIN
    -- Finding employee age by date of birth
    SELECT MONTHS_BETWEEN(TO_DATE(sysdate,'DD-MON-YYYY'), TO_DATE(:new."dob",'DD-MON-
YYYY'))/12
   INTO EMP_AGE FROM DUAL;

   -- Check whether employee age is greater than 18 or not
    IF (EMP_AGE < 18) THEN
      RAISE_APPLICATION_ERROR(-20000,'Employee age must be greater than or equal to 18.');
    END IF;
```

```
   -- Allow only past date of death
   IF(:new."dob" > sysdate) THEN
     RAISE_APPLICATION_ERROR(-20000,'Date of birth can not be Future date.');
   END IF;
END;
/

create table CUSTOMER
(
   "customer_id" number PRIMARY KEY,
   "customer_name" varchar(20),
   "address" varchar(30),
   "mobile" INTEGER
);

CREATE OR REPLACE TRIGGER customer_update
BEFORE DELETE OR INSERT OR UPDATE ON customer
FOR EACH ROW
WHEN (NEW."customer_id" > 0)

BEGIN
   dbms_output.put_line('Changes to CUSTOMER table triggered');
END;

/

INSERT INTO CUSTOMER
WITH input AS
   ( SELECT 1, 'Ramcharan', 'Telangana', 9090897867  FROM DUAL
     UNION ALL
     SELECT 2, 'Prabhas', 'Andhra Pradesh', 8768757890 FROM DUAL
     UNION ALL
     SELECT 3, 'Allu Arjun', 'Tamil Nadu', 8659090897 FROM DUAL
     UNION ALL
     SELECT 4, 'Yash', 'Tamil Nadu', 8659090897 FROM DUAL
   )
SELECT * FROM input;
```

Result:

Programs related to Trigger in PL/SQL have been successfully implemented.

EXP-14:
EXCEPTION HANDLING IN PL/SQL

AIM:

To implement the concept of exception handling in PL/SQL.

In PL/SQL a warning or error condition is called an exception. Exceptions can be internally defined (by the runtime system) or user-defined. Examples of internally defined exceptions include division by zero and out of memory.

Predefined Exceptions

CURSOR_ALREADY_OPEN is raised if you try to OPEN an already open cursor.

DUP_VAL_ON_INDEX is raised if you try to store duplicate values in a database column that is constrained by a unique index.

INVALID_CURSOR is raised if you try an illegal cursor operation. For example, if you try to CLOSE an unopened cursor.

INVALID_NUMBER is raised in a SQL statement if the conversion of a character

string to a number fails.

LOGIN_DENIED is raised if you try logging on to ORACLE with an invalid username/password.

NO_DATA_FOUND is raised if a SELECT INTO statement returns no rows or if you reference an uninitialized row in a PL/SQL table.

NOT_LOGGED_ON is raised if your PL/SQL program issues a database call without being logged on to ORACLE. PROGRAM_ERROR is raised if PL/SQL has an internal problem.

STORAGE_ERROR is raised if PL/SQL runs out of memory or if memory is corrupted. TIMEOUT_ON_RESOURCE is raised if a timeout occurs while ORACLE is waiting for a resource.

TOO_MANY_ROWS is raised if a SELECT INTO statement returns more than one row. VALUE_ERROR is raised if an arithmetic, conversion, truncation, or constraint error occurs.

ZERO_DIVIDE is raised if you try to divide a number by zero.

Handling Raised Exception

Syntax :

```
EXCEPTION
WHEN ... THEN
- handle the error differently
WHEN ... OR ... THEN
- handle the error differently
WHEN OTHERS THEN
- handle the error differently
END;
```

User Defined Exception:

Unlike predefined exceptions, user-defined exceptions must be declared and must be raised explicitly by RAISE statements. Exceptions can be declared only in the declarative part of a PL/SQL block, subprogram, or package. You declare

an exception by introducing its name, followed by the keyword EXCEPTION. Exception

Declaration Ex.

DECLARE

past_due EXCEPTION; acct_num

NUMBER(5);

BEGIN

Exceptions and variable declarations are similar. But remember, an exception is an error condition, not an object. Unlike variables, exceptions cannot appear in assignment statements or SQL statements.

Syntax.

Exception-name Exception;

Using Raise statement

User-defined exceptions must be raised explicitly by RAISE statements. Syntax

RAISE exception-name;

Raise_Application_Error

This is a procedure to issue user-defined error messages from a stored subprogram or database trigger.

Syntax : raise_application_error(error_number, error_message);

where error_number is a negative integer in the range -20000..-20999 and

error_message is a character string up to 512 bytes in length.

Ex. ….

IF salary is NULL THEN

raise_application_error(-20101, 'Salary is missing');

```
SQL> SET SERVEROUTPUT ON
```

Q.1 Write a PL/SQL program that accepts a customer id as an input and returns the customer name using exception handling.

```
SQL> create table customer(c_id number(10), c_name
varchar2(10)); Table  created.
SQL> insert into customer values(1, 'VD'); 1
row(s) inserted.
SQL> insert into customer values(2, 'Jack'); 1 row(s)
inserted.
SQL> insert into customer values(3, 'Tim'); 1
row(s) inserted.
SQL> insert into customer values(4, 'SK'); 1
row(s) inserted.

SQL> DECLARE
2 l_name customer.c_name%TYPE;
3 l_customer_id customer.c_id%TYPE := 4;
4 BEGIN
5 SELECT c_name INTO l_name
6 FROM customer
7 WHERE c_id = l_customer_id;
8 dbms_output.put_line('Customer name is ' || l_name);
9 EXCEPTION
10 WHEN NO_DATA_FOUND THEN
11 dbms_output.put_line('Customer ' || l_customer_id || '  does
```

```
not exist');
12 END;
13 /

Customer name is SK

SQL> DECLARE
2 l_name customer.c_name%TYPE;
3 l_customer_id customer.c_id%TYPE := 10;
4 BEGIN
5 SELECT c_name INTO l_name
6 FROM customer
7 WHERE c_id = l_customer_id;
8 dbms_output.put_line('Customer name
is ' || l_name);  9 EXCEPTION
10 WHEN NO_DATA_FOUND THEN
11 dbms_output.put_line('Customer ' ||
l_customer_id || ' does not exist');  12 END;
13 /

Customer 10 does not exist
SQL>SPOOL OFF
```

Result:
The implementation of the concept exception handling in PL/SQL has been successfully implemented