18CSC303J - DATABASE MANAGEMENT SYSTEMS

SEMESTER – VI

2021 – 2022 (EVEN)

Name : HARSH SAXENA

Register No. : RA1911033010060

Branch : CSE - SWE

Section : M2

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. : RA1911033010060

Certified to be the bonafide record of work done by **HARSH SAXENA of CSE with spl. in SWE, B.Tech. degree course in the** Practical of **18CSC303J - DATABASE MANAGEMENT SYSTEMS** under the guidance of **Ms. Sasi Rekha Sankar** 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

Data Definition Language 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

Example:

```
CREATE TABLE Student
        (Reg_no varchar2(10),
         Name char(30),
         DOB date,
         Address varchar2(50));
```

The DROP Command

Syntax:

```
DROP TABLE <table_name>
```

Example:

```
DROP TABLE Student;
```

It will destroy the table and all data which will be recorded in it.

The TRUNCATE Command

Syntax:

```
TRUNCATE TABLE <Table_name>
```

Example:

```
TRUNCATE TABLE Student;
```

The RENAME Command

Syntax:

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

Example:

```
RENAME <Student> TO <Stu>
```

The old name table was **Student** now new name is the **Stu.**

The ALTER Table Command

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

Adding New Columns

Syntax:

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

Example:

```
ALTER TABLE Student ADD (Age number(2), Marks number(3));
```

The Student table is already exist and then we added two more columns **Age** and **Marks** respectively, by the use of above command.

Dropping a Column from the Table

Syntax:

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

Example:

```
ALTER TABLE Student DROP COLUMN Age;
```

This command will drop particular column

Modifying Existing Table

Syntax:

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

Example:

```
ALTER TABLE Student MODIFY (Name Varchar2(40));
```

The Name column already exist in Student table, it was char and size 30, now it is modified by Varchar2 and size 40.

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-1 :

DATA DEFINITION LANGUAGE

AIM : To execute DDL commands and queries in SQL.

Q1. Create a table called EMP with the following structure.
Name Type
---------- ------------
EMPNO NUMBER(6)
ENAME VARCHAR2(20)
JOB VARCHAR2(10)
DEPTNO NUMBER(3)
SAL NUMBER(7,2)
Allow NULL for all columns except ename and job.

QUERY :

```
CREATE TABLE EMP
(
  "EMPNO" NUMBER(6),
  "ENAME" VARCHAR2(20) NOT NULL,
  "JOB" VARCHAR2(10) NOT NULL,
  "DEPTNO" NUMBER(3),
  "SAL" NUMBER(7,2)
);
DESCRIBE EMP;
```

OUTPUT :

```
Table created.
```

TABLE EMP

Column	Null?	Type
EMPNO	-	NUMBER(6,0)
ENAME	NOT NULL	VARCHAR2(20)
JOB	NOT NULL	VARCHAR2(10)
DEPTNO	-	NUMBER(3,0)
SAL	-	NUMBER(7,2)

Download CSV
5 rows selected.

Q2. Add column experience to the emp table. experience numeric null allowed.

QUERY :

```
10
11  ALTER TABLE EMP
12      ADD EXPERIENCE NUMBER(2);
13
14  DESCRIBE EMP;
```

OUTPUT :

TABLE EMP

Column	Null?	Type
EMPNO	-	NUMBER(6,0)
ENAME	NOT NULL	VARCHAR2(20)
JOB	NOT NULL	VARCHAR2(10)
DEPTNO	-	NUMBER(3,0)
SAL	-	NUMBER(7,2)
EXPERIENCE	-	NUMBER(2,0)

Download CSV
6 rows selected.

Q3. Modify the column width of the job field of the emp table.

QUERY :

```
16  ALTER TABLE EMP
17      MODIFY JOB VARCHAR2(20);
18
19  DESCRIBE EMP;
```

OUTPUT :

TABLE EMP

Column	Null?	Type
EMPNO	-	NUMBER(6,0)
ENAME	NOT NULL	VARCHAR2(20)
JOB	NOT NULL	VARCHAR2(20)
DEPTNO	-	NUMBER(3,0)
SAL	-	NUMBER(7,2)
EXPERIENCE	-	NUMBER(2,0)

Download CSV
6 rows selected.

Q4. Create dept table with the following structure.
Name Type
---------- ------
DEPTNO NUMBER (2)
DNAME VARCHAR2 (10)
LOC VARCHAR2 (10)
Deptno as the primary key.

QUERY :

```
1  CREATE TABLE dept
2  (
3      DEPTNO NUMBER(2),
4      DNAME VARCHAR2(10),
5      LOC VARCHAR2(10),
6      PRIMARY KEY (DEPTNO)
7  );
8
9  DESCRIBE dept;
```

OUTPUT :

TABLE DEPT

Column	Null?	Type
DEPTNO	NOT NULL	NUMBER(2,0)
DNAME	-	VARCHAR2(10)
LOC	-	VARCHAR2(10)

Download CSV
3 rows selected.

Q5. Create the emp1 table with ename and empno, add constraints to check the empno value while entering (i.e) empno > 100.

QUERY :

```
1  CREATE TABLE emp1
2  (
3    ename VARCHAR2(20),
4    empno NUMBER(6),
5    CHECK (empno > 100)
6  );
7
8  DESCRIBE emp1;
```

OUTPUT :

```
Table created.
```

TABLE EMP1

Column	Null?	Type
ENAME	-	VARCHAR2(20)
EMPNO	-	NUMBER(6,0)

Download CSV

2 rows selected.

Q6. Drop a column experience in the emp table.

QUERY :

```
1  ALTER TABLE EMP
2      DROP COLUMN EXPERIENCE;
3
4  DESCRIBE EMP;
```

OUTPUT :

```
Table altered.
```

TABLE EMP

Column	Null?	Type
EMPNO	-	NUMBER(6,0)
ENAME	NOT NULL	VARCHAR2(20)
JOB	NOT NULL	VARCHAR2(20)
DEPTNO	-	NUMBER(3,0)
SAL	-	NUMBER(7,2)

Download CSV

5 rows selected.

Q7. Truncate the emp table and drop the dept table.

QUERY :

```
1  TRUNCATE TABLE EMP;
2
3  DROP TABLE dept;
4
```

OUTPUT :

```
Table truncated.

Table dropped.
```

RESULT :

Hence successfully executed DDL queries in SQL.

EXERCISE – 2

DML (Data Manipulation Language)

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,..)

Example,

Consider a table **Student** with following fields.

S_id	S_Name	Age

INSERT into Student values (101,'Adam',15);

The above command will insert a record into **Student** table.

S_id	S_Name	Age
101	Adam	15

Example to Insert NULL value to a column

Example to Insert NULL Value to a column

Both the statements below will insert NULL value into **age** column of the Student table.

```
INSERT into Student(id,name) values(102,'Alex');
```

Or,

```
INSERT into Student values(102,'Alex',null);
```

The above command will insert only two column value other column is set to null.

S_id	S_Name	Age
101	Adam	15
102	Alex	

Example to Insert Default value to a column

```
INSERT into Student values(103,'Chris',default)
```

S_id	S_Name	Age
101	Adam	15
102	Alex	
103	Chris	14

Suppose the **age** column of student table has default value of 14.

Also, if you run the below query, it will insert default value into the age column, whatever the default value may be.

```
INSERT into Student values(103,'Chris')
```

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**;

Lets see an example,

update Student set age=18 where s_id=102;

S_id	S_Name	Age
101	Adam	15
102	Alex	18
103	Chris	14

Example to Update multiple columns

UPDATE Student set s_name='Abhi',age=17 where s_id=103;

The above command will update two columns of a record.

S_id	S_Name	Age
101	Adam	15
102	Alex	18
103	Abhi	17

DELETE COMMAND

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

DELETE from *table-name*;

Example to Delete all Records from a Table

DELETE from Student;

The above command will delete all the records from **Student** table.

Example to Delete a particular Record from a Table

Consider the following **Student** table

S_id	S_Name	Age
101	Adam	15
102	Alex	18
103	Abhi	17

DELETE from Student where s_id=103;

The above command will delete the record where s_id is 103 from **Student** table.

S_id	S_Name	Age
101	Adam	15
102	Alex	18

WHERE clause

Where clause is used to specify condition while retriving data from table. ***Where*** clause is used mostly with *Select*, *Update* and *Delete* query. If condititon 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]**;

Example using WHERE clause

Consider a **Student** table,

s_id	**s_Name**	**age**	**address**
101	Adam	15	Noida
102	Alex	18	Delhi
103	Abhi	17	Rohtak
104	Ankit	22	Panipat

Now we will use a SELECT statement to display data of the table, based on a condition, which we will add to the SELECT query using WHERE clause.

SELECT s_id, s_name, age, address

from Student **WHERE** s_id=101;

s_id	s_Name	age	address
101	Adam	15	Noida

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*;

Example for SELECT Query

Consider the following **Student** table,

S_id	S_Name	age	address
101	Adam	15	Noida
102	Alex	18	Delhi
103	Abhi	17	Rohtak
104	Ankit	22	Panipat

SELECT s_id, s_name, age from Student;

The above query will fetch information of s_id, s_name and age column from Student table

S_id	S_Name	age

101	Adam	15
102	Alex	18
103	Abhi	17
104	Ankit	22

Example to Select all Records from Table

A special character **asterisk** * is used to address all the data(belonging to all columns) in a query. *SELECT* statement uses * character to retrieve all records from a table.

```
SELECT * from student;
```

The above query will show all the records of Student table, that means it will show complete Student table as result.

S_id	**S_Name**	**age**	**address**
101	Adam	15	Noida
102	Alex	18	Delhi
103	Abhi	17	Rohtak
104	Ankit	22	Panipat

Example to Select particular Record based on Condition

SELECT * from Student **WHERE** s_name = 'Abhi';

103	Abhi	17	Rohtak

Example to Perform Simple Calculations using Select Query

Conside the following **Employee** table.

Eid	**Name**	**Age**	**salary**
101	Adam	26	5000
102	Ricky	42	8000
103	Abhi	22	10000
104	Rohan	35	5000

```
SELECT eid, name, salary+3000  from Employee;
```

The above command will display a new column in the result, showing 3000 added into existing salaries of the employees.

Eid	**Name**	**salary+3000**
101	Adam	8000
102	Ricky	11000
103	Abhi	13000
104	Rohan	8000

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 clause.

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

Example of LIKE clause

Consider the following **Student** table.

s_id	s_Name	age
101	Adam	15
102	Alex	18
103	Abhi	17

```
SELECT * from Student where s_name like 'A%';
```

The above query will return all records where **s_name** starts with character 'A'.

s_id	s_Name	age
101	Adam	15
102	Alex	18
103	Abhi	17

Example

```
SELECT * from Student where s_name like '_d%';
```

The above query will return all records from **Student** table where **s_name** contain 'd' as second character.

s_id	s_Name	age
101	Adam	15

Example

```
SELECT * from Student where s_name like '%x';
```

The above query will return all records from **Student** table where **s_name** contain 'x' as last character.

s_id	s_Name	age
102	Alex	18

Order By Clause

Order by clause is used with **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*;

Example using Order by

Consider the following **Emp** table,

eid	name	Age	salary

401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	10000
405	Tiger	35	8000

SELECT * from Emp **order by** salary;

The above query will return result in ascending order of the **salary**.

Eid	**name**	**Age**	**salary**
403	Rohan	34	6000
402	Shane	29	8000
405	Tiger	35	8000
401	Anu	22	9000
404	Scott	44	10000

Example of Order by DESC

Consider the **Emp** table described above,

SELECT * from Emp order by salary DESC;

The above query will return result in descending order of the **salary**.

Eid	**name**	**Age**	**salary**

404	Scott	44	10000
401	Anu	22	9000
405	Tiger	35	8000
402	Shane	29	8000
403	Rohan	34	6000

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
```

Example of Group by in a Statement

Consider the following **Emp** table.

Eid	**name**	**Age**	**salary**
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000

404	Scott	44	9000
405	Tiger	35	8000

Here we want to find name and age of employees grouped by their salaries

SQL query for the above requirement will be,

SELECT name, age from Emp **group by** salary;

Result will be,

Name	**age**
Rohan	34
Shane	29
Anu	22

Example of Group by in a Statement with WHERE clause

Consider the following **Emp** table

eid	**name**	**Age**	**salary**
401	Anu	22	9000
402	Shane	29	8000
403	Rohan	34	6000
404	Scott	44	9000

405	Tiger	35	8000

SQL query will be,

```
select name, salary from Emp where age > 25group by salary;
```

Result will be.

Name	**Salary**
Rohan	6000
Shane	8000
Scott	9000

You must remember that Group By clause will always come at the end, just like the Order by clause.

HAVING Clause

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

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
```

Example of HAVING Statement

Consider the following **Sale** table.

oid	**order_name**	**previous_balance**	**customer**
11	ord1	2000	Alex

12	ord2	1000	Adam
13	ord3	2000	Abhi
14	ord4	1000	Adam
15	ord5	2000	Alex

Suppose we want to find the customer whose previous_balance sum is more than 3000.

We will use the below SQL query,

```
SELECT * from sale group customer having sum(previous_balance) > 3000;
```

Result will be,

oid	order_name	previous_balance	customer
11	ord1	2000	Alex

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*;

Example

Consider the following **Emp** table.

Eid	name	Age	salary
401	Anu	22	5000

402	Shane	29	8000
403	Rohan	34	10000
404	Scott	44	10000
405	Tiger	35	8000

```
select distinct salary from Emp;
```

The above query will return only the unique salary from **Emp** table

salary
5000
8000
10000

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.

Example of AND

Consider the following **Emp** table

eid	**name**	**Age**	**salary**

401	Anu	22	5000
402	Shane	29	8000
403	Rohan	34	12000
404	Scott	44	10000
405	Tiger	35	9000

SELECT * from Emp WHERE salary < 10000 **AND** age > 25 ;

The above query will return records where salary is less than 10000 and age greater than 25.

eid	**name**	**Age**	**salary**
402	Shane	29	8000
405	Tiger	35	9000

OR operator

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

Example of OR

Consider the following **Emp** table

eid	**name**	**Age**	**Salary**

401	Anu	22	5000
402	Shane	29	8000
403	Rohan	34	12000
404	Scott	44	10000
405	Tiger	35	9000

SELECT * from Emp WHERE salary > 10000 **OR** age > 25 ;

The above query will return records where either salary is greater than 10000 or age greater than 25.

402	Shane	29	8000
403	Rohan	34	12000
404	Scott	44	10000
405	Tiger	35	9000

LAB-2 :

SQL - DATA MANIPULATION LANGUAGE

AIM : To execute DML commands and queries in SQL.

Q1. Create a table **"STUDENT"** and populate all values like below.

RegNo	Name	Gender	DOB	mobileno	City
312	Randheer	M	2000-12-20	8096735597	Rajahmundry
9531	Sarika	F	2015-09-15	9848035597	Rajahmundry
8088	Satya Vani	F	1986-12-31	9705710159	Rajahmundry
2609	Durga Rao	M	1973-09-26	9949028509	Rajahmundry
3001	Sanju	M	2002-01-30	9884792252	Rajahmundry
601	Varija Sri	F	1999-01-06	7674978787	Rajahmundry

QUERY :

```
CREATE TABLE STUDENT
(
    "RegNo" NUMBER(6),
    "Name" VARCHAR2(12),
    "Gender" CHAR(1),
    "DOB" DATE,
    "MobileNo" NUMBER(10),
    "City" VARCHAR2(12)
);

INSERT INTO
    STUDENT ("RegNo", "Name", "Gender", "DOB", "MobileNo", "City")
WITH input AS
(
    SELECT 312, 'Randheer', 'M', DATE '2000-12-20', 8096735597, 'Rajahmundry' FROM DUAL
    UNION ALL
    SELECT 9531, 'Sarika', 'F', DATE  '2015-09-15', 9848035597, 'Rajahmundry' FROM DUAL
    UNION ALL
    SELECT 8088, 'Satya Vani', 'F', DATE  '1986-12-31', 9705710159, 'Rajahmundry' FROM DUAL
    UNION ALL
    SELECT 2609, 'Durga Rao', 'M', DATE  '1973-09-26', 9949028509, 'Rajahmundry' FROM DUAL
    UNION ALL
    SELECT 3001, 'Sanju', 'M', DATE  '2002-01-30', 9884792252, 'Rajahmundry' FROM DUAL
    UNION ALL
    SELECT 601, 'Varija Sri', 'F', DATE  '1999-01-06', 7674978787, 'Rajahmundry' FROM DUAL
)SELECT * FROM input;

SELECT * FROM STUDENT;
```

OUTPUT :

Table created.

6 row(s) inserted.

RegNo	Name	Gender	DOB	MobileNo	City
312	Randheer	M	20-DEC-00	8096735597	Rajahmundry
9531	Sarika	F	15-SEP-15	9848035597	Rajahmundry
8088	Satya Vani	F	31-DEC-86	9705710159	Rajahmundry
2609	Durga Rao	M	26-SEP-73	9949028509	Rajahmundry
3001	Sanju	M	30-JAN-02	9884792252	Rajahmundry
601	Varija Sri	F	06-JAN-99	7674978787	Rajahmundry

Download CSV

6 rows selected.

Q2. Update the value of the student name whose register number is '312'.

QUERY :

```
1  UPDATE STUDENT
2  SET "Name" = 'Ram' where "RegNo" = 312;
3
4  SELECT * FROM STUDENT;
```

OUTPUT :

1 row(s) updated.

RegNo	Name	Gender	DOB	MobileNo	City
312	Ram	M	20-DEC-00	8096735597	Rajahmundry
9531	Sarika	F	15-SEP-15	9848035597	Rajahmundry
8088	Satya Vani	F	31-DEC-86	9705710159	Rajahmundry
2609	Durga Rao	M	26-SEP-73	9949028509	Rajahmundry
3001	Sanju	M	30-JAN-02	9884792252	Rajahmundry
601	Varija Sri	F	06-JAN-99	7674978787	Rajahmundry

Download CSV
6 rows selected.

Q3. Modify the date of birth for the student whose name is 'RAM' with a value '1983-05-01'.

QUERY :

```
1  UPDATE STUDENT
2  SET "DOB" = DATE '1983-05-01' where "Name" = 'Ram';
3
4  SELECT * FROM STUDENT;
```

OUTPUT :

1 row(s) updated.

RegNo	Name	Gender	DOB	MobileNo	City
312	Ram	M	01-MAY-83	8096735597	Rajahmundry
9531	Sarika	F	15-SEP-15	9848035597	Rajahmundry
8088	Satya Vani	F	31-DEC-86	9705710159	Rajahmundry
2609	Durga Rao	M	26-SEP-73	9949028509	Rajahmundry
3001	Sanju	M	30-JAN-02	9884792252	Rajahmundry
601	Varija Sri	F	06-JAN-99	7674978787	Rajahmundry

Download CSV
6 rows selected.

Q4. Create an employee table **"EMP"** and populate all values.

QUERY :

```
CREATE TABLE EMP
(
    "EMPID" NUMBER(6),
    "EMPName" VARCHAR2(20),
    "JOB" VARCHAR2(20),
    "SALARY" NUMBER(8,2),
    "MobileNo" NUMBER(10)
);

INSERT INTO
    EMP ("EMPID", "EMPName", "JOB", "SALARY", "MobileNo")
WITH input2 AS
(
    SELECT 4012, 'Ria', 'Assistant Professor', 48000.00, 8096735597 FROM DUAL
    UNION ALL
    SELECT 8970, 'Arjun', 'Professor', 95000.00, 9848035597 FROM DUAL
    UNION ALL
    SELECT 4578, 'Roy', 'Assistant Professor', 52000.00, 9705710159 FROM DUAL
    UNION ALL
    SELECT 1257, 'Shyam', 'Assistant Professor', 45000.00, 9949028509 FROM DUAL
    UNION ALL
    SELECT 6895, 'Vas', 'Professor', 105000.00, 9884792252 FROM DUAL
    UNION ALL
    SELECT 7895, 'Dev', 'Assistant Professor', 46500.00, 7674978787 FROM DUAL
)SELECT * FROM input2;

SELECT * FROM EMP;

```

OUTPUT :

```
Table created.


6 row(s) inserted.
```

EMPID	EMPName	JOB	SALARY	MobileNo
4012	Ria	Assistant Professor	48000	8096735597
8970	Arjun	Professor	95000	9848035597
4578	Roy	Assistant Professor	52000	9705710159
1257	Shyam	Assistant Professor	45000	9949028509
6895	Vas	Professor	105000	9884792252
7895	Dev	Assistant Professor	46500	7674978787

Download CSV

6 rows selected.

Q5. Update the emp table to set the salary of all employees to Rs. 15000/- who are working as Assistant professor.

QUERY :

```
1  UPDATE EMP
2  SET "SALARY" = 15000 WHERE "JOB" = 'Assistant Professor';
3
4  SELECT * FROM EMP;
```

OUTPUT :

```
4 row(s) updated.
```

EMPID	EMPName	JOB	SALARY	MobileNo
4012	Ria	Assistant Professor	15000	8096735597
8970	Arjun	Professor	95000	9848035597
4578	Roy	Assistant Professor	15000	9705710159
1257	Shyam	Assistant Professor	15000	9949028509
6895	Vas	Professor	105000	9884792252
7895	Dev	Assistant Professor	15000	7674978787

Download CSV

6 rows selected.

Q6. Create a pseudo table employee with the same structure as the table emp and insert rows into the table using select clauses.

QUERY :

```
1  CREATE TABLE Pseudo_EMP AS
2  SELECT * FROM EMP;
3
4  SELECT * FROM Pseudo_EMP;
```

OUTPUT :

Table created.

EMPID	EMPName	JOB	SALARY	MobileNo
4012	Ria	Assistant Professor	15000	8096735597
8970	Arjun	Professor	95000	9848035597
4578	Roy	Assistant Professor	15000	9705710159
1257	Shyam	Assistant Professor	15000	9949028509
6895	Vas	Professor	105000	9884792252
7895	Dev	Assistant Professor	15000	7674978787

Download CSV
6 rows selected.

Q7. Select employee name, job from the emp table.

QUERY :

```
1  SELECT "EMPName", "JOB" FROM EMP;
2
```

OUTPUT :

EMPName	JOB
Ria	Assistant Professor
Arjun	Professor
Roy	Assistant Professor
Shyam	Assistant Professor
Vas	Professor
Dev	Assistant Professor

Download CSV
6 rows selected.

RESULT :

Hence successfully executed the DML commands in SQL.

EXERCISE – 3

SQL DCL & TCL COMMANDS

AIM:

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

Data base 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.

Syntax,

GRANT privileges_names ON object TO user;

Example:

Create user first identified by passwd;

Grant select on customers to first;

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

Syntax,

REVOKE privileges ON object FROM user;

Example:

Revoke select on customers from first;

Transaction Control Language (TCL) Commands:

- **COMMIT:** Commits a Transaction.

Syntax:

COMMIT;

Example:

INSERT INTO customers

VALUES ('1006','2020-03-04',3200,'DL', '1008');

Commit;

Select * from customers;

- **ROLLBACK:** Rollbacks a transaction in case of any error occurs.

Syntax:

Rollback;

Example:

DELETE FROM customers

WHERE store_state = 'MH'

AND customer_id = '1002';

Select * from customers;

Rollback;

Select * from customers;

- **SAVEPOINT:**Sets a savepoint within a transaction.

Syntax:

SAVEPOINT SAVEPOINT_NAME;

This command is used only in the creation of SAVEPOINT among all the transactions.
In general ROLLBACK is used to undo a group of transactions.

Syntax for rolling back to Savepoint command:

ROLLBACK TO SAVEPOINT_NAME;

Example:

SAVEPOINT SP1;

DELETE FROM customers

WHERE store_state = 'MH'

AND customer_id = '1002';

SAVEPOINT SP2;

ROLLBACK TO SP1;

Select * from customers;

Result:

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

LAB-3 :

SQL - DATA CONTROL LANGUAGE
AND
SQL - TRANSACTION CONTROL LANGUAGE

AIM : To execute different DCL and TCL commands in SQL.

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

CREATE TABLE :

```
1  CREATE TABLE CUSTOMER
2  (
3      "customer_id" INTEGER,
4      "sale_date" DATE,
5      "sale_amount" NUMBER,
6      "sales_person" VARCHAR2(256),
7      "store_state" VARCHAR2(256),
8      "order_id" VARCHAR2(256)
9  );
10
11 DESC CUSTOMER;
12
```

```
Table created.
```

TABLE CUSTOMER

Column	Null?	Type
customer_id	-	NUMBER
sale_date	-	DATE
sale_amount	-	NUMBER
sales_person	-	VARCHAR2(256)
store_state	-	VARCHAR2(256)
order_id	-	VARCHAR2(256)

Download CSV

6 rows selected.

```sql
INSERT INTO CUSTOMER("customer_id", "sale_date", "sale_amount", "sales_person", "store_state", "order_id")
WITH input AS
(
    SELECT 1001, DATE '2020-05-23', 1200, 'Raj K', 'KA', '1001' FROM DUAL
    UNION ALL
    SELECT 1001, DATE '2020-05-22', 1200, 'M K', 'NULL', '1002' FROM DUAL
    UNION ALL
    SELECT 1002, DATE '2020-05-23', 1200, 'Malika Rakesh', 'MH', '1003' FROM DUAL
    UNION ALL
    SELECT 1003, DATE '2020-05-22', 1500, 'Malika Rakesh', 'MH', '1004' FROM DUAL
    UNION ALL
    SELECT 1004, DATE '2020-05-22', 1210, 'M K', 'NULL', '1003' FROM DUAL
    UNION ALL
    SELECT 1005, DATE '2019-12-12', 4200, 'R K Rakesh', 'MH', '1007' FROM DUAL
    UNION ALL
    SELECT 1002, DATE '2020-05-21', 1200, 'Molly Samberg', 'DL', '1001' FROM DUAL

)SELECT * FROM input;

SELECT * FROM CUSTOMER;

```

```
7 row(s) inserted.
```

customer_id	sale_date	sale_amount	sales_person	store_state	order_id
1001	23-MAY-20	1200	Raj K	KA	1001
1001	22-MAY-20	1200	M K	NULL	1002
1002	23-MAY-20	1200	Malika Rakesh	MH	1003
1003	22-MAY-20	1500	Malika Rakesh	MH	1004
1004	22-MAY-20	1210	M K	NULL	1003
1005	12-DEC-19	4200	R K Rakesh	MH	1007
1002	21-MAY-20	1200	Molly Samberg	DL	1001

```
Download CSV
7 rows selected.
```

QUERY-1 : GRANT

```
SQL> grant select on CUSTOMER to sudheer;
```

OUTPUT :

```
Grant succeeded.
```

QUERY-2 : REVOKE

```
SQL> revoke select on CUSTOMER from sudheer;
```

OUTPUT :

```
Revoke succeeded.
```

QUERY-3 : COMMIT

```
1  INSERT INTO CUSTOMER
2      VALUES ('1006','2020-03-04',3200,'DL', '1008');
3
4  COMMIT;
5
6  SELECT * FROM CUSTOMER;
```

OUTPUT :

```
Statement processed.
```

customer_id	sale_date	sale_amount	sales_person	store_state	order_id
1001	23-MAY-20	1200	Raj K	KA	1001
1001	22-MAY-20	1200	M K	-	1002
1002	23-MAY-20	1200	Malika Rakesh	MH	1003
1003	22-MAY-20	1500	Malika Rakesh	MH	1004
1004	22-MAY-20	1210	M K	-	1003
1005	12-DEC-19	4200	R K Rakesh	MH	1007
1002	21-MAY-20	1280	Molly Sanberg	DL	1001

```
Download CSV
7 rows selected.
```

QUERY-4 : ROLLBACK

```
1  DELETE FROM CUSTOMER WHERE "store_state"='MH' AND "customer_id"=1002;
2
3  SELECT * FROM CUSTOMER;
4
5  ROLLBACK;
6
7  SELECT * FROM CUSTOMER;
```

OUTPUT :

1 row(s) deleted.

customer_id	sale_date	sale_amount	sales_person	store_state	order_id
1001	23-MAY-20	1200	Raj K	KA	1001
1001	22-MAY-20	1200	M K	-	1002
1003	22-MAY-20	1500	Malika Rakesh	MH	1004
1004	22-MAY-20	1210	M K	-	1003
1005	12-DEC-19	4200	R K Rakesh	MH	1007
1002	21-MAY-20	1280	Molly Sanberg	DL	1001

Download CSV

6 rows selected.

Statement processed.

customer_id	sale_date	sale_amount	sales_person	store_state	order_id
1001	23-MAY-20	1200	Raj K	KA	1001
1001	22-MAY-20	1200	M K	-	1002
1002	23-MAY-20	1200	Malika Rakesh	MH	1003
1003	22-MAY-20	1500	Malika Rakesh	MH	1004
1004	22-MAY-20	1210	M K	-	1003
1005	12-DEC-19	4200	R K Rakesh	MH	1007
1002	21-MAY-20	1280	Molly Sanberg	DL	1001

Download CSV

7 rows selected.

QUERY-5 : SAVEPOINT

```
SQL> SAVEPOINT SP1;

Savepoint created.

SQL> DELETE FROM CUSTOMER WHERE "store_state"='MH' AND "customer_id"=1002;

1 row deleted.

SQL> SAVEPOINT SP2;
```

OUTPUT :

```
SQL> ROLLBACK TO SP1;

Rollback complete.
```

RESULT :

Hence successfully executed the DCL and TCL commands in SQL.

EXERCISE – 4

Built-In functions in SQL

Functions

Function 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..). Function can be classifies into **single row function 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 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 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-4 :

BUILT-IN FUNCTIONS IN SQL

AIM : To execute built-in functions in SQL.

1. Display all the details of the records whose employee name starts with 'S'.

QUERY :

```
1  SELECT * FROM EMP WHERE "EMPName" LIKE 'S%';
```

OUTPUT :

EMPID	EMPName	JOB	SALARY	MobileNo
1257	Shyam	Assistant Professor	45000	9949028509

Download CSV

2. Display all the details of the records whose employee name does not starts with 'S'.

QUERY :

```
1  SELECT * FROM EMP WHERE "EMPName" NOT LIKE 'S%';
```

OUTPUT :

EMPID	EMPName	JOB	SALARY	MobileNo
4012	Ria	Assistant Professor	48000	8096735597
8970	Arjun	Professor	95000	9848035597
4578	Roy	Assistant Professor	52000	9705710159
6895	Vas	Professor	105000	9884792252
7895	Dev	Assistant Professor	46500	7674978787

Download CSV

5 rows selected.

3. Display the rows whose empno ranges from 50000 to 100000.

QUERY :

```
1  SELECT * FROM EMP WHERE "SALARY" BETWEEN 50000 AND 100000;
```

OUTPUT :

EMPID	EMPName	JOB	SALARY	MobileNo
8970	Arjun	Professor	95000	9848035597
4578	Roy	Assistant Professor	52000	9705710159

Download CSV

2 rows selected.

4. Calculate the square root of the salary of all employees.

QUERY :

```
1  SELECT "EMPName", sqrt("SALARY") FROM EMP;
```

OUTPUT :

EMPName	SQRT("SALARY")
Ria	219.089023002066445382787913120320853581
Arjun	308.22070014844882251250961907271221 1262
Roy	228.035085019827595827209805113350895815
Shyam	212.132034355964257320253308631454711785
Vas	324.037034920393011548298371804399832885
Dev	215.638586528478246747339417990011847577

Download CSV

6 rows selected.

5. Count the total records in the emp table.

QUERY :

```
1  SELECT count(*) FROM EMP;
```

OUTPUT :

COUNT(*)
6

Download CSV

6. Calculate the total and average salary amount of the emptable.

QUERY :

```
1  SELECT sum(SALARY), avg(SALARY) FROM EMP;
```

OUTPUT :

SUM(SALARY)	AVG(SALARY)
391500	65250

Download CSV

7. Determine the max and min salary and rename the column as max_salary and min_salary.

QUERY :

```
1  SELECT max(SALARY) AS max_salary, min(SALARY) AS min_salary FROM EMP;
```

OUTPUT :

MAX_SALARY	MIN_SALARY
105000	45000

Download CSV

8. Display total salary spent for employees.

QUERY :

```
1  SELECT sum(SALARY) AS total_salary FROM EMP;
```

OUTPUT :

TOTAL_SALARY
391500

Download CSV

9. Display the date 60 days before the current date.

QUERY :

```
1  SELECT add_months(sysdate, -2) FROM DUAL;
```

OUTPUT :

ADD_MONTHS(SYSDATE,-2)
08-DEC-21

Download CSV

10. Display lowest paid employee details under each job level.

QUERY :

```
1  SELECT JOB, min(SALARY) FROM EMP GROUP BY JOB;
```

OUTPUT :

JOB	MIN(SALARY)
Assistant Professor	45000
Professor	95000

Download CSV

2 rows selected.

RESULT :

Hence successfully, executed built-in functions in SQL.

EXERCISE – 5

ER DIAGRAM

LAB-5 :

CONSTRUCTION OF ER DIAGRAM

AIM : To construct an Entity Relationship diagram for Car Rental Management System (DBMS Mini Project).

TOOLS : Online Visual Paradigm

PROCEDURE :

1. Mention entities involved - **USER**, **CUSTOMER**, **AGENCY**, **CAR**, **RENT**. Represent all the entities using the symbol rectangle.

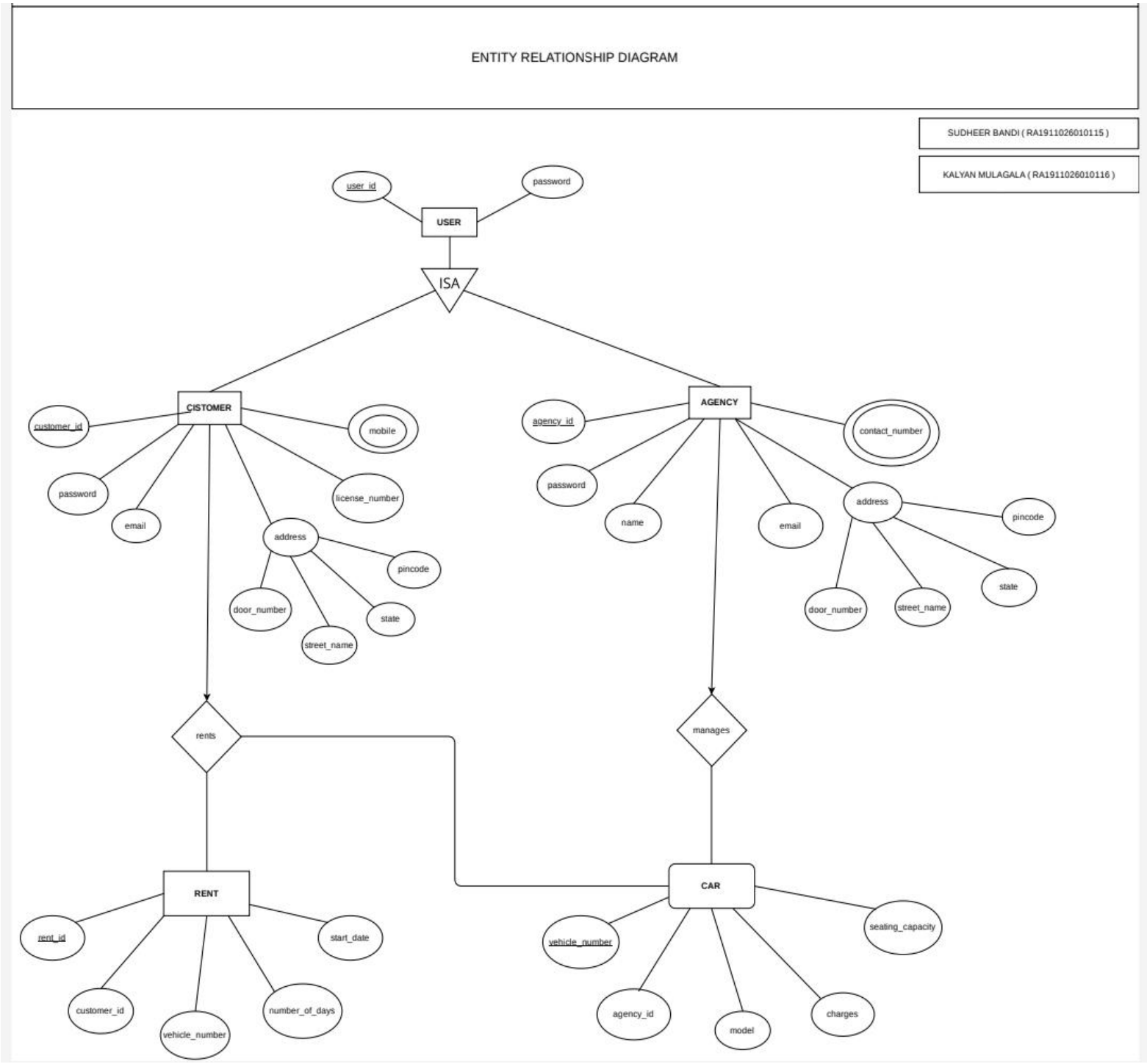
2. For each entity, all its attributes are drawn inside an eclipse.
    - **Multivalued attributes** are represented using double eclipse. - mobile, contact_number
    - **Composite attributes** are represented using eclipse of eclipses. - address
    - A **Primary Key** is represented as underlined text inside an ellipse. user_id, customer_id, agency_id, rent_id, vehicle_number.

3. All relationship sets are represented by diamond symbols - rents, manages.

4. All the cardinality Relationships drawn are as follows:
    - In the one-to-many relationship a car is associated with at most one customer via relationship rents, a customer is associated with several (including 0) cars via relationship rents.
    - In the one-to-many relationship a car is associated with at most one agency via relationship manages, an agency is associated with several (including 0) cars via relationship manages.

5. There are no weak entities as every entity can be uniquely identified.

OUTPUT : CAR RENTAL MANAGEMENT SYSTEM ER DIAGRAM



RESULT :

Hence successfully constructed ER diagram for Car Rental Management System.

EXERCISE – 6

JOIN QUERIES

Join in SQL

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
CROSS JOIN
table-name2;
```

Example of Cross JOIN
The **class** table,

ID	NAME
1	abhi
2	adam
4	alex

The **class_info** table,

ID	Address
1	DELHI
2	MUMBAI
3	CHENNAI

Cross JOIN query will be,

```
SELECT *
 from class,
```

cross JOIN class_info;

The result table will look like,

ID	NAME	ID	Address
1	abhi	1	DELHI
2	adam	1	DELHI
4	alex	1	DELHI
1	abhi	2	MUMBAI
2	adam	2	MUMBAI
4	alex	2	MUMBAI
1	abhi	3	CHENNAI
2	adam	3	CHENNAI
4	alex	3	CHENNAI

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*
INNER JOIN
table-name2
WHERE table-name1.column-name = table-name2.column-name;

Example of Inner JOIN

The **class** table,

ID	NAME
1	abhi
2	adam
3	alex
4	anu

The **class_info** table,

ID	Address
1	DELHI
2	MUMBAI
3	CHENNAI

Inner JOIN query will be,

SELECT * from class, class_info where class.id = class_info.id;

The result table will look like,

ID	NAME	ID	Address
1	abhi	1	DELHI
2	adam	2	MUMBAI
3	alex	3	CHENNAI

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
table-name2;

Example of Natural JOIN

The **class** table,

ID	NAME
1	abhi
2	adam
3	alex
4	anu

The **class_info** table,

ID	Address
1	DELHI
2	MUMBAI
3	CHENNAI

Natural join query will be,

SELECT * from class NATURAL JOIN class_info;

The result table will look like,

ID	NAME	Address
1	Abhi	DELHI
2	Adam	MUMBAI
3	Alex	CHENNAI

In the above example, both the tables being joined have ID column(same name and same datatype), hence the records for which value of ID matches in both the tables will be the result of Natural Join of these two tables.

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
table-name2
on table-name1.column-name = table-name2.column-name;

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(+);

Example of Left Outer Join

The **class** table,

ID	NAME
1	abhi
2	adam
3	alex
4	anu
5	ashish

The **class_info** table,

ID	Address
1	DELHI
2	MUMBAI
3	CHENNAI
7	NOIDA
8	PANIPAT

Left Outer Join query will be,

SELECT * FROM class LEFT OUTER JOIN class_info ON (class.id=class_info.id);

The result table will look like,

ID	NAME	ID	Address
1	abhi	1	DELHI
2	adam	2	MUMBAI
3	alex	3	CHENNAI
4	anu	null	null
5	ashish	null	null

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
table-name2

on table-name1.column-name = table-name2.column-name;

Example of Right Outer Join
The **class** table,

ID	NAME
1	abhi
2	adam
3	alex
4	anu
5	ashish

The **class_info** table,

ID	Address
1	DELHI
2	MUMBAI
3	CHENNAI
7	NOIDA
8	PANIPAT

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,

ID	NAME	ID	Address
1	abhi	1	DELHI
2	adam	2	MUMBAI
3	alex	3	CHENNAI
null	null	7	NOIDA
null	null	8	PANIPAT

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
table-name2
on table-name1.column-name = table-name2.column-name;
Example of Full outer join is,
The **class** table,

ID	NAME
1	abhi
2	adam
3	alex
4	anu
5	ashish

The **class_info** table,

ID	Address
1	DELHI
2	MUMBAI
3	CHENNAI
7	NOIDA
8	PANIPAT

Full Outer Join query will be like,

```
SELECT * FROM class FULL OUTER JOIN class_info on (class.id=class_info.id);
```

The result table will look like,

ID	NAME	ID	Address
1	abhi	1	DELHI
2	adam	2	MUMBAI
3	alex	3	CHENNAI
4	anu	null	null
5	ashish	null	null
null	null	7	NOIDA
null	null	8	PANIPAT

LAB-6 :

JOIN QUERIES IN SQL

AIM : To execute join queries in SQL.

1. CREATE A TABLE ORDERS HAVING COLUMNS LIKE ORDER_ID, ORDER_NUMBER, PERSON_ID.

QUERY :

```
1  create table ORDERS
2  (
3      ORDER_ID number,
4      ORDER_NUMBER number,
5      PERSON_ID number
6  );
7
8  DESC ORDERS;
```

OUTPUT :

```
Table created.
```

TABLE ORDERS

Column	Null?	Type
ORDER_ID	-	NUMBER
ORDER_NUMBER	-	NUMBER
PERSON_ID	-	NUMBER

Download CSV

3 rows selected.

2. CREATE ANOTHER TABLE PERSON HAVING COLUMNS LIKE PERSON_ID, LASTNAME, FIRSTNAME, ADDRESS AND CITY.

QUERY :

```
create table PERSON
(
    PERSON_ID number,
    LASTNAME varchar(20),
    FIRSTNAME varchar(20),
    ADDRESS varchar(30),
    CITY varchar(20)
);

DESC PERSON;
```

OUTPUT :

```
Table created.
```

TABLE PERSON

Column	Null?	Type
PERSON_ID	-	NUMBER
LASTNAME	-	VARCHAR2(20)
FIRSTNAME	-	VARCHAR2(20)
ADDRESS	-	VARCHAR2(30)
CITY	-	VARCHAR2(20)

Download CSV

5 rows selected.

3. INSERT VALUES FOR THE ABOVE TABLES.

ORDERS

QUERY :

```
insert into ORDERS(ORDER_ID, ORDER_NUMBER, PERSON_ID)
with input as
    ( select 456, 2, 12 from dual
      union all
      select 843, 5, 192 from dual
      union all
      select 1686, 1, 245 from dual
      union all
      select 345, 2, 12 from dual
    )
select * from input;

select * from ORDERS;

```

OUTPUT :

4 row(s) inserted.

ORDER_ID	ORDER_NUMBER	PERSON_ID
456	2	12
843	5	192
1686	1	245
345	2	12

Download CSV

4 rows selected.

PERSON

QUERY :

```
insert into PERSON(PERSON_ID, LASTNAME, FIRSTNAME, ADDRESS, CITY)
with input2 as
    ( select 12, 'Konidela', 'Ramcharan', 'Telangana', 'Hyderabad'  from dual
      union all
      select 192, 'Raju', 'Prabhas', 'Andhra Pradesh', 'Rajahmundry' from dual
      union all
      select 245, 'Allu', 'Arjun', 'Tamil Nadu', 'Chennai' from dual
    )
select * from input2;

select * from PERSON;

```

OUTPUT :

3 row(s) inserted.

PERSON_ID	LASTNAME	FIRSTNAME	ADDRESS	CITY
12	Konidela	Ramcharan	Telangana	Hyderabad
192	Raju	Prabhas	Andhra Pradesh	Rajahmundry
245	Allu	Arjun	Tamil Nadu	Chennai

Download CSV

3 rows selected.

4. USE VARIOUS JOIN OPERATIONS ON ABOVE DATABASE.

CROSS JOIN :

QUERY :

```
1  select * from ORDERS cross join PERSON;
```

OUTPUT :

ORDER_ID	ORDER_NUMBER	PERSON_ID	PERSON_ID	LASTNAME	FIRSTNAME	ADDRESS	CITY
456	2	12	12	Konidela	Ramcharan	Telangana	Hyderabad
843	5	192	12	Konidela	Ramcharan	Telangana	Hyderabad
1686	1	245	12	Konidela	Ramcharan	Telangana	Hyderabad
345	2	12	12	Konidela	Ramcharan	Telangana	Hyderabad
456	2	12	192	Raju	Prabhas	Andhra Pradesh	Rajahmundry
843	5	192	192	Raju	Prabhas	Andhra Pradesh	Rajahmundry
1686	1	245	192	Raju	Prabhas	Andhra Pradesh	Rajahmundry
345	2	12	192	Raju	Prabhas	Andhra Pradesh	Rajahmundry
456	2	12	245	Allu	Arjun	Tamil Nadu	Chennai
843	5	192	245	Allu	Arjun	Tamil Nadu	Chennai
1686	1	245	245	Allu	Arjun	Tamil Nadu	Chennai
345	2	12	245	Allu	Arjun	Tamil Nadu	Chennai

Download CSV

12 rows selected.

INNER JOIN :

QUERY :

```
1  select * from ORDERs, PERSON where ORDERS.PERSON_ID = PERSON.PERSON_ID;
```

OUTPUT :

ORDER_ID	ORDER_NUMBER	PERSON_ID	PERSON_ID	LASTNAME	FIRSTNAME	ADDRESS	CITY
456	2	12	12	Konidela	Ramcharan	Telangana	Hyderabad
843	5	192	192	Raju	Prabhas	Andhra Pradesh	Rajahmundry
1686	1	245	245	Allu	Arjun	Tamil Nadu	Chennai
345	2	12	12	Konidela	Ramcharan	Telangana	Hyderabad

Download CSV

4 rows selected.

NATURAL JOIN :

QUERY :

```
1  select * from ORDERS natural join PERSON;
```

OUTPUT :

PERSON_ID	ORDER_ID	ORDER_NUMBER	LASTNAME	FIRSTNAME	ADDRESS	CITY
12	456	2	Konidela	Ramcharan	Telangana	Hyderabad
192	843	5	Raju	Prabhas	Andhra Pradesh	Rajahmundry
245	1686	1	Allu	Arjun	Tamil Nadu	Chennai
12	345	2	Konidela	Ramcharan	Telangana	Hyderabad

Download CSV

4 rows selected.

DELETE - To illustrate matchings and unmatchings in OUTER JOINS.

```
1  DELETE FROM ORDERS where ORDERS.ORDER_ID=843;
2
3  select * from ORDERS;
```

ORDER_ID	ORDER_NUMBER	PERSON_ID
456	2	12
1686	1	245
345	2	12

Download CSV

3 rows selected.

LEFT OUTER JOIN :

QUERY :

```
1  select * from ORDERS left outer join PERSON on (ORDERS.PERSON_ID = PERSON.PERSON_ID);
```

OUTPUT :

ORDER_ID	ORDER_NUMBER	PERSON_ID	PERSON_ID	LASTNAME	FIRSTNAME	ADDRESS	CITY
456	2	12	12	Konidela	Ramcharan	Telangana	Hyderabad
345	2	12	12	Konidela	Ramcharan	Telangana	Hyderabad
1686	1	245	245	Allu	Arjun	Tamil Nadu	Chennai

Download CSV

3 rows selected.

RIGHT OUTER JOIN :

QUERY :

```
1  select * from ORDERS right outer join PERSON on (ORDERS.PERSON_ID = PERSON.PERSON_ID);
```

OUTPUT :

ORDER_ID	ORDER_NUMBER	PERSON_ID	PERSON_ID	LASTNAME	FIRSTNAME	ADDRESS	CITY
456	2	12	12	Konidela	Ramcharan	Telangana	Hyderabad
1686	1	245	245	Allu	Arjun	Tamil Nadu	Chennai
345	2	12	12	Konidela	Ramcharan	Telangana	Hyderabad
-	-	-	192	Raju	Prabhas	Andhra Pradesh	Rajahmundry

Download CSV

4 rows selected.

FULL OUTER JOIN :

QUERY :

```
1  select * from ORDERS full outer join PERSON on (ORDERS.PERSON_ID = PERSON.PERSON_ID);
```

OUTPUT :

ORDER_ID	ORDER_NUMBER	PERSON_ID	PERSON_ID	LASTNAME	FIRSTNAME	ADDRESS	CITY
456	2	12	12	Konidela	Ramcharan	Telangana	Hyderabad
1686	1	245	245	Allu	Arjun	Tamil Nadu	Chennai
345	2	12	12	Konidela	Ramcharan	Telangana	Hyderabad
-	-	-	192	Raju	Prabhas	Andhra Pradesh	Rajahmundry

Download CSV

4 rows selected.

RESULT :

Hence, successfully executed JOIN queries in SQL.

EXERCISE- 7

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

SQL Subquery Example:
1) Usually, a subquery should return only one record, but sometimes it can also return multiple records when used with operators LIKE IN, NOT IN in the where clause. The query syntax would be like,

```
SELECT first_name, last_name, subject
FROM student_details
WHERE games NOT IN ('Cricket', 'Football');
```

Subquery output would be similar to:

first_name	last_name	subject
----------	----------	----------
Shekar	Gowda	Badminton
Priya	Chandra	Chess

2) Lets 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-Corelated 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 statements 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 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 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. user 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-7 :

SUBQUERIES IN SQL

AIM : To execute subqueries in SQL.

CREATE TABLES

```
1  CREATE TABLE EMP
2  (
3      emp_id INT,
4      first_name VARCHAR2(20),
5      last_name VARCHAR2(20),
6      job_id VARCHAR2(20),
7      mobile NUMBER(10),
8      salary NUMBER(10,2),
9      PRIMARY KEY (emp_id)
10 );
11
12 CREATE TABLE DEPT
13 (
14     dept_id INT,
15     dept_name VARCHAR2(20),
16     loc_id INT,
17     PRIMARY KEY (dept_id)
18 );
19
20 CREATE TABLE LOC
21 (
22     loc_id INT,
23     city VARCHAR2(20),
24     PRIMARY KEY (loc_id)
25 );
26
```

OUTPUT

```
Table created.

Table created.

Table created.
```

ADD FOREIGN KEYS

```
1  ALTER TABLE EMP
2  ADD FOREIGN KEY (dept_id) REFERENCES DEPT(dept_id);
3
4  ALTER TABLE DEPT
5  ADD FOREIGN KEY (loc_id) REFERENCES LOC(loc_id);
6
7  DESC EMP;
8
9  DESC DEPT;
10
11 DESC LOC;
12
```

OUTPUT

```
Table altered.


Table altered.
```

TABLE EMP

Column	Null?	Type
EMP_ID	NOT NULL	NUMBER
FIRST_NAME	-	VARCHAR2(20)
LAST_NAME	-	VARCHAR2(20)
JOB_ID	-	VARCHAR2(20)
MOBILE	-	NUMBER(10,0)
SALARY	-	NUMBER(10,2)
DEPT_ID	-	NUMBER

Download CSV

7 rows selected.

TABLE DEPT

Column	Null?	Type
DEPT_ID	NOT NULL	NUMBER
DEPT_NAME	-	VARCHAR2(20)
LOC_ID	-	NUMBER

Download CSV

3 rows selected.

TABLE LOC

Column	Null?	Type
LOC_ID	NOT NULL	NUMBER
CITY	-	VARCHAR2(20)

Download CSV

2 rows selected.

INSERT VALUES

```
INSERT INTO LOC
WITH input AS
(   SELECT 1450, 'Chennai' FROM DUAL
    UNION ALL
    SELECT 1500, 'Mumbai' FROM DUAL
    UNION ALL
    SELECT 1700, 'Hyderabad' FROM DUAL
    UNION ALL
    SELECT 452, 'Bengaluru' FROM DUAL
    UNION ALL
    SELECT 234, 'Delhi' FROM DUAL
) SELECT * FROM input;

SELECT * FROM LOC;
```

```
5 row(s) inserted.
```

LOC_ID	CITY
1450	Chennai
1500	Mumbai
1700	Hyderabad
452	Bengaluru
234	Delhi

Download CSV
5 rows selected.

```
INSERT INTO DEPT
WITH input2 AS
(   SELECT 3, 'Marketing', 1450 FROM DUAL
    UNION ALL
    SELECT 10, 'Software', 1700 FROM DUAL
    UNION ALL
    SELECT 24, 'Management', 1500 FROM DUAL
    UNION ALL
    SELECT 80, 'Sales', 452 FROM DUAL
    UNION ALL
    SELECT 124, 'Administration', 1700 FROM DUAL
) SELECT * FROM input2;

SELECT * FROM DEPT;
```

```
5 row(s) inserted.
```

DEPT_ID	DEPT_NAME	LOC_ID
3	Marketing	1450
10	Software	1700
24	Management	1500
80	Sales	452
124	Administration	1700

Download CSV
5 rows selected.

```
INSERT INTO EMP
WITH INPUT AS
(   SELECT 1, 'Anil', 'Ravipudi', 'SALES_MAN', 9678909686, 10000, 10 FROM DUAL
    UNION ALL
    SELECT 2, 'Siva', 'Koratala', 'HR', 9567890897, 30000, 80 FROM DUAL
    UNION ALL
    SELECT 5, 'Prasanth', 'Neel', 'IT', 9878907834, 40000, 24 FROM DUAL
    UNION ALL
    SELECT 7, 'Surender', 'Reddy', 'MANAGER', 8679856787, 50000, 3 FROM DUAL
    UNION ALL
    SELECT 10, 'Sreenu', 'Boyapati', 'MANAGER', 8675848483, 56000, 80 FROM DUAL
) SELECT * FROM input;


SELECT * FROM EMP;

```

```
5 row(s) inserted.
```

EMP_ID	FIRST_NAME	LAST_NAME	JOB_ID	MOBILE	SALARY	DEPT_ID
1	Anil	Ravipudi	SALES_MAN	9678909686	10000	10
2	Siva	Koratala	HR	9567890897	30000	80
5	Prasanth	Neel	IT	9878907834	40000	24
7	Surender	Reddy	MANAGER	8679856787	50000	3
10	Sreenu	Boyapati	MANAGER	8675848483	56000	80

Download CSV
5 rows selected.

1. Create employee, department and location tables and do the following:

a. Retrieve all employees who have a salary greater than the average salaries in department 80.

QUERY :

```
1  SELECT * FROM EMP WHERE salary > ( SELECT AVG(salary) FROM EMP ) AND dept_id = 80;
```

OUTPUT :

EMP_ID	FIRST_NAME	LAST_NAME	JOB_ID	MOBILE	SALARY	DEPT_ID
10	Sreenu	Boyapati	MANAGER	8675848483	56000	80

Download CSV

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."

QUERY :

```
1  SELECT emp_id, last_name FROM EMP WHERE dept_id  IN ( SELECT dept_id FROM EMP where last_name LIKE '%u%');
```

OUTPUT :

EMP_ID	LAST_NAME
1	Ravipudi

Download CSV

c. Retrieve the last name, department number, and job ID of all employees whose department location ID is 1700.

QUERY :

```
1  SELECT last_name, dept_id, job_id FROM EMP WHERE dept_id in ( SELECT dept_id from DEPT WHERE loc_id = 1700);
```

OUTPUT :

LAST_NAME	DEPT_ID	JOB_ID
Ravipudi	10	SALES_MAN

Download CSV

2. Create a product table and do the following:

```
1  CREATE TABLE PRODUCT
2  (
3      product_id INT,
4      product_name  VARCHAR2(20),
5      category_no INT,
6      price NUMBER(5, 2),
7      PRIMARY KEY (product_id)
8  );
9
10 DESC PRODUCT;
11
```

Table created.

TABLE PRODUCT

Column	Null?	Type
PRODUCT_ID	NOT NULL	NUMBER
PRODUCT_NAME	-	VARCHAR2(20)
CATEGORY_NO	-	NUMBER
PRICE	-	NUMBER(5,2)

Download CSV

4 rows selected.

```
1  INSERT INTO PRODUCT
2  WITH input AS
3  (   SELECT 1, 'Link Ocean', 45, 20 FROM DUAL
4      UNION ALL
5      SELECT 2, 'Classmate Bok', 80, 40 FROM DUAL
6      UNION ALL
7      SELECT 3, 'Camel Crayons', 56, 50 FROM DUAL
8      UNION ALL
9      SELECT 56, 'Apsara Book', 80, 40 FROM DUAL
10     UNION ALL
11     SELECT 64, 'Camel Stapler', 52, 50 FROM DUAL
12     UNION ALL
13     SELECT 67, 'Cello Pen', 60, 10 FROM DUAL
14 ) SELECT * FROM input;
15
16 SELECT * FROM PRODUCT;
17
```

```
6 row(s) inserted.
```

PRODUCT_ID	PRODUCT_NAME	CATEGORY_NO	PRICE
1	Link Ocean	45	20
2	Classmate Bok	80	40
3	Camel Crayons	56	50
56	Apsara Book	80	40
64	Camel Stapler	52	50
67	Cello Pen	60	10

Download CSV

6 rows selected.

a. Retrieve the products whose category number is the same as that of product 64.

QUERY :

```
1  SELECT * FROM PRODUCT WHERE category_no IN ( SELECT category_no FROM PRODUCT WHERE product_id=64);
```

OUTPUT :

PRODUCT_ID	PRODUCT_NAME	CATEGORY_NO	PRICE
64	Camel Stapler	52	50

Download CSV

b. Retrieve all products that cost more than the average price in category no. 60.

QUERY :

```
1  SELECT * FROM PRODUCT WHERE price > ( SELECT AVG(price) FROM PRODUCT WHERE category_no = 60 );
2
```

OUTPUT :

PRODUCT_ID	PRODUCT_NAME	CATEGORY_NO	PRICE
1	Link Ocean	45	20
2	Classmate Bok	80	40
3	Camel Crayons	56	50
56	Apsara Book	80	40
64	Camel Stapler	52	50

Download CSV
5 rows selected.

c. Retrieve all products whose price is equal to one of the prices of products in category 80.

QUERY :

```
1  SELECT * FROM PRODUCT WHERE price IN ( SELECT price FROM PRODUCT WHERE category_no = 80);
```

OUTPUT :

PRODUCT_ID	PRODUCT_NAME	CATEGORY_NO	PRICE
2	Classmate Bok	80	40
56	Apsara Book	80	40

Download CSV
2 rows selected.

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

QUERY :

```
1  SELECT * FROM PRODUCT WHERE price > ( SELECT MAX(price) FROM PRODUCT WHERE category_no = 80);
```

OUTPUT :

PRODUCT_ID	PRODUCT_NAME	CATEGORY_NO	PRICE
3	Camel Crayons	56	50
64	Camel Stapler	52	50

Download CSV

2 rows selected.

RESULT :

Hence successfully executed SUBQUERIES in SQL

EXERCISE – 8
SET OPERATIONS and VIEWS

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

· **Union**
· **Union all**
· **Intersect**
· **Minus**

Rule:

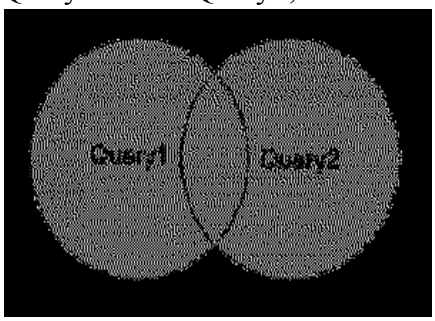
The queries which are related by the set operators should have a same number of column and column definition.

Union:

Returns all distinct rows selected by both the queries

Syntax:

Query1 Union Query2;



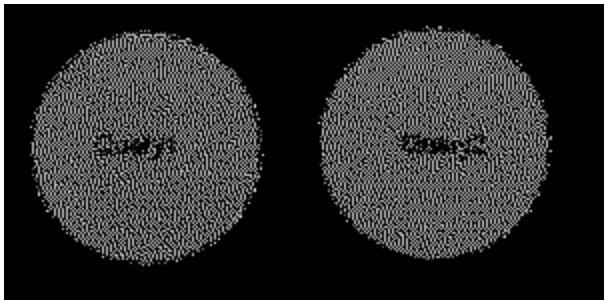
Exp: SELECT * FROM table1 UNION SELECT * FROM table2;

Union all:

Returns all rows selected by either query including the duplicates.

Syntax:

Query1 Union all Query2;



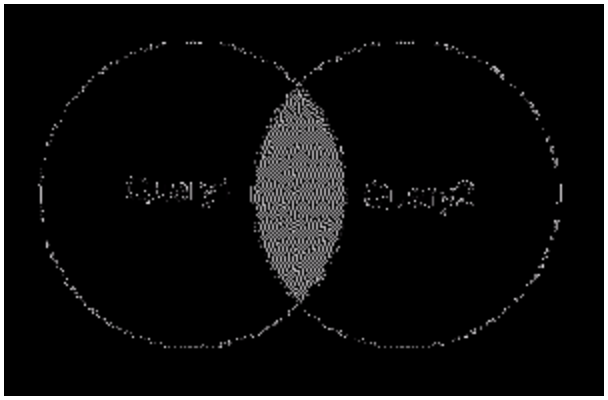
Exp: SELECT * FROM table1 UNION ALL SELECT * FROM table2;

Intersect

Returns rows selected that are common to both queries.

Syntax:

Query1 Intersect Query2;



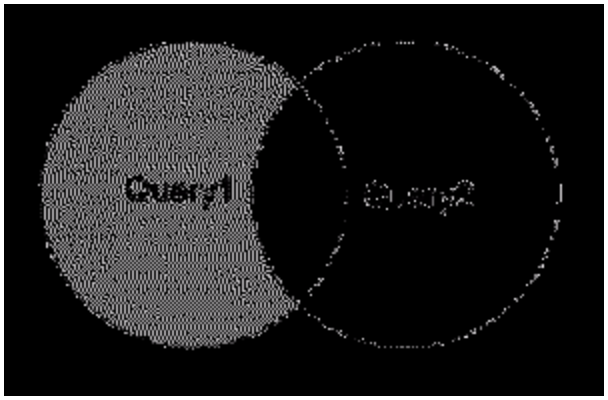
Exp: SELECT * FROM table1 INTERSECT SELECT * FROM table2;

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>;

Example:
Create or replace view empview as select * from emp;
Drop view empview;

SQL> select * from emp;
EMPNO ENAME JOB DEPTNO SAL
---------- ---------- ---------- ---------- ----------
1 Mathi AP 1 10000
2 Arjun ASP 2 12000
3 Gugan ASP 2 20000
4 Karthik AP 1 15000

LAB-8 :

SET OPERATORS AND VIEWS

AIM : To execute queries using set operators and implement views.

A. SET OPERATORS

Create employee tables and perform all set operations (UNION, INTERSECTION, UNIONALL AND MINUS).

```
1  CREATE TABLE EMP1
2  (
3      emp_id INT,
4      emp_name VARCHAR2(20),
5      salary NUMBER(6,2),
6      mobile NUMBER(10)
7  );
8
9  INSERT INTO EMP1
10 WITH input AS
11 (   SELECT 1, 'Arjun', 1000, 9234567890 FROM DUAL
12     UNION ALL
13     SELECT 2, 'Ajay', 2000, 9789078960 FROM DUAL
14     UNION ALL
15     SELECT 3, 'Surya', 1500, 8679567890 FROM DUAL
16 ) SELECT * FROM input;
17
18 SELECT * FROM EMP1;
```

```
Table dropped.

Table created.

3 row(s) inserted.
```

EMP_ID	EMP_NAME	SALARY	MOBILE
1	Arjun	1000	9234567890
2	Ajay	2000	9789078960
3	Surya	1500	8679567890

Download CSV

3 rows selected.

```sql
CREATE TABLE EMP2
(
    emp_id INT,
    emp_name VARCHAR2(20),
    salary NUMBER(6,2),
    mobile NUMBER(10)
);

INSERT INTO EMP2
WITH input AS
(   SELECT 1, 'Arjun', 1000, 9234567890 FROM DUAL
    UNION ALL
    SELECT 2, 'Ajay', 2000, 9789078960 FROM DUAL
    UNION ALL
    SELECT 3, 'Surya', 1500, 8679567890 FROM DUAL
    UNION ALL
    SELECT 4, 'Bindu', 3500, 9080978968 FROM DUAL
    UNION ALL
    SELECT 5, 'Vijay', 2400, 8067890784 FROM DUAL
) SELECT * FROM input;

SELECT * FROM EMP2;
```

```
Table created.


5 row(s) inserted.
```

EMP_ID	EMP_NAME	SALARY	MOBILE
1	Arjun	1000	9234567890
2	Ajay	2000	9789078960
3	Surya	1500	8679567890
4	Bindu	3500	9080978968
5	Vijay	2400	8067890784

Download CSV

5 rows selected.

UNION

QUERY :

```
1  SELECT * FROM EMP1 UNION SELECT * FROM EMP2;
```

OUTPUT :

EMP_ID	EMP_NAME	SALARY	MOBILE
1	Arjun	1000	9234567890
2	Ajay	2000	9789078960
3	Surya	1500	8679567890
4	Bindu	3500	9080978968
5	Vijay	2400	8067890784

Download CSV

5 rows selected.

UNION ALL

QUERY :

```
1  SELECT * FROM EMP1 UNION ALL SELECT * FROM EMP2;
```

OUTPUT :

EMP_ID	EMP_NAME	SALARY	MOBILE
1	Arjun	1000	9234567890
2	Ajay	2000	9789078960
3	Surya	1500	8679567890
1	Arjun	1000	9234567890
2	Ajay	2000	9789078960
3	Surya	1500	8679567890
4	Bindu	3500	9080978968
5	Vijay	2400	8067890784

Download CSV

8 rows selected.

INTERSECT

QUERY :

```
1  SELECT * FROM EMP1 INTERSECT SELECT * FROM EMP2;
```

OUTPUT :

EMP_ID	EMP_NAME	SALARY	MOBILE
1	Arjun	1000	9234567890
2	Ajay	2000	9789078960
3	Surya	1500	8679567890

Download CSV
3 rows selected.

MINUS

QUERY :

```
1  SELECT * FROM EMP2 MINUS SELECT * FROM EMP1;
```

OUTPUT :

EMP_ID	EMP_NAME	SALARY	MOBILE
4	Bindu	3500	9080978968
5	Vijay	2400	8067890784

Download CSV
2 rows selected.

1. UNION OR UNIONALL – Which is faster? Justify your answer once you get the output.

A. UNIONALL is faster. The UNION operator removes 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 you use along with Union/Unionall Operators to sort the result returned from the query?

A. ORDER BY

QUERY :

```
1  SELECT * FROM (SELECT * FROM EMP1 UNION SELECT * FROM EMP2) ORDER BY salary;
```

OUTPUT :

EMP_ID	EMP_NAME	SALARY	MOBILE
1	Arjun	1000	9234567890
3	Surya	1500	8679567890
2	Ajay	2000	9789078960
5	Vijay	2400	8067890784
4	Bindu	3500	9080978968

Download CSV

5 rows selected.

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

A. Intersect is an operator and Inner join is a type of join.
Intersect can return matching null values but inner join can't.

Intersect doesn't return any duplicate values but inner join returns

duplicate values if it's present in the tables.

QUERY :

```
1  SELECT * FROM EMP1 INTERSECT SELECT * FROM EMP2;
```

OUTPUT :

EMP_ID	EMP_NAME	SALARY	MOBILE
1	Arjun	1000	9234567890
2	Ajay	2000	9789078960
3	Surya	1500	8679567890

Download CSV

3 rows selected.

QUERY :

```
1  SELECT * FROM EMP1 INNER JOIN EMP2 ON EMP1.emp_id=EMP2.emp_id;
```

OUTPUT :

1 row(s) inserted.

EMP_ID	EMP_NAME	SALARY	MOBILE	EMP_ID	EMP_NAME	SALARY	MOBILE
1	Arjun	1000	9234567890	1	Arjun	1000	9234567890
2	Ajay	2000	9789078960	2	Ajay	2000	9789078960
3	Surya	1500	8679567890	3	Surya	1500	8679567890

Download CSV

3 rows selected.

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

A. The MINUS operator filters duplicate rows and return only DISTINCT rows from the left query that aren't in the right query's results, whereas NOT IN operator in Oracle does not filter the duplicates rows.

```
1  INSERT INTO EMP1
2  VALUES ( 6, 'Priya', 2400, 8090897657);
```

```
1 row(s) inserted.
```

QUERY :

```
1  SELECT * FROM EMP1 MINUS SELECT * FROM EMP2;
```

OUTPUT :

EMP_ID	EMP_NAME	SALARY	MOBILE
6	Priya	2400	8090897657

Download CSV

QUERY :

```
1  SELECT * FROM EMP1 WHERE emp_id NOT IN ( SELECT emp_id FROM EMP2);
```

OUTPUT :

EMP_ID	EMP_NAME	SALARY	MOBILE
6	Priya	2400	8090897657
6	Priya	2400	8090897657

Download CSV

2 rows selected.

B. VIEWS

Create a table named "STUDENT INFORMATION" which contains the following information.

Student Information Table :

Student_id varchar2(20)
Last_name varchar2(25)
First_name varchar2(20)
Dob DATE
Address varchar2(300)
City varchar2(20)
State varchar2(2)
ZipCode INT
Telephone INT
Fax INT
Email varchar2(100)

Department Information Table :

Department_Id INT primarykey
Department_Name varchar2(25)

```
1  CREATE TABLE STUDENT_INFO
2  (
3      student_id VARCHAR2(20),
4      first_name VARCHAR2(20),
5      last_name VARCHAR2(25),
6      dob DATE,
7      address VARCHAR2(300),
8      city VARCHAR2(20),
9      state VARCHAR2(20),
10     zipcode INT,
11     telephone INT,
12     fax INT,
13     email VARCHAR2(100)
14 );
15
16 DESC STUDENT_INFO;
```

Table created.

TABLE STUDENT_INFO

Column	Null?	Type
STUDENT_ID	-	VARCHAR2(20)
FIRST_NAME	-	VARCHAR2(20)
LAST_NAME	-	VARCHAR2(25)
DOB	-	DATE
ADDRESS	-	VARCHAR2(300)
CITY	-	VARCHAR2(20)
STATE	-	VARCHAR2(20)
ZIPCODE	-	NUMBER
TELEPHONE	-	NUMBER
FAX	-	NUMBER
EMAIL	-	VARCHAR2(100)

Download CSV

11 rows selected.

```
1  CREATE TABLE DEPARTMENT_INFO
2  (
3      dept_id INT PRIMARY KEY,
4      dept_name VARCHAR2(25)
5  );
6
7  DESC DEPARTMENT_INFO;
8
```

Table created.

TABLE DEPARTMENT_INFO

Column	Null?	Type
DEPT_ID	NOT NULL	NUMBER
DEPT_NAME	-	VARCHAR2(25)

Download CSV

2 rows selected.

```
1  INSERT INTO DEPARTMENT_INFO
2  WITH input AS
3  (
4      SELECT 1, 'CSE' FROM DUAL
5      UNION ALL
6      SELECT 2, 'EEE' FROM DUAL
7      UNION ALL
8      SELECT 3, 'MECH' FROM DUAL
9      UNION ALL
10     SELECT 4, 'ROBOTICS' FROM DUAL
11 ) SELECT * FROM input;
12
13 SELECT * FROM DEPARTMENT_INFO;
```

```
INSERT INTO STUDENT_INFO
WITH input AS
(
    SELECT 'RA60', 'Ajay', 'Gosh', DATE '1998-09-23', '3rd street, Sri Buildings', 'Chennai', 'Tamil Nadu', 603203, 9090909090, 4565676789, 'ajay@gamil.com', 4 FROM DUAL
    UNION ALL
    SELECT 'RA61', 'Surya', 'Pratap', DATE '1999-07-04', '567, 2nd street', 'Chennai', 'Tamil Nadu', 603201, 9898989898, 4578898967, 'surya@yahoo.com', 2 FROM DUAL
    UNION ALL
    SELECT 'RA62', 'Kiran', 'Raj', DATE '2000-03-11', '4th street, Balraji Buildings', 'Chennai', 'Tamil Nadu', 602340, 7897898787, 4567967890, 'kiran@yahoo.com', 2 FROM DUAL
    UNION ALL
    SELECT 'RA66', 'Priya', 'Sari', DATE '2000-11-03', '452, street_no-1, Homes Avenue', 'Chennai', 'Tamil Nadu', 604302, 6906909098, 4568980990, 'priya@google.com', 1 FROM DUAL
    UNION ALL
    SELECT 'RA77', 'Sindhu', 'Anantha', DATE '1999-04-08', '2nd street, Park Homes', 'Chennai', 'Tamil Nadu', 690567, 8798798787, 4567890872, 'sindhu@yahoo.com', 4 FROM DUAL
) SELECT * FROM input;

SELECT * FROM STUDENT_INFO;
```

```
4 row(s) inserted.
```

DEPT_ID	DEPT_NAME
1	CSE
2	EEE
3	MECH
4	ROBOTICS

Download CSV

4 rows selected.

```
5 row(s) inserted.
```

STUDENT_ID	FIRST_NAME	LAST_NAME	DOB	ADDRESS	CITY	STATE	ZIPCODE	TELEPHONE	FAX	EMAIL	DEPT_ID
RA60	Ajay	Gosh	23-SEP-98	3rd street, Sri Buildings	Chennai	Tamil Nadu	603203	9090909090	4565676789	ajay@gamil.com	4
RA61	Surya	Pratap	04-JUL-99	567, 2nd street	Chennai	Tamil Nadu	603201	9898989898	4578898967	surya@yahoo.com	2
RA62	Kiran	Raj	11-MAR-00	4th street, Balraji Buildings	Chennai	Tamil Nadu	602340	7897898787	4567967890	kiran@yahoo.com	2
RA66	Priya	Sari	03-NOV-00	452, street_no-1, Homes Avenue	Chennai	Tamil Nadu	604302	6906909098	4568980990	priya@google.com	1
RA77	Sindhu	Anantha	08-APR-99	2nd street, Park Homes	Chennai	Tamil Nadu	690567	8798798787	4567890872	sindhu@yahoo.com	4

Download CSV

5 rows selected.

(1) Create a view named student from student information and department information tables that contains only the following columns student_id, first name, last name and department_id.

QUERY :

```
1  CREATE VIEW STUDENT AS SELECT student_id, first_name, last_name, dept_id FROM STUDENT_INFO;
2
3  SELECT * FROM STUDENT;
```

OUTPUT :

View created.

STUDENT_ID	FIRST_NAME	LAST_NAME	DEPT_ID
RA60	Ajay	Gosh	4
RA61	Surya	Pratap	2
RA62	Kiran	Raj	2
RA66	Priya	Sari	1
RA77	Sindhu	Anantha	4

Download CSV

5 rows selected.

(2) Update the column of newly created view student. Observe the changes in the base tables.

QUERY :

```
1  UPDATE STUDENT
2      SET dept_id=1 WHERE student_id='RA62';
3
4  SELECT student_id, dept_id FROM STUDENT WHERE student_id='RA62';
5
6  SELECT student_id, dept_id FROM STUDENT_INFO WHERE student_id='RA62';
```

OUTPUT :

```
1 row(s) updated.
```

STUDENT_ID	DEPT_ID
RA62	1

Download CSV

STUDENT_ID	DEPT_ID
RA62	1

Download CSV

OBSERVATION :

Changes made to the virtual table also applied to main table.

(3) Delete the column newly created view student. Observe the changes in the base tables.

QUERY :

```
1  delete from student where first_name='Sindhu';
2
3  select * from student;
4
5  select * from student_info;
```

OUTPUT :

```
1 row(s) deleted.
```

STUDENT_ID	FIRST_NAME	LAST_NAME	DEPT_ID
RA60	Ajay	Gosh	4
RA61	Surya	Pratap	2
RA62	Kiran	Raj	1
RA66	Priya	Sari	1

Download CSV

4 rows selected.

STUDENT_ID	FIRST_NAME	LAST_NAME	DOB	ADDRESS	CITY	STATE	ZIPCODE	TELEPHONE	FAX	EMAIL	DEPT_ID
RA60	Ajay	Gosh	23-SEP-98	3rd street, Sri Buildings	Chennai	Tamil Nadu	603203	9090909090	4565676789	ajay@gamil.com	4
RA61	Surya	Pratap	04-JUL-99	567, 2nd street	Chennai	Tamil Nadu	603201	9898989898	4578898967	surya@yahoo.com	2
RA62	Kiran	Raj	11-MAR-00	4th street, Balraji Buildings	Chennai	Tamil Nadu	602340	7897898787	4567967890	kiran@yahoo.com	1
RA66	Priya	Sari	03-NOV-00	452, street_no-1, Homes Avenue	Chennai	Tamil Nadu	604302	6906909098	4568980990	priya@google.com	1

Download CSV

4 rows selected.

OBSERVATION :

The row which was deleted in the virtual table is also deleted in the base table.

RESULT :

Hence successfully executed operations using SET operators and implemented VIEWS.

EXERCISE – 9

PL/SQL BASIC 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:

Sequence of statements can be executed on satisfying certain condition . If statements are being used and differe forms 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;

Example:

Declare

A number:=10;

Begin

Loop

```
a := a+25;
exit when a=250; end loop;
dbms_output.put_line(to_char(a));
 end;
/
```

WHILE LOOP

Syntax

WHILE condition LOOP statement1; statement2;

END LOOP;

Example:

```
Declare
number:=0; j number:=0; begin

While i<=100 Loop j := j+i;
:= i+2;
end loop;
dbms_output.put_line('the value of j is' ||j); end;
/
```

FOR LOOP

Syntax:

FOR counter IN [REVERSE] LowerBound..UpperBound

LOOP
statement1;
statement2;
END LOOP;

Example:

Begin
For I in 1..2 Loop
Update emp set field = value where condition End loop;
End;
/

LAB-9 :

SIMPLE PL/SQL PROGRAMS

AIM : To execute simple PL/SQL programs.

1. Write a PL/SQL code to set the sales commission to 10%, if the sales revenue is greater than 200,000. Else, the sales commission is set to 5%.

CODE :

```
SQL> DECLARE
  2      sales_revenue NUMBER(8,2) := 100000;
  3      sales_commission NUMBER(8,2) := 0;
  4  BEGIN
  5      IF sales_revenue > 200000 THEN
  6          sales_commission := sales_revenue * 0.1;
  7      ELSE
  8          sales_commission := sales_revenue * 0.05;
  9      dbms_output.put_line('Sales Revenue    = ' || sales_revenue);
 10      dbms_output.put_line('Sales Commission = ' || sales_commission);
 11      END IF;
 12  END;
 13  /
```

OUTPUT :

```
Sales Revenue    = 100000
Sales Commission = 5000

PL/SQL procedure successfully completed.
```

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'.

CODE :

```
SQL> DECLARE
  2      monthly_value NUMBER := 6000;
  3      income_level VARCHAR(20);
  4  BEGIN
  5      CASE
  6          WHEN monthly_value <= 4000 THEN
  7              income_level := 'low income';
  8          WHEN monthly_value <= 5000 THEN
  9              income_level := 'average income';
 10          ELSE
 11              income_level := 'high income';
 12      END CASE;
 13      dbms_output.put_line('Monthly salary = ' || monthly_value);
 14      dbms_output.put_line('Income level   = ' || income_level);
 15  END;
 16  /
```

OUTPUT :

```
Monthly salary = 6000
Income level   = high income

PL/SQL procedure successfully completed.
```

3. Write a PL/SQL program to add numbers in a given range.

CODE :

```
SQL> DECLARE
  2      n INT := &n;
  3      sum1 INT := 0;
  4      i INT;
  5  BEGIN
  6      FOR i in 1..n
  7          LOOP
  8              sum1 := sum1 + i;
  9          END LOOP;
 10      dbms_output.put_line('Sum = ' || sum1);
 11  END;
 12  /
```

OUTPUT :

```
Enter value for n: 10
old   2:      n INT := &n;
new   2:      n INT := 10;
Sum = 55

PL/SQL procedure successfully completed.
```

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.

CREATE TABLE :

```
SQL> create table CUSTOMER
  2  (
  3       customer_id number PRIMARY KEY,
  4       customer_name varchar(20),
  5       address varchar(30),
  6       mobile INTEGER
  7  );
```

```
SQL> INSERT INTO CUSTOMER(customer_id, customer_name, address, mobile)
  2  WITH input AS
  3      ( SELECT 1, 'Ramcharan', 'Telangana', 9090897867  FROM DUAL
  4        UNION ALL
  5        SELECT 2, 'Prabhas', 'Andhra Pradesh', 8768757890 FROM DUAL
  6        UNION ALL
  7        SELECT 3, 'Allu Arjun', 'Tamil Nadu', 8659090897 FROM DUAL
  8        UNION ALL
  9        SELECT 4, 'Yash', 'Tamil Nadu', 8659090897 FROM DUAL
 10      )
 11  SELECT * FROM input;
```

```
SQL> SELECT * FROM CUSTOMER;
```

```
Table created.
```

```
4 rows created.
```

```
CUSTOMER_ID CUSTOMER_NAME        ADDRESS                         MOBILE
----------- -------------------- ------------------------------ ----------
          1 Ramcharan            Telangana                      9090897867
          2 Prabhas              Andhra Pradesh                 8768757890
          3 Allu Arjun           Tamil Nadu                     8659090897
          4 Yash                 Tamil Nadu                     8659090897
```

CODE :

```
SQL> DECLARE
  2      id CUSTOMER.customer_id%TYPE;
  3      name CUSTOMER.customer_name%TYPE;
  4  BEGIN
  5      id := &id;
  6      SELECT customer_name INTO name FROM CUSTOMER WHERE customer_id = id;
  7      dbms_output.put_line('Name of the customer with id = ' || id || ' is ' || name || '.');
  8  END;
  9  /
```

OUTPUT :

```
Enter value for id: 2
old   5:      id := &id;
new   5:      id := 2;
Name of the customer with id = 2 is Prabhas.

PL/SQL procedure successfully completed.
```

RESULT :

Hence successfully executed simple PL/SQL programs.

EXERCISE – 10

PROCEDURES IN PL/SQL

A **subprogram** is a program unit/module 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 the **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** This is again an optional part. It contains the code that handles run-time errors.

Creating a Procedure

A procedure is created with the **CREATE OR REPLACE PROCEDURE**statement. 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.

Example

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
   dbms_output.put_line('Hello World!');
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result −

```
Procedure created.
```

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

The above procedure named **'greetings'** can be called with the EXECUTE keyword as −

```
EXECUTE greetings;
```

The above call will display −

```
Hello World

PL/SQL procedure successfully completed.
```

The procedure can also be called from another PL/SQL block −

```
BEGIN
   greetings;
END;
/
```

The above call will display −

```
Hello World

PL/SQL procedure successfully completed.
```

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 an 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.**

IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

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

When the above code is executed at the SQL prompt, it produces the following result −

```
Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.
```

IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

```
DECLARE
   a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
  x := x * x;
END;
BEGIN
   a:= 23;
   squareNum(a);
   dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Square of (23): 529

PL/SQL procedure successfully completed.
```

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);
```

LAB-10 :

PROCEDURE

AIM : To implement procedure in PL/SQL code.

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.

```
Declare
        Salary number(5);
Begin
        Select sal into salary from emp where empno=7369;
        -- complete remaining statements


End;
/
```

CREATE TABLE :

```
CREATE TABLE EMP
(
    "emp_id" INT,
    "first_name" VARCHAR2(20),
    "last_name" VARCHAR2(20),
    "job_id" VARCHAR2(20),
    "mobile" NUMBER(10),
    "salary" NUMBER(10,2),
    "dept_id" INT,
    PRIMARY KEY ("emp_id")
);

INSERT INTO EMP
WITH INPUT AS
(   SELECT 12, 'Anil', 'Ravipudi', 'SALES_MAN', 9678909686, 10000, 10 FROM DUAL
    UNION ALL
    SELECT 2556, 'Siva', 'Koratala', 'HR', 9567890897, 30000, 80 FROM DUAL
    UNION ALL
    SELECT 2365, 'Prasanth', 'Neel', 'IT', 9878907834, 40000, 24 FROM DUAL
    UNION ALL
    SELECT 7678, 'Surender', 'Reddy', 'MANAGER', 8679856787, 50000, 3 FROM DUAL
    UNION ALL
    SELECT 7369, 'Sreenu', 'Boyapati', 'MANAGER', 8675848483, 56000, 80 FROM DUAL
) SELECT * FROM input;
```

```
select * from emp;
```

```
Table created.


5 row(s) inserted.
```

emp_id	first_name	last_name	job_id	mobile	salary	dept_id
12	Anil	Ravipudi	SALES_MAN	9678909686	10000	10
2556	Siva	Koratala	HR	9567890897	30000	80
2365	Prasanth	Neel	IT	9878907834	40000	24
7678	Surender	Reddy	MANAGER	8679856787	50000	3
7369	Sreenu	Boyapati	MANAGER	8675848483	56000	80

Download CSV

5 rows selected.

CODE :

```
CREATE PROCEDURE update_salary("empid" IN emp."emp_id"%TYPE)
AS
    "sal" emp."salary"%TYPE;
    "incr_per" NUMBER(3,2);
    "commission" NUMBER(3,2) := 0.10;
BEGIN
    select "salary" into "sal" from emp where "emp_id"="empid";
    IF "sal"<25000 THEN
        "incr_per" := 0.25;
    ELSIF "sal">=25000 AND "sal"<=50000 THEN
        "incr_per" := 0.20;
    ELSE
        "incr_per" := "commission";
    END IF;
    UPDATE emp
        SET "salary" = "sal" + "sal"*"incr_per" WHERE "emp_id" = "empid";
    dbms_output.put_line('Salary updated');
END;
```

```
DECLARE
    "empid" emp."emp_id"%type := 7369;
BEGIN
    UPDATE_SALARY("empid");
END;
```

```
select "salary" from emp where "emp_id"=7369;
```

OUTPUT :

```
Procedure created.
```

```
Procedure created.
Salary updated
```

salary
61600

Download CSV

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

```
Declare
        deptname dept.dname%type;
Begin       -- complete the block


End;
/
```

CREATE TABLE :

```
CREATE TABLE DEPT
(
    "dept_id" INT,
    "dept_name" VARCHAR2(20),
    "loc_id" INT,
    PRIMARY KEY ("dept_id")
);
```

```
INSERT INTO DEPT
WITH input AS
(   SELECT 3, 'Marketing', 1450 FROM DUAL
    UNION ALL
    SELECT 10, 'Software', 1700 FROM DUAL
    UNION ALL
    SELECT 24, 'Management', 1500 FROM DUAL
    UNION ALL
    SELECT 71, 'Sales', 452 FROM DUAL
    UNION ALL
    SELECT 48, 'Administration', 1700 FROM DUAL
) SELECT * FROM input;

SELECT * FROM DEPT;
```

```
Table created.


5 row(s) inserted.
```

DEPT_ID	DEPT_NAME	LOC_ID
3	Marketing	1450
10	Software	1700
24	Management	1500
71	Sales	452
48	Administration	1700

Download CSV

5 rows selected.

CODE :

```
CREATE PROCEDURE modify_dept_name("deptid" IN dept."dept_id"%type)
AS
    "deptname" dept."dept_name"%type;
BEGIN
    select "dept_name" into "deptname" from dept where "dept_id"="deptid";
    IF "deptname" != 'HRD' THEN
        UPDATE DEPT
            SET "dept_name"='HRD' WHERE "dept_id" = "deptid";
        dbms_output.put_line('Modified department name to HRD');
    ELSE
        dbms_output.put_line('Not Modified department name ( already HRD )');
    END IF;
END;
```

```
DECLARE
    "deptid" dept."dept_id"%type := 71;
BEGIN
    MODIFY_DEPT_NAME("deptid");
END;

```

```
SELECT "dept_name" FROM DEPT WHERE "dept_id"=71;
```

OUTPUT :

```
Procedure created.
```

```
Procedure created.
Modified department name to HRD
```

dept_name
HRD

Download CSV

RESULT :

Hence successfully implemented procedures in PL/SQL codes.

EXERCISE – 11

FUNCTIONS IN PL/SQL

A function is 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.

Example

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

We will use the CUSTOMERS table, which we had created in the PL/SQL Variables chapter −

```
Select * from customers;

+----+----------+-----+-----------+----------+
```

```
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
   total number(2) := 0;
BEGIN
   SELECT count(*) into total
   FROM customers;

   RETURN total;
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result −

```
Function created.
```

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.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function **totalCustomers** from an anonymous block −

```
DECLARE
   c number(2);
BEGIN
```

```
   c := totalCustomers();
   dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Total no. of Customers: 6

PL/SQL procedure successfully completed.
```

Example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
   a number;
   b number;
   c number;
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;
BEGIN
   a:= 23;
   b:= 45;
```

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

When the above code is executed at the SQL prompt, it produces the following result −

```
Maximum of (23,45): 45

PL/SQL procedure successfully completed.
```

PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as **recursion**.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as −

```
n! = n*(n-1)!
   = n*(n-1)*(n-2)!
      ...
   = n*(n-1)*(n-2)*(n-3)... 1
```

The following program calculates the factorial of a given number by calling itself recursively −

```
DECLARE
   num number;
   factorial number;

FUNCTION fact(x number)
RETURN number
IS
   f number;
BEGIN
   IF x=0 THEN
      f := 1;
   ELSE
```

```
      f := x * fact(x-1);
   END IF;
RETURN f;
END;

BEGIN
   num:= 6;
   factorial := fact(num);
   dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Factorial 6 is 720

PL/SQL procedure successfully completed.
```

LAB-11 :

FUNCTIONS

AIM : To implement functions in PL/SQL code.

1. Write a PL/SQL Function to find factorial of a given number.

CODE :

```
DECLARE
   num number;
   factorial number;

FUNCTION fact(x number)
RETURN number
IS
   f number;
BEGIN
   IF x=0 THEN
      f := 1;
   ELSE
      f := x * fact(x-1);
   END IF;
RETURN f;
END;

BEGIN
   num:= 8;
   factorial := fact(num);
   dbms_output.put_line('Factorial of '|| num || ' is ' || factorial);
END;
```

OUTPUT :

```
Statement processed.
Factorial of 8 is 40320
```

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

CODE :

```
DECLARE
   a number;
   b number;
   c number;
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;
BEGIN
   a:= 10;
   b:= 40;
   c := findMax(a, b);
   dbms_output.put_line('Maximum value of 10 and 40 : ' || c);
END;

```

OUTPUT :

```
Statement processed.
Maximum value of 10 and 40 : 40
```

RESULT :

Hence successfully implemented functions in PL/SQL code.

EXERCISE - 12

CURSORS

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.

Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example.

Example

We will be using the CUSTOMERS table we had created and used in the previous chapters.

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

```
DECLARE
   total_rows number(2);
BEGIN
   UPDATE customers
   SET salary = salary + 500;
```

```
   IF sql%notfound THEN
      dbms_output.put_line('no customers selected');
   ELSIF sql%found THEN
      total_rows := sql%rowcount;
      dbms_output.put_line( total_rows || ' customers selected ');
   END IF;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
6 customers selected

PL/SQL procedure successfully completed.
```

If you check the records in customers table, you will find that the rows have been updated −

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2500.00 |
|  2 | Khilan   |  25 | Delhi     |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2500.00 |
|  4 | Chaitali |  25 | Mumbai    |  7000.00 |
|  5 | Hardik   |  27 | Bhopal    |  9000.00 |
|  6 | Komal    |  22 | MP        |  5000.00 |
+----+----------+-----+-----------+----------+
```

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;
```

Example

Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
   c_id customers.id%type;
   c_name customerS.No.ame%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
```

```
BEGIN
   OPEN c_customers;
   LOOP
   FETCH c_customers into c_id, c_name, c_addr;
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
   CLOSE c_customers;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP

PL/SQL procedure successfully completed.
```

LAB-12 :

CURSOR

AIM : To implement cursors in PL/SQL code.

CREATE TABLE :

```
CREATE TABLE EMP
(
    emp_id INT,
    first_name VARCHAR2(20),
    last_name VARCHAR2(20),
    job_id VARCHAR2(20),
    mobile NUMBER(10),
    salary NUMBER(10,2),
    dept_id INT,
    PRIMARY KEY (emp_id)
);

INSERT INTO EMP
WITH INPUT AS
(   SELECT 12, 'Anil', 'Ravipudi', 'SALES_MAN', 9678909686, 10000, 10 FROM DUAL
    UNION ALL
    SELECT 2556, 'Siva', 'Koratala', 'HR', 9567890897, 30000, 20 FROM DUAL
    UNION ALL
    SELECT 2365, 'Prasanth', 'Neel', 'IT', 9878907834, 40000, 24 FROM DUAL
    UNION ALL
    SELECT 7678, 'Surender', 'Reddy', 'MANAGER', 8679856787, 50000, 3 FROM DUAL
    UNION ALL
    SELECT 7369, 'Sreenu', 'Boyapati', 'MANAGER', 8675848483, 56000, 20 FROM DUAL
) SELECT * FROM input;

SELECT * FROM EMP;
```

```
Table created.


5 row(s) inserted.
```

EMP_ID	FIRST_NAME	LAST_NAME	JOB_ID	MOBILE	SALARY	DEPT_ID
12	Anil	Ravipudi	SALES_MAN	9678909686	10000	10
2556	Siva	Koratala	HR	9567890897	30000	20
2365	Prasanth	Neel	IT	9878907834	40000	24
7678	Surender	Reddy	MANAGER	8679856787	50000	3
7369	Sreenu	Boyapati	MANAGER	8675848483	56000	20

Download CSV

5 rows selected.

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 total number of employee who got salary updated.

CODE :

```
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;
```

```
SELECT * FROM EMP;
```

OUTPUT :

```
Table created.
Salaries for 2 employees are updated
```

EMP_ID	FIRST_NAME	LAST_NAME	JOB_ID	MOBILE	SALARY	DEPT_ID
12	Anil	Ravipudi	SALES_MAN	9678909686	10000	10
2556	Siva	Koratala	HR	9567890897	34500	20
2365	Prasanth	Neel	IT	9878907834	40000	24
7678	Surender	Reddy	MANAGER	8679856787	50000	3
7369	Sreenu	Boyapati	MANAGER	8675848483	64400	20

Download CSV

5 rows selected.

2. Create a table emp_grade with columns empno & grade. Write PL/SQL block to insert values into the table emp_grade by processing the 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'.

CODE :

```
CREATE TABLE EMP_GRADE(emp_id INT, grade CHAR(1));

DECLARE
    CURSOR cur IS SELECT emp_id, salary FROM EMP;
    empid EMP.emp_id%TYPE;
    sal EMP.salary%TYPE;
BEGIN
    OPEN cur;
    IF cur%ISOPEN THEN
        LOOP
            FETCH cur INTO empid, sal;
            IF cur%NOTFOUND THEN
                EXIT;
            END IF;
            IF sal <= 20000 THEN
                INSERT INTO EMP_GRADE VALUES(empid,'C');
            ELSIF sal BETWEEN 20001 AND 40000 THEN
                INSERT INTO EMP_GRADE VALUES(empid, 'B');
            ELSE
                INSERT INTO EMP_GRADE VALUES(empid, 'A');
            END IF;
        END LOOP;
    ELSE
        OPEN cur;
    END IF;
END;
```

```
SELECT * FROM EMP_GRADE;
```

OUTPUT :

```
Table created.


Table created.
```

EMP_ID	GRADE
12	C
2556	B
2365	B
7678	A
7369	A

Download CSV

5 rows selected.

RESULT :

Hence successfully implemented cursors in PL/SQL code.

EXERCISE - 13

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.

Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters −

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff := :NEW.salary  - :OLD.salary;
   dbms_output.put_line('Old salary: ' || :OLD.salary);
   dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
```

```
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Trigger created.
```

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 create trigger, **display_salary_changes** will be fired and it will display the following result −

```
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 UPDATE 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 create trigger, **display_salary_changes** will be fired and it will display the following result −

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

LAB-13 :

TRIGGERS

AIM : To implement triggers in PL/SQL code.

```
SQL> CREATE TABLE EMP1
  2  (
  3       "emp_id" INT,
  4       "emp_name" VARCHAR2(20),
  5       "job_id" VARCHAR2(20),
  6       "mobile" NUMBER(10),
  7       "salary" NUMBER(10,2),
  8       "dob" DATE,
  9       "dept_id" INT,
 10       PRIMARY KEY ("emp_id")
 11  );

Table created.
```

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

CODE :

```
SQL> CREATE OR REPLACE TRIGGER age_validation
  2  BEFORE INSERT on EMP1
  3  FOR EACH ROW
  4
  5  DECLARE
  6  emp_age number;
  7
  8  BEGIN
  9      -- Finding employee age by date of birth
 10     SELECT MONTHS_BETWEEN(TO_DATE(sysdate,'DD-MON-YYYY'), TO_DATE(:new."dob",'DD-MON-YYYY'))/12
 11     INTO EMP_AGE FROM DUAL;
 12
 13     -- Check whether employee age is greater than 18 or not
 14      IF (EMP_AGE < 18) THEN
 15        RAISE_APPLICATION_ERROR(-20000,'Employee age must be greater than or equal to 18.');
 16      END IF;
 17
 18      -- Allow only past date of death
 19      IF(:new."dob" > sysdate) THEN
 20        RAISE_APPLICATION_ERROR(-20000,'Date of birth can not be Future date.');
 21      END IF;
 22  END;
 23  /
```

OUTPUT :

```
Trigger created.
```

```
SQL> INSERT INTO EMP1
  2      VALUES(12, 'Anil', 'SALES_MAN', 9678909686, 10000, DATE '2015-09-11', 10);
INSERT INTO EMP1
            *
ERROR at line 1:
ORA-20000: Employee age must be greater than or equal to 18.
ORA-06512: at "RA1911026010115.AGE_VALIDATION", line 11
ORA-04088: error during execution of trigger 'RA1911026010115.AGE_VALIDATION'
```

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

```
SQL> create table CUSTOMER
  2  (
  3      "customer_id" number PRIMARY KEY,
  4      "customer_name" varchar(20),
  5      "address" varchar(30),
  6      "mobile" INTEGER
  7  );

Table created.
```

CODE :

```
SQL> CREATE OR REPLACE TRIGGER customer_update
  2  BEFORE DELETE OR INSERT OR UPDATE ON customer
  3  FOR EACH ROW
  4  WHEN (NEW."customer_id" > 0)
  5
  6  BEGIN
  7      dbms_output.put_line('Changes to CUSTOMER table triggered');
  8  END;
  9  /
```

```
SQL> INSERT INTO CUSTOMER
  2  WITH input AS
  3      ( SELECT 1, 'Ramcharan', 'Telangana', 9090897867  FROM DUAL
  4        UNION ALL
  5        SELECT 2, 'Prabhas', 'Andhra Pradesh', 8768757890 FROM DUAL
  6        UNION ALL
  7        SELECT 3, 'Allu Arjun', 'Tamil Nadu', 8659090897 FROM DUAL
  8        UNION ALL
  9        SELECT 4, 'Yash', 'Tamil Nadu', 8659090897 FROM DUAL
 10      )
 11  SELECT * FROM input;
```

OUTPUT :

```
Trigger created.
```

```
Changes to CUSTOMER table triggered
Changes to CUSTOMER table triggered
Changes to CUSTOMER table triggered
Changes to CUSTOMER table triggered

4 rows created.
```

RESULT :

Hence successfully implemented triggers in PL/SQL code.

EXERCISE - 14

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;

1) QB 1 Handling **NO_DATA_FOUND** and **ZERO_DIVIDE** Exceptions

```
Declare
n1 number;
n2 number;
Begin
n2 := &n2;
Select sal into n1 from emp where empno=7654;
n1 := n1/n2;
Exception
when zero_divide then
dbms_output.put_line('Zero Divide Error !');
when no_data_found then
dbms_output.put_line('No such Row in EMP table');
when others then
```

dbms_output.put_line(‘Unknown exception’);

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;*

Q2) Write PL/SQL block to raise ‘out-of-balance’ exception if balance fall below 100.

DECLARE

out_of_balance EXCEPTION;

```
bal NUMBER;
BEGIN
IF bal < 100 THEN
RAISE out_of_stock;
END IF;
.
EXCEPTION
WHEN out_of_balance THEN
dbms_output.put_line(‘Low balance. Unable to do Transactions’);
END;
```

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');
….
```

LAB-14 :

EXCEPTION HANDLING

AIM : To implement exception handling in PL/SQL code.

CREATE TABLE :

```
SQL> create table CUSTOMER
  2  (
  3       customer_id number PRIMARY KEY,
  4       customer_name varchar(20),
  5       address varchar(30),
  6       mobile INTEGER
  7  );
```

```
SQL> INSERT INTO CUSTOMER(customer_id, customer_name, address, mobile)
  2  WITH input AS
  3      ( SELECT 1, 'Ramcharan', 'Telangana', 9090897867  FROM DUAL
  4        UNION ALL
  5        SELECT 2, 'Prabhas', 'Andhra Pradesh', 8768757890 FROM DUAL
  6        UNION ALL
  7        SELECT 3, 'Allu Arjun', 'Tamil Nadu', 8659090897 FROM DUAL
  8        UNION ALL
  9        SELECT 4, 'Yash', 'Tamil Nadu', 8659090897 FROM DUAL
 10      )
 11  SELECT * FROM input;
```

```
SQL> SELECT * FROM CUSTOMER;
```

```
Table created.
```

```
4 rows created.
```

```
CUSTOMER_ID CUSTOMER_NAME        ADDRESS                        MOBILE
----------- -------------------- ------------------------------ ----------
          1 Ramcharan            Telangana                      9090897867
          2 Prabhas              Andhra Pradesh                 8768757890
          3 Allu Arjun           Tamil Nadu                     8659090897
          4 Yash                 Tamil Nadu                     8659090897
```

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

CODE :

```
SQL> DECLARE
  2     c_id CUSTOMER.customer_id%type := &c_id;
  3     c_name CUSTOMER.customer_name%type;
  4
  5  BEGIN
  6      SELECT customer_name INTO  c_name FROM CUSTOMER WHERE customer_id = c_id;
  7      dbms_output.put_line('Name: '||  c_name);
  8      EXCEPTION
  9          WHEN no_data_found THEN
 10              dbms_output.put_line('No such customer!');
 11          WHEN others THEN
 12              dbms_output.put_line('Error!');
 13  END;
 14  /
```

OUTPUT :

```
Enter value for c_id: 4
old   2:     c_id CUSTOMER.customer_id%type := &c_id;
new   2:     c_id CUSTOMER.customer_id%type := 4;
Name: Yash

PL/SQL procedure successfully completed.
```

RESULT :

Hence successfully implemented exception handling in PL/SQL code.