shamshad ahmed

SQL

- SQL is a db language which was introduced by IBM.
- The initial name was "SEQUEL" later renamed as "SQL".
- It is used to communicate with database.
- SQL queries are not a case sensitive i.e we can write in upper / lower case characters.

  Ex:

  ```
  SELECT  * FROM EMP;----executed
  select * from emp; -----executed
  SeleCT * From Emp;----executed
  ```

- Every sql query should ends with a ";" .
- SQL is having the following five sub languages are:

1. Data Definition Language(DDL):

- > create
- > alter
  - > alter - modify
  - > alter - add
  - > alter - rename
  - > alter - drop
- > rename
- > truncate
- > drop

New Features:

- > recyclebin
- > flashback
- > purge

2. Data Manipulation Language(DML):

- > insert
- > update
- > delete

New Features:

- > insert all
- > merge

3. Data Retrieval / Query Language (DRL / DQL):

- > select

4. Transaction Control Language(TCL):

```
        > commit
        > rollback
        > savepoint

5. Data Control Language(DCL):
==========================
        > grant
        > revoke
=================================================================
create:
======
        - is used to create a new database object in oracle.
                Ex: table,view,synonym,sequence,.......etc

How to create a new table in oracle:
================================
syntax:
======
create table <table name>
(
<column name1> <datatype>[size],
<column name2> <datatype>[size],
.................................................
);

Ex:
create table student
(
STID INT,
SNAME CHAR(10),
SFEE NUMBER(6,2)
);

TO VIEW THE LIST OF TABLES IN ORACLE DB:
=======================================
SYNTAX:
========
SELECT * FROM TAB; (tab is system defined table)

TO VIEW THE STRUCTURE OF A TABLE:
================================
SYNTAX:
========
DESC <TABLE NAME>; (DESC - describe)

EX:
DESC STUDENT;

II) ALTER:
=========
        - to change the structure of a table.
        - sub commands of alter command.

i) alter - modify:
```

```
=============
       - to modify the datatype of a column and also change the size of
a datatype.
syntax:
=======
alter table <tn> modify <column name> <new datatype>[new size];

Ex:
SQL> ALTER TABLE STUDENT MODIFY SNAME VARCHAR2(5);

ii) alter - add:
============
       - to add a new column to a table.

syntax:
======
alter table <tn> add <new column name> <datatype>[size];

Ex:
SQL> ALTER TABLE STUDENT ADD SADDRESS VARCHAR2(20);

iii) alter - rename:
===============
       - to change a column name in a table.

syntax:
======
alter table <tn> rename <column> <old column name> to <new column name>;

Ex:
SQL> ALTER TABLE STUDENT RENAME COLUMN SNAME TO STUDENT_NAME;

iv) alter - drop:
=============
              - to drop / delete a column from a table.

syntax:
======
alter table <tn> drop <column> <column name>;

Ex:
SQL> ALTER TABLE STUDENT DROP COLUMN SFEE;

3) rename:
=========
       - to change a table name.

syntax:
=======
rename <old table name> to <new table name>;

Ex:
SQL> RENAME STUDENT TO SDETAILS;
SQL> RENAME SDETAILS TO STUDENT;
```

4) TRUNCATE:
=============
- to delete all rows but not columns from a table.
- deleting rows permanently.(i.e we cannot restored)
- cannot delete a specific row data from a table because it doesnot supports "where clause".

syntax:
=======
truncate table <table name>;

Ex:
SQL> TRUNCATE TABLE STUDENT;

5) DROP:
=======
- to delete a table (i.e rows & columns) from database.

syntax:
======
drop table <table name>;

EX:
SQL> DROP TABLE STUDENT;

NOTE:
=====
- before oracle10g enterprise edition once we drop a table from oracle db then it's permanently deleted whereas from oracle10g enterprise edition once we drop a table from oracle db then it's temporarly deleted.

Oracle10g enterprise edition new features:
==
i) recyclebin
ii) flashback
iii) purge

NOTE: these features are working under "user account" not in "system(admin) account".

i) Recyclebin:
============
- it is a system defined table in oracle.
- it can store the information about deleted tables.
- it is similar to "windows recyclebin" in computer.

How to view the structure of recyclebin table:
==
syntax:
=======
desc recyclebin;

EX:

=======
desc recyclebin;

How to view the information about deleted tables in recyclebin:
==========
syntax:
======
SQL> SELECT OBJECT_NAME,ORIGINAL_NAME FROM RECYCLEBIN;

OBJECT_NAME
ORIGINAL_NAME

BIN$KB/3u/nRQC2zjhL7ZDH5pQ==$0 STUDENT

ii) flashback:
==========
- this command is used to restore a table from recyclebin to database.

syntax:
======
flashback table <table name> to before drop;

Ex:
SQL> FLASHBACK TABLE STUDENT TO BEFORE DROP;

iii) purge:
========
- this command is used to delete a table from recyclebin permanently.
(or)
- this command is used to delete a table from database permanently.

To delete a specific table from recyclebin:
==========
syntax:
======
purge table <table name>;

Ex:
SQL> PURGE TABLE TEST2;

To delete all tables from recyclebin:
==========
syntax:
======
purge recyclebin;

EX:
SQL> PURGE RECYCLEBIN;

To delete a table from database permanently:

```
=====================================
syntax:
======
drop table <table name> purge;

EX:
SQL> DROP TABLE TEST3 PURGE;

2. Data Manipulation Language(DML):
================================
INSERT:
=======
       - to insert a new row data into a table.

method-1:
=========
insert into <table name> values(value1,value2,.........);

Ex:
SQL> CONN system/tiger
SQL> CREATE USER MYDB4PM IDENTIFIED BY MYDB4PM;
SQL> GRANT CONNECT,CREATE TABLE,UNLIMITED TABLESPACE TO MYDB4PM;

SQL> CONN MYDB4PM/MYDB4PM
SQL> CREATE TABLE STUDENT(STID INT,SNAME VARCHAR2(10),SFEE NUMBER(6,2));
SQL> INSERT INTO STUDENT VALUES(1,'SMITH',2500);

method-2:
========
insert into <table name>(required column
names)values(value1,value2,.....);

EX:
SQL> INSERT INTO STUDENT(STID,SNAME,SFEE)VALUES(2,'WARD',4200);
SQL> INSERT INTO STUDENT(STID,SNAME)VALUES(3,'JONES');
SQL> INSERT INTO STUDENT(STID)VALUES(4);
SQL> INSERT INTO STUDENT(SFEE,STID,SNAME) VALUES(3500,5,'ADAMS');

HOW TO INSERT MULTIPLE ROWS INTO A TABLE:
=========================================
       & : insert values into columns by dynamically.

method-1:
=========
insert into <tn> values(&<col1>,&<col2>,.....);

EX:
SQL> INSERT INTO STUDENT VALUES(&STID,'&SNAME',&SFEE);
Enter value for stid: 7
Enter value for sname: MILLER
Enter value for sfee: 3300

SQL> / ( RE-EXECUTE THE LAST EXECUTED SQL QUERY IN SQLPLUS)
.....................................
```

```
.....................................
.....................................

method-2:
=========
insert into <tn>(<COL1>,<COL2>,....) values(&<col1>,&<col2>,.....);

EX:
SQL> INSERT INTO STUDENT(STID)VALUES(&STID);
Enter value for stid: 10

SQL> /
.................................

HOW TO INSERT "NULLS" INTO A TABLE:
===================================
METHOD-1:
==========
SQL> INSERT INTO STUDENT VALUES(NULL,NULL,NULL);

METHOD-2:
=========
SQL> INSERT INTO STUDENT(STID,SNAME,SFEE)VALUES(NULL,NULL,NULL);

UPDATE:
========
       - to update all rows data in a table.
                          (or)
       - to update a specific row data in a table by using "where"
condition.

syntax:
======
update <tn> set <column name1>=<value1>,<column name2>=<value2>,......
.......... [ where <condition> ];

Ex:
SQL> UPDATE STUDENT SET
  2  STID=1021,
  3  SNAME='SCOTT',
  4  SFEE=8000
  5  WHERE STID=1;

Ex:
SQL> UPDATE STUDENT SET
  2  SNAME='SMITH',
  3  SFEE= 5000
  4  WHERE STID=7;

Ex:
SQL> UPDATE STUDENT SET
  2  SFEE=6200
  3  WHERE STID=11;
```

Ex:

```
SQL> UPDATE STUDENT SET
  2  STID=8
  3  SNAME='MILLER',
  4  SFEE=2200
  5  WHERE STID IS NULL;
```

EX:

```
SQL> UPDATE STUDENT SET
  2  SNAME='WARNER'
  3  WHERE SNAME IS NULL;
```

EX:

```
SQL> UPDATE STUDENT SET
  2  STID=NULL,
  3  SNAME=NULL,
  4  SFEE=NULL
  5  WHERE STID=6;
```

EX:

```
SQL> UPDATE STUDENT SET
  2  SNAME=NULL
  3  WHERE SNAME='SMITH';
```

EX:

```
SQL> UPDATE STUDENT SET SFEE=5000;
```

DELETE:
=======

- to delete all rows from a table at a time.
  (or)
- to delete a specific row from a table by using "where" condition.

syntax:
=======

```
delete from <table name> [where <condition>];
```

EX:

```
SQL> DELETE FROM STUDENT WHERE SNAME='WARNER';
SQL> DELETE FROM STUDENT WHERE SNAME IS NULL;
SQL> DELETE FROM STUDENT;
SQL> DELETE FROM STUDENT WHERE STID IS NULL;
```

DELETE vs TRUNCATE:
==========

DELETE	TRUNCATE
1. can delete a specific row.	1. no.
2. supporting "where" condition.	2. not supports.
3. temporary data deletion.	3. permanent data deletion.

4. can restore deleted data by using "rollback".

4. not possible.

5. execution speed is slow. (deleting rows in one by one)

5. execution speed is fast. (deleting all rows as a page)

3. Data Retrieval / Query Language (DRL / DQL):
==========
SELECT:
=======
- to retrieval all rows from a table.
(or)
- to retrieval a specific row from a table by using "where" condition.

syntax:
======
SELECT * / <LIST OF COLUMNS> FROM <TABLE NAME> [WHERE <CONDITION>];

Here, " * " ---------- all columns in a table.

EX:
```
SQL> SELECT * FROM EMP;
SQL> SELECT EMPNO,ENAME,SAL FROM EMP;
SQL> SELECT EMPNO,ENAME,SAL FROM EMP WHERE DEPTNO=20;
SQL> SELECT * FROM EMP WHERE JOB='MANAGER';
SQL> SELECT * FROM EMP WHERE EMPNO=7788;
SQL> SELECT * FROM EMP WHERE COMM IS NOT NULL;
SQL> SELECT * FROM EMP WHERE COMM IS NULL;
```

CONCATENATION OPERATOR(||):
==========
- to add / join two or more than two string expressions.

syntax:
======
<string1> || <string2> || <string3> || .....

EX:
```
SQL> SELECT 'Mr.'||ENAME||' '||'WORKING AS A'||' '||JOB FROM EMP;
```

OUTPUT:

```
Mr.SMITH WORKING AS A CLERK
```

ALIAS NAMES:
==========
- it is a temporary name / alternate name.
- created at two levels those are

i) column level alias names:
==========
- when we created alias name for columns then we called as column level alias name.

ii) table level alias names:
=====================
- when we created alias name for table then we called as a table level alias name.

syntax:
======
select <column name1> <column alias name1>,<column name2> <column alias name2>,.....from <tn> <table alias name>;

Ex:
SQL> SELECT DEPTNO X,DNAME Y,LOC Z FROM DEPT D;

DISTINCT KEYWORD:
==================
- to eliminate duplicate values from a column.

syntax:
======
distinct <column name>

Ex:
SQL> SELECT DISTINCT JOB FROM EMP;
SQL> SELECT DISTINCT DEPTNO FROM EMP;
SQL> SELECT DISTINCT DEPTNO FROM EMP ORDER BY DEPTNO;

PAGESIZE n:
==========
- to display no.of rows per a single page.
- by default a page can display 14 rows only.
- "n" ---- no.of rows.
- maximum rows upto 50000.

syntax:
=======
set pagesize n;

Ex:
set pagesize 100;

Lines n:
=======
- to assign no.of bytes to each line in sqlplus environment.
- by default each line is having 80 bytes.
- here "n" ----- no.of bytes.
- maximum size for a line is upto 32767 bytes.

syntax:
======
set lines n;

Ex:
set lines 160;

HOW TO CREATE A NEW TABLE FROM AN EXISTING TABLE(OLD TABLE):
==

SYNTAX FOR CREATED TABLE WITH COPY OF ALL ROWS & COLUMNS:
==========
SYNTAX:
========
CREATE TABLE <NEW TN> AS SELECT * FROM <OLD TN>;

EX:
SQL> CREATE TABLE NEWDEPT AS SELECT * FROM DEPT;
(OR)
SQL> CREATE TABLE NEWDEPT1 AS SELECT * FROM DEPT WHERE 1=1;

SYNTAX FOR CREATED TABLE WITH COPY OF COLUMNS BUT NOT ROWS:
==========
SYNTAX:
========
CREATE TABLE <NEW TN> AS SELECT * FROM <OLD TN>
WHERE <FALSE CONDITION>;

EX:
SQL> CREATE TABLE NEWDEPT2 AS SELECT * FROM DEPT WHERE 1=2;

CREATED TABLE WITH SPECIFIC ROWS:
==========
SYNTAX:
========
CREATE TABLE <NEW TN> AS SELECT * FROM <OLD TN>
WHERE <ROW CONDITION>;

EX:
CREATE TABLE NEWEMP AS SELECT * FROM EMP WHERE DEPTNO=20;

CREATED TABLE WITH SPECIFIC COLUMNS:
==========
SYNTAX:
========
CREATE TABLE <NEW TN> AS SELECT <LIST OF COLUMNS> FROM <OLD TN> ;

EX:
CREATE TABLE NEWEMP1 AS SELECT EMPNO,ENAME,SAL FROM EMP;

HOW COPY DATA FROM ONE TABLE TO ANOTHER TABLE:
==========
SYNTAX:
========
INSERT INTO <DESTINATION TN> SELECT * FROM <SOURCE TN>;

EX:
SQL> SELECT * FROM DEPT;
SQL> CREATE TABLE DUMMYDEPT AS SELECT * FROM DEPT WHERE 1=0;

SQL> SELECT * FROM DEPT;----------SOURCE TABLE
SQL> SELECT * FROM DUMMYDEPT;-------DESTINATION TABLE

SQL> INSERT INTO DUMMYDEPT SELECT * FROM DEPT;

```
INSERT ALL:
===========
       - TO INSERT ROWS INTO MULTIPLE TABLES BUT ROWS(DATA)
MUST BE AN EXISTING TABLE ONLY.

SYNTAX:
========
INSERT ALL INTO <TN1> VALUES(<COL1>,<COL2>,......)
INTO <TN2> VALUES(<COL1>,<COL2>,.........................)
.....................................................................
.................
.....................................................................
.................
INTO <TN n> VALUES(<COL1>,<COL2>,.......................)
SELECT * FROM <OLD TABLE NAME>;

EX:
CREATE THREE EMPTY TABLES:
==========================
SQL> CREATE TABLE TEST1 AS SELECT * FROM DEPT WHERE 1=0;
SQL> CREATE TABLE TEST2 AS SELECT * FROM DEPT WHERE 1=0;
SQL> CREATE TABLE TEST3 AS SELECT * FROM DEPT WHERE 1=0;

SQL> INSERT ALL INTO TEST1 VALUES(DEPTNO,DNAME,LOC)
  2  INTO TEST2 VALUES(DEPTNO,DNAME,LOC)
  3  INTO TEST3 VALUES(DEPTNO,DNAME,LOC)
  4  SELECT * FROM DEPT;

SQL> SELECT * FROM TEST1;
SQL> SELECT * FROM TEST2;
SQL> SELECT * FROM TEST3;
```

```
ORACLE DEMO TABLES:
====================
CREATE TABLE EMP
       (EMPNO NUMBER(4) NOT NULL,
        ENAME VARCHAR2(10),
        JOB VARCHAR2(9),
        MGR NUMBER(4),
        HIREDATE DATE,
        SAL NUMBER(7, 2),
        COMM NUMBER(7, 2),
        DEPTNO NUMBER(2));

INSERT INTO EMP VALUES
        (7369, 'SMITH',  'CLERK',     7902,
        TO_DATE('17-DEC-1980', 'DD-MON-YYYY'),  800, NULL, 20);
INSERT INTO EMP VALUES
        (7499, 'ALLEN',  'SALESMAN',  7698,
        TO_DATE('20-FEB-1981', 'DD-MON-YYYY'), 1600,  300, 30);
```

```
INSERT INTO EMP VALUES
        (7521, 'WARD',   'SALESMAN',  7698,
        TO_DATE('22-FEB-1981', 'DD-MON-YYYY'), 1250,  500, 30);
INSERT INTO EMP VALUES
        (7566, 'JONES',  'MANAGER',   7839,
        TO_DATE('2-APR-1981', 'DD-MON-YYYY'),  2975, NULL, 20);
INSERT INTO EMP VALUES
        (7654, 'MARTIN', 'SALESMAN',  7698,
        TO_DATE('28-SEP-1981', 'DD-MON-YYYY'), 1250, 1400, 30);
INSERT INTO EMP VALUES
        (7698, 'BLAKE',  'MANAGER',   7839,
        TO_DATE('1-MAY-1981', 'DD-MON-YYYY'),  2850, NULL, 30);
INSERT INTO EMP VALUES
        (7782, 'CLARK',  'MANAGER',   7839,
        TO_DATE('9-JUN-1981', 'DD-MON-YYYY'),  2450, NULL, 10);
INSERT INTO EMP VALUES
        (7788, 'SCOTT',  'ANALYST',   7566,
        TO_DATE('09-DEC-1982', 'DD-MON-YYYY'), 3000, NULL, 20);
INSERT INTO EMP VALUES
        (7839, 'KING',   'PRESIDENT', NULL,
        TO_DATE('17-NOV-1981', 'DD-MON-YYYY'), 5000, NULL, 10);
INSERT INTO EMP VALUES
        (7844, 'TURNER', 'SALESMAN',  7698,
        TO_DATE('8-SEP-1981', 'DD-MON-YYYY'),  1500,    0, 30);
INSERT INTO EMP VALUES
        (7876, 'ADAMS',  'CLERK',     7788,
        TO_DATE('12-JAN-1983', 'DD-MON-YYYY'), 1100, NULL, 20);
INSERT INTO EMP VALUES
        (7900, 'JAMES',  'CLERK',     7698,
        TO_DATE('3-DEC-1981', 'DD-MON-YYYY'),   950, NULL, 30);
INSERT INTO EMP VALUES
        (7902, 'FORD',   'ANALYST',   7566,
        TO_DATE('3-DEC-1981', 'DD-MON-YYYY'),  3000, NULL, 20);
INSERT INTO EMP VALUES
        (7934, 'MILLER', 'CLERK',     7782,
        TO_DATE('23-JAN-1982', 'DD-MON-YYYY'), 1300, NULL, 10);

CREATE TABLE DEPT
       (DEPTNO NUMBER(2),
        DNAME VARCHAR2(14),
        LOC VARCHAR2(13) );

INSERT INTO DEPT VALUES (10, 'ACCOUNTING', 'NEW YORK');
INSERT INTO DEPT VALUES (20, 'RESEARCH',   'DALLAS');
INSERT INTO DEPT VALUES (30, 'SALES',      'CHICAGO');
INSERT INTO DEPT VALUES (40, 'OPERATIONS', 'BOSTON');

CREATE TABLE SALGRADE
        (GRADE NUMBER,
         LOSAL NUMBER,
         HISAL NUMBER);

INSERT INTO SALGRADE VALUES (1,  700, 1200);
INSERT INTO SALGRADE VALUES (2, 1201, 1400);
```

```
INSERT INTO SALGRADE VALUES (3, 1401, 2000);
INSERT INTO SALGRADE VALUES (4, 2001, 3000);
INSERT INTO SALGRADE VALUES (5, 3001, 9999);
```

OPERATORS:
==========

- operator are used to perform some operation on the given operand values.oracle supports the following operators are:

i) Assignment operator => =
ii) Arithmetic operators => + , - ,* , /
iii) Relational operators => < , > , <= , >= ,!=(or)<>
iv) Logical operators => AND,OR,NOT
v) Set operators => UNION,UNION ALL, INTERSECT,MINUS

vi) Special operators =>

(+ve) operators	(-ve) operator
IN	NOT IN
BETWEEN	NOT BETWEEN
IS NULL	IS NOT NULL
LIKE	NOT LIKE

i) Assignment operator:
=======================

- to assign a values to varibale / attribute.

syntax:
======

<variable name / attribute name> <assignment operator> <value>

Ex:

```
SELECT * FROM EMP WHERE EMPNO=7788;
UPDATE EMP SET SAL=50000 WHERE DEPTNO=10;
DELETE FORM EMP WHERE JOB='CLERK';
```

ii) Arithmetic operators:
=========================

- to perform some mathematical operations like addition, subtraction,multiple and division.

syntax:
======

<column name> <arithmetic operator> <value>

Ex:

```
waq to display SQL> SELECT SAL OLD_SALARY,SAL+1000 NEW_SALARY
  2  FROM EMP;

OLD_SALARY     NEW_SALARY
-------------------  ----------------------
       800            1800
```

Ex:
waq to display empno,ename,salary and annual salary of employees
who are working under deptno=10;

```
SQL> SELECT EMPNO,ENAME,SAL BASIC_SALARY,
  2  SAL*12 ANNUAL_SALARY FROM EMP
  3  WHERE DEPTNO=10;
```

Ex:
waq to display employees salaries after increment of 10%?

```
SQL> SELECT ENAME,SAL BEFORE_INCREMENT,
  2  SAL+SAL*10/100 AFTER_INCREMENT FROM EMP;
```

Ex:
waq to update employees salaries with the increment of 5% who are
working as a "manager"?

```
SQL> UPDATE EMP SET SAL=SAL+SAL*0.05 WHERE JOB='MANAGER';
```

iii) Relational operators:
==========

- to comparing a specific column values with user defined condition in the query.

syntax:
======

```
where <column name> <relational operator> <value>;
```

Ex:
waq to display employees who are joined before 1981?

```
SQL> SELECT * FROM EMP WHERE HIREDATE<'01-JAN-81';
```

Ex:
waq to display employees who are joined after 1981?

```
SQL> SELECT * FROM EMP WHERE HIREDATE>'31-DEC-81';
```

iv) Logical operators:
==========

- to check more than one condition in the query.
- these operators are AND,OR,NOT.

AND:
=====

- it return a value if both conditions are true in the query.

cond1	cond2	
T	T	----> T
T	F	----> F
F	T	----> F
F	F	----> F

syntax:
======

```
where <condition1> AND <condition2>
```

Ex:
waq to display employees whose name is "SMITH" and working as a "CLERK"?
SQL> SELECT * FROM EMP WHERE ENAME='SMITH' AND JOB='CLERK';

OR:
===
- it returns a value if any one condition is true from the given group of conditions in the query.

cond1	cond2	
T	T	----> T
T	F	----> T
F	T	----> T
F	F	----> F

syntax:
=======
where <condition1> or <condition2>

Ex:
waq to display empoyees who are working as a "analyst" or whose salary is more than 2000?
SQL> SELECT * FROM EMP WHERE JOB='ANALYST' OR SAL>2000;

Ex:
waq to display employees whose empno is 7369,7566,7788,7900?
```
SQL> SELECT * FROM EMP WHERE
  2  EMPNO=7369 OR EMPNO=7566
  3  OR EMPNO=7788 OR EMPNO=7900;
```

NOT:
====
- it return all values from a table except the given conditional values in the query.

syntax:
======
where not <column name>=<value> and not <column name>=<value>

Ex:
waq to display employees who are not working as a "clerk" and as a "salesman"?
```
SQL> SELECT * FROM EMP WHERE
  2  NOT JOB='CLERK' AND NOT JOB='SALESMAN';
```

v) Set operators
==============
- ARE USED TO COMBINED THE RESULTS OF TWO SELECT STATEMENTS AS A SINGLE
SET OF VALUES.

SYNTAX:

========
<SELECT QUERY1> <SET OPERATOR> <SELECT QUERY2>;

EX:
A={10,20,30} B={30,40,50}

UNION:
======
- TO COMBINED BOTH SETS VALUES WITHOUT DUPLICATES.

A U B = {10,20,30,40,50}

UNION ALL:
==========
- TO COMBINED BOTH SETS VALUES WITH DUPLICATES.

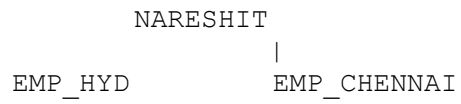
A UL B = {10,20,30,30,40,50}

INTERSECT:
==========
- TO RETURN COMMON VALUES FROM BOTH SETS.

A I B = {30}

MINUS:
======
- IT RETURNS UNCOMMON VALUES FROM THE LEFT SET ONLY.

A - B = { 10,20}
B - A = {40,50}



SQL> SELECT * FROM EMP_HYD;

EID	ENAME	SAL
1021	SMITH	85000
1022	ALLEN	68000
1023	WARD	47000

SQL> SELECT * FROM EMP_CHENNAI;

EID	ENAME	SAL
1021	SMITH	85000
1024	MILLER	55000
1025	JONES	27000

EX:

```
WAQ TO DISPLAY EMPLOYEES WHO ARE WORKING IN BOTH BRANCHES?
SQL> SELECT * FROM EMP_HYD INTERSECT SELECT * FROM EMP_CHENNAI;

EX:
WAQ TO DISPLAY EMPLOYEES WHO ARE WORKING IN HYD BUT NOT IN CHENNAI BRANCH?
SQL> SELECT ENAME FROM EMP_HYD
  2  MINUS
  3  SELECT ENAME FROM EMP_CHENNAI;

ENAME
----------
ALLEN
WARD

EX:
WAQ TO DISPLAY ALL EMPLOYEES WHO ARE WORKING UNDER "NARESHIT"
ORGANIZATION?
SQL> SELECT * FROM EMP_HYD
  2  UNION ALL
  3  SELECT * FROM EMP_CHENNAI; ----> INCLUDING DUPLICATES
       (OR)
SQL> SELECT * FROM EMP_HYD
  2  UNION
  3  SELECT * FROM EMP_CHENNAI;-----> TO ELIMINATE DUPLICATE

vi) Special operators =>        (+ve) operators          (-ve) operator
                                 =============           ============
                                   IN                     NOT IN
                                   BETWEEN                  NOT BETWEEN
                                   IS NULL        IS NOT NULL
                                   LIKE                   NOT LIKE


IN :
===
       - COMPARING THE LIST OF VALUES IN A SINGLE CONDITION.
SYNTAX:
========
       WHERE <COLUMN NAME> IN(VALUE1,VALUE2,.......);

EX:
TO DISPLAY EMPLOYEES WHOSE EMPNO IS 7369,7566,7788,7900?
SQL> SELECT * FROM EMP WHERE EMPNO IN(7369,7566,7788,7900);

EX:
WAQ TO DISPLAY EMPLOYEES WHO ARE NOT WORKING AS A "CLERK","SALESMAN",
"MANAGER"?
SQL> SELECT * FROM EMP WHERE JOB NOT IN('CLERK','SALESMAN','MANAGER');

BETWEEN:
=========
       - WORKING ON A PARTICULAR RANGE VALUE.
       - IT RETURNS INCLUDING SOURCE AND DESTINATION VALUES.
       - IT ALWAYS WORK ON LOW VALUE TO HIGH VALUE.
```

SYNTAX:
========
BETWEEN <LOW VALUE> AND <HIGH VALUE>;

EX:
WAQ TO DISPLAY EMPLOYEES WHOSE SALARY IS BETWEEN 1600 AND 3000?
SQL> SELECT * FROM EMP WHERE SAL BETWEEN 1600 AND 3000;

EX:
TO DISPLAY EMPLOYEES WHO ARE JOINED IN 1981?
SQL> SELECT * FROM EMP WHERE HIREDATE BETWEEN '01-JAN-81' AND '31-DEC-81';

EX:
TO DISPLAY EMPLOYEES WHO ARE NOT JOINED IN 1981?
SQL> SELECT * FROM EMP WHERE HIREDATE NOT BETWEEN
'01-JAN-81' AND '31-DEC-81';

IS NULL:
========
- COMPARING NULLS IN A TABLE.

SYNTAX:
========
WHERE <COLUMN NAME> IS NULL;

EX:
WAQ TO DISPLAY EMPLOYEES WHOSE COMMISSION IS NULL / EMPTY ?
SQL> SELECT * FROM EMP WHERE COMM IS NULL;

EX:
WAQ TO DISPLAY EMPLOYEES WHOSE COMMISSION IS NOT NULL / IS NOT EMPTY ?
SQL> SELECT * FROM EMP WHERE COMM IS NOT NULL;

WHAT IS NULL?
==============
- UNKNOWN / UNDEFINED VALUE / EMPTY .
- NULL != 0 & NULL != SPACE.

EX:
WAQ TO DISPLAY EMPNO,ENAME,SALARY,COMMISSION AND ALSO
SALARY+COMMISSION FROM EMP TABLE WHOSE ENAME IS "SMITH'?

SQL> SELECT EMPNO,ENAME,SAL,COMM,SAL+COMM FROM EMP
WHERE ENAME='SMITH';

EMPNO	ENAME	SAL	COMM	SAL+COMM
7369	SMITH	800		

NOTE:
======
- IF ANY ARITHMETIC OPERATOR IS PERFORM SOME OPERATION WITH NULL

THEN IT AGAIN RETURN "NULL" ONLY.

Ex:

I) IF X =1000;
=> X+NULL ==> 1000+NULL ===> NULL
=> X-NULL ==> 1000-NULL ====> NULL
=> X*NULL ==> 1000*NULL ====> NULL
=> X/NULL ==> 1000/NULL =====> NULL

- TO OVERCOME THE ABOVE PROBLEM WE USE A PRE-DEFINED FUNCTION IN ORACLE DB IS "NVL()".

NVL(EXP1,EXP2):
==========

- IT IS A PRE-DEFINED FUNCTION WHICH IS USED TO REPLACE A USER DEFINED VALUE IN PLACE OF NULL IN THE GIVEN EXPRESSION.
- IT IS HAVING TWO ARGUMENTS ARE EXPRESSION1 AND EXPRESSION2.
IF EXP1 IS NULL ----------> RETURN EXP2 VALUE(UD VALUE)
IF EXP1 IS NOT NULL ---> RETURN EXP1 VALUE ONLY.

EX:

```
SQL> SELECT NVL(NULL,0) FROM DUAL;

NVL(NULL,0)
-----------
          0

SQL> SELECT NVL(NULL,100) FROM DUAL;

NVL(NULL,100)
-------------
          100

SQL> SELECT NVL(0,100) FROM DUAL;

NVL(0,100)
----------
         0

SQL> SELECT NVL(200,100) FROM DUAL;

NVL(200,100)
------------
         200
```

SOLUTION:
==========

```
SQL> SELECT EMPNO,ENAME,SAL,COMM,SAL+NVL(COMM,0) FROM EMP
         WHERE ENAME='SMITH';

  EMPNO               ENAME      SAL       COMM   SAL+NVL(COMM,0)
----------------      ----------      ----------    ----------
------------------------------
      7369            SMITH       800              800
```

NVL2(EXP1,EXP2,EXP3):
=====================
- IT IS AN EXTENSION OF NVL().HAVING 3 ARGUMENTS ARE EXPRESSION1,EXPRESSION2 AND EXPRESSION3.

=> IF EXP1 IS NULL ------> RETURN EXP3 VALUE(UD VALUE)
=> IF EXP1 IS NOT NULL ---> RETURN EXP2 VALUE(UD VALUE)

EX:
```
SQL> SELECT NVL2(NULL,100,200) FROM DUAL;

NVL2(NULL,100,200)
------------------
               200

SQL> SELECT NVL2(0,100,200) FROM DUAL;

NVL2(0,100,200)
-------------------------
       100
```

EX:
WAQ TO UPDATE ALL EMPLOYEES COMMISSIONS IN A TABLE BASED ON THE FOLLOWING CONDITIONS ARE,
I) IF COMM IS NULL THEN UPDATE THOSE EMPLOYEES COMMISSIONS WITH 800.
II) IF COMM IS NOT NULL THEN UPDATE THOSE EMPLOYEES COMMISSIONS WITH COMM+800.

SOLUTION:
```
SQL> UPDATE EMP SET COMM=NVL2(COMM,COMM+800,800);
```

LIKE:
=====
- TO COMPARING A SPECIFIC STRING CHARACTER PATTERN.
- WHEN WE USE "LIKE" OPERATOR WE SHOULD USE THE FOLLOWING WILDCARD OPERATORS ARE

i) % - THE REMAINING GROUP OF CHAR'S AFTER SELECTED CHARACTER.
ii) _ - COUNTING A SINGLE CHARACTER IN THE EXPRESSION.

SYNTAX:
========
WHERE <COLUMN NAME> LIKE '<WILDCARD OPERATOR> <CHAR.PATTERN> <WILDCARD OPERATOR> ';

EX:
TO DISPLAY EMPLOYEES WHOSE NAME STARTS WITH 'S' CHARACTER?
```
SQL> SELECT * FROM EMP WHERE ENAME LIKE 'S%';
```

EX:
TO DISPLAY EMPLOYEES WHOSE NAME ENDS WITH 'N' CHARACTER?

```
SQL> SELECT * FROM EMP WHERE ENAME LIKE '%N';
```

EX:
TO DISPLAY EMPLOYEES WHOSE NAME IS HAVING "I" CHARACTER?

```
SQL> SELECT * FROM EMP WHERE ENAME LIKE '%I%';
```

EX:
TO DISPLAY EMPLOYEES WHOSE NAME IS HAVING 4 CHARACTERS?

```
SQL> SELECT * FROM EMP WHERE ENAME LIKE '____';
```

EX:
TO DISPLAY EMPLOYEES WHOSE NAME IS HAVING 2ND CHARACTER IS "O" ?

```
SQL> SELECT * FROM EMP WHERE ENAME LIKE '_O%';
```

EX:
TO DISPLAY EMPLOYEES WHOSE EMPNO IS STARTS WITH 7 AND ENDS WITH 8?

```
SQL> SELECT * FROM EMP WHERE ENAME LIKE '7%8';
```

EX:
TO DISPLAY EMPLOYEES WHO ARE JOINED IN 1981?

```
SQL> SELECT * FROM EMP WHERE HIREDATE LIKE '%81';
```

EX:
TO DISPLAY EMPLOYEES WHO ARE JOINED IN THE MONTH OF "DECEMBER"?

```
SQL> SELECT * FROM EMP WHERE HIREDATE LIKE '%DEC%';
```

EX:
TO DISPLAY EMPLOYEES WHO ARE JOINED IN THE MONTH OF "MAY","DECEMBER"?

```
SQL> SELECT * FROM EMP WHERE HIREDATE LIKE '%MAY%'
         OR HIREDATE LIKE '%DEC%';
```

LIKE OPERATOR WITH SPECIAL CHARACTERS:

```
========================================
ENAME                SAL
----------           ----------
%SMITH               1200
WAR_NER              1500
JAME@S               3200
MILL#ER              2500
_JONES               1100
```

EX:
TO DISPLAY EMPLOYEES WHOSE NAME IS HAVING "@" SYMBOL?

```
SQL> SELECT * FROM TEST WHERE ENAME LIKE '%@%';
```

EX:
TO DISPLAY EMPLOYEES WHOSE NAME IS HAVING "#" SYMBOL?

```
SQL> SELECT * FROM TEST WHERE ENAME LIKE '%#%';
```

EX:
TO DISPLAY EMPLOYEES WHOSE NAME IS HAVING "_" SYMBOL?

```
SQL> SELECT * FROM TEST WHERE ENAME LIKE '%_%';
```

NOTE:

======

- WHEN WE PERFORM LIKE OPERATOR ON "_ , %" THEN ORACLE SERVER WILL TREAT AS
"WILDCARD OPERATORS" BUT NOT TREAT AS "SPECIAL CHARACTERS" SO TO OVERCOME THIS PROBLEM WE SHOULD USE A SPECIAL KEYWORD IS CALLED AS " ESCAPE'\' ".

SOLUTION:

```
SQL> SELECT * FROM TEST WHERE ENAME LIKE '%\_%'ESCAPE'\';
SQL> SELECT * FROM TEST WHERE ENAME LIKE '%\%%'ESCAPE'\';
```

EX:

WAQ TO DISPLAY EMPLOYEES WHOSE NAME NOT STARTS WITH "S" CHARACTER?

```
SQL> SELECT * FROM EMP WHERE ENAME NOT LIKE 'S%';
```

SMITH

SUMAN
SCOTT
SURESH

```
121
25.36
250000.00
'SAI'
'87HJ87KJ98'
25-JAN-2022
10:20:35.44
0101010101111000000
```

FUNCTIONS:
==========

- TO PERFORM SOME TASK AND MUST RETURN A VALUE.
- ORACLE SUPPORTING THE FOLLOWING TWO TYPES OF FUNCTIONS.
    1. PRE-DEFINED FUNCTIONS
        > USE IN SQL & PL/SQL.
    2. USER-DEFINED FUNCTIONS
        > USE IN PL/SQL ONLY

1. PRE-DEFINED FUNCTIONS:
==========
- THESE FUNCTIONS ARE ALSO CALLED AS "BUILT IN FUNCTIONS" IN ORACLE

i) SINGLE ROW FUNCTIONS (SCALAR FUNCTIONS)

ii) MULTIPLE ROW FUNCTIONS(GROUPING / AGGREGATIVE FUNCTIONS)

i) SINGLE ROW FUNCTIONS:
==========
- THESE FUNCTIONS ARE ALSO RETURNS A SINGLE VALUE.
    1) NUMERIC FUNCTIONS
    2) STRING FUNCTIONS
    3) DATE FUNCTIONS
    4) CONVERSION FUNCTIONS
    5) ANALYTICAL FUNCTIONS(SEE IN SUBQUERY)

HOW TO CALL A FUNCTION:
==========
SYNTAX:
========
```
SELECT <FNAME>(VALUE/(S)) FROM DUAL;
```

WHAT IS DUAL?
==========
- PRE-DEFINED TABLE IN ORACLE.
- IS USED TO TEST FUNCTION FUNCTIONALITIES(WORKING STYLE).
- IS HAVING A SINGLE ROW & A SINGLE COLUMN.
- IS ALSO CALLED AS "DUMMY TABLE" IN ORACLE.

TO VIEW THE STRUCTURE OF DUAL TABLE:
==========
SYNTAX:
========
```
DESC DUAL;
```

```
 Name
Null?    Type
```

```
 DUMMY
VARCHAR2(1)
```

TO VIEW DATA OF DUAL TABLE:
==========
SYNTAX:
========
```
SELECT * FROM DUAL;
```

```
D
-
X
```

1) NUMERIC FUNCTIONS

ABS():

- TO CONVERT (-ve) SIGN VALUES INTO (+ve) SIGN VALUES.

SYNTAX:

```
ABS(n)
```

EX:

```
SQL> SELECT ABS(-12) FROM DUAL;---------12
SQL> SELECT EMPNO,ENAME,SAL,COMM,ABS(COMM-SAL) FROM EMP;
```

CEIL():

- IT RETURNS UPPER BOUND VALUE.

SYNTAX:

```
CEIL(n)
```

EX:

```
SQL> SELECT CEIL(9.3) FROM DUAL;

 CEIL(9.3)
----------
        10

SQL> SELECT CEIL(9.8) FROM DUAL;

 CEIL(9.8)
----------
        10
```

FLOOR():

- IT RETURNS LOWER BOUND VALUE.

SYNTAX:

```
FLOOR(n)
```

EX:

```
SQL> SELECT FLOOR(9.8) FROM DUAL;

FLOOR(9.8)
----------
         9
```

```
SQL> SELECT FLOOR(9.3) FROM DUAL;

FLOOR(9.3)
----------
         9
MOD():
======
       - IT RETURNS REMAINDER VALUE.
SYNTAX:
=======
       MOD(m,n)

EX:
SQL> SELECT MOD(10,2) FROM DUAL;

 MOD(10,2)
----------
         0

ROUND():
========
       - IT RETURNS THE NEAREST  VALUE OF GIVEN EXPRESSION.
       - WILL CONSIDER 0.5 VALUE.
                IF EXPRESSION IS  >= 0.5 THEN WE ADD "1" TO THE
EXPRESSION.
                IF EXPRESSION IS <0.5 THEN WE ADD "0" TO THE EXPRESSION.

SYNTAX:
=======
       ROUND(EXPRESSION,[DECIMAL PALCES])

EX:
SQL> SELECT ROUND(36.45) FROM DUAL;

ROUND(36.45)
------------
          36

SQL> SELECT ROUND(36.50) FROM DUAL;

ROUND(36.50)
------------
          37

SQL> SELECT ROUND(36.82) FROM DUAL;

ROUND(36.82)
------------
          37

SQL> SELECT ROUND(36.824,2) FROM DUAL;

ROUND(36.824,2)
---------------
```

```
          36.82

SQL> SELECT ROUND(36.825,2) FROM DUAL;

ROUND(36.825,2)
---------------
          36.83

SQL> SELECT ROUND(36.827,2) FROM DUAL;

ROUND(36.827,2)
---------------
          36.83

TRUNC():
========
       - IT RETURN AN EXACT VALUE FROM GIVEN EXPRESSION.
       - IT IS NOT CONSIDER 0.5 VALUE IN THE EXPRESSION.

SYNTAX:
========
       TRUNC(EXPRESSION,[DECIMAL PALCE])

EX:
SQL> SELECT TRUNC(36.45) FROM DUAL;

TRUNC(36.45)
------------
          36

SQL> SELECT TRUNC(36.855,2) FROM DUAL;

TRUNC(36.855,2)
---------------
          36.85

STRING FUNCTIONS:
==================
LENGTH():
=========
       - IT RETURNS THE NO.OF CHARACTERS IN THE GIVEN EXPRESSION.

SYNTAX:
========
       LENGTH(STRING)

EX:
SQL> SELECT LENGTH('HELLO') FROM DUAL;

LENGTH('HELLO')
---------------
              5

SQL> SELECT LENGTH('HEL LO') FROM DUAL;
```

```
LENGTH('HELLO')
---------------
              6
SQL> SELECT ENAME,LENGTH(ENAME) FROM EMP;
SQL> SELECT * FROM EMP WHERE LENGTH(ENAME)=6;

LOWER():
========
       - TO CONVERT UPPER CASE CHAR's INTO LOWER CASE CHAR's.

SYNTAX:
========
       LOWER(STRING)

EX:
SQL> SELECT LOWER('HELLO') FROM DUAL;
SQL> SELECT ENAME,LOWER(ENAME) FROM EMP;
SQL> UPDATE EMP SET ENAME=LOWER(ENAME) WHERE JOB='MANAGER';
SQL> UPDATE EMP SET ENAME=LOWER(ENAME);

UPPER():
=======
       - TO CONVERT LOWER CASE CHAR's INTO UPPER CASE CHAR's.

SYNTAX:
=======
       UPPER(STRING)
EX:
SQL> UPDATE EMP SET ENAME=UPPER(ENAME);

INITCAP():
=========
       - THE FIRST CHARACTER IS CAPITAL IN THE GIVEN STRING.

SYNTAX:
========
       INITCAP(STRING)

EX:
SQL> SELECT INITCAP('HELLO') FROM DUAL;

INITC
-----
Hello

SQL> SELECT INITCAP('hello') FROM DUAL;

INITC
-----
Hello

SQL> SELECT ENAME,INITCAP(ENAME) FROM EMP;
SQL> UPDATE EMP SET ENAME=INITCAP(ENAME);
```

```
LTRIM():
=======
       - TO REMOVE UNWANTED CHARACTERS / SPACES FROM THE GIVEN STRING ON
LEFT SIDE.

SYNTAX:
========
       LTRIM(STRING1[,STRING2])

EX:
SQL> SELECT LTRIM('  HELLO') FROM DUAL;

LTRIM
-----
HELLO

SQL> SELECT LTRIM('XXXXXHELLO','X') FROM DUAL;

LTRIM
-----
HELLO

SQL> SELECT LTRIM('123HELLO','123') FROM DUAL;

LTRIM
-----
HELLO

RTRIM():
=======
       - TO REMOVE UNWANTED CHARACTERS / SPACES FROM THE GIVEN STRING ON
RIGHT SIDE.

SYNTAX:
========
       RTRIM(STRING1[,STRING2])


EX:
SQL> SELECT RTRIM('HELLO123','123') FROM DUAL;

RTRIM
-----
HELLO

TRIM():
======
       - TO REMOVE UNWANTED CHARACTERS FROM THE BOTH SIDES OF A STRING.

SYNTAX:
=======
       TRIM('TRIMMING CHARACTER' FROM STRING)
```

```
EX:
SQL> SELECT TRIM('X' FROM 'XXXXXXHELLOXXXX') FROM DUAL;

TRIM(
-----
HELLO

CONCAT():
========
       - ADDING TWO STRING EXPRESSION.

SYNTAX:
=======
       CONCAT(STRING1,STRING2)

EX:
SQL> SELECT CONCAT('GOOD','EVENING') FROM DUAL;

CONCAT('GOO
-----------
GOODEVENING

SQL> SELECT CONCAT('Mr.',ENAME) FROM EMP;

CONCAT('MR.',
-------------
Mr.SMITH

LPAD():
======
       - TO FILL A STRING WITH A SPECIFIC CHARACTER ON LEFT SIDE.

SYNTAX:
========
       LPAD(STRING1,LENGTH,'<SPECIFIC CHARACTER>')

EX:
SQL> SELECT LPAD('HELLO',10,'@') FROM DUAL;

LPAD('HELL
----------
@@@@@HELLO

RPAD():
======
       - TO FILL A STRING WITH A SPECIFIC CHARACTER ON RIGHT SIDE.

SYNTAX:
========
       RPAD(STRING1,LENGTH,'<SPECIFIC CHARACTER>')


EX:
SQL> SELECT RPAD('HELLO',10,'@') FROM DUAL;
```

```
RPAD('HELL
----------
HELLO@@@@@

REPLACE():
=========
       - TO REPLACE A PARTICULAR STRING CHARACTERS WITH ANOTHER STRING
CHARACTERS.

SYNTAX:
========
       REPLACE(STRING,<OLD CHAR's>,<NEW CHAR's>)

EX:
SQL> SELECT REPLACE('HELLO','EL','XYZ') FROM DUAL;

REPLAC
------
HXYZLO

SQL> SELECT REPLACE('JACK AND JUE','J','BL') FROM DUAL;

REPLACE('JACKA
--------------
BLACK AND BLUE

TRANSLATE():
============
       - TO TRANSLATE CHARACTER BY CHARACTER.

SYNTAX:
=======
       TRANSLATE(STRING,<CHAR's>,<CHAR's>)

EX:
SQL> SELECT TRANSLATE('HELLO','ELO','ABC') FROM DUAL;

TRANS
-----
HABBC

SQL> SELECT TRANSLATE('HELLO','ELO','AB') FROM DUAL;

TRAN
----
HABB

SUBSTR():
========
       - TO RETRUN THE REQUIRED SUB STRING FROM THE GIVEN STRING
EXPRESSION.

SYNTAX:
```

========
SUBSTR(STRING,<STARTING POSITION OF CHAR>,<LENGTH OF CHAR's>)

EX:
SQL> SELECT SUBSTR('WELCOME',2,4) FROM DUAL;

SUBS

ELCO

SQL> SELECT SUBSTR('WELCOME',-2,1) FROM DUAL;

S
-
M

INSTR():
======
- IT RETURNS THE OCCURRENCE POSITION OF CHARACTER FROM THE GIVEN STRING EXPRESSION.

SYNTAX:
=======
INSTR(STRING1,STRING2,<STARTING POSITION OF CHARACTER>,<OCCURRENCE POSITION OF CHAR>)

NOTE:
=====
- THE POSITIONS OF CHARACTERS ARE ALWAYS FIXED IF WE ARE COUNTING FROM LEFT TO
RIGHT (OR) FROM RIGHT TO LEFT.

EX:
WELCOME HELLO
1234567 8 90123

EX:
SQL> SELECT INSTR('WELCOME HELLO','O') FROM DUAL;

INSTR('WELCOMEHELLO','O')

5

SQL> SELECT INSTR('WELCOME HELLO','O',1,2) FROM DUAL;

INSTR('WELCOMEHELLO','O',1,2)

13

SQL> SELECT INSTR('WELCOME HELLO','O',7,2) FROM DUAL;

INSTR('WELCOMEHELLO','O',7,2)

```
                            0

SQL> SELECT INSTR('WELCOME HELLO','O',7,1) FROM DUAL;

INSTR('WELCOMEHELLO','O',7,1)
-----------------------------
                           13

SQL> SELECT INSTR('WELCOME HELLO','E',1,1) FROM DUAL;

INSTR('WELCOMEHELLO','E',1,1)
-----------------------------
                            2

SQL>
SQL> SELECT INSTR('WELCOME HELLO','E',1,2) FROM DUAL;

INSTR('WELCOMEHELLO','E',1,2)
-----------------------------
                            7

SQL> SELECT INSTR('WELCOME HELLO','E',1,3) FROM DUAL;

INSTR('WELCOMEHELLO','E',1,3)
-----------------------------
                           10

SQL>
SQL> SELECT INSTR('WELCOME HELLO','E',1,4) FROM DUAL;

INSTR('WELCOMEHELLO','E',1,4)
-----------------------------
                            0

SQL>
SQL> SELECT INSTR('WELCOME HELLO','L',11,1) FROM DUAL;

INSTR('WELCOMEHELLO','L',11,1)
------------------------------
                            11

SQL> SELECT INSTR('WELCOME HELLO','L',-4,1) FROM DUAL;

INSTR('WELCOMEHELLO','L',-4,1)
------------------------------
                             3

SQL> SELECT INSTR('WELCOME HELLO','E',-4,1) FROM DUAL;

INSTR('WELCOMEHELLO','E',-4,1)
------------------------------
                            10

SQL> SELECT INSTR('WELCOME HELLO','E',-4,2) FROM DUAL;
```

```
INSTR('WELCOMEHELLO','E',-4,2)
------------------------------
                             7

SQL> SELECT INSTR('WELCOME HELLO','E',-4,3) FROM DUAL;

INSTR('WELCOMEHELLO','E',-4,3)
------------------------------
                             2

SQL> SELECT INSTR('WELCOME HELLO','E',-4,4) FROM DUAL;

INSTR('WELCOMEHELLO','E',-4,4)
------------------------------
                             0

DATE FUNCTIONS:
===============
SYSDATE:
========
       - TO RETURN THE CURRENT DATE INFORMATION OF THE SYSTEM.

SYNTAX:
=======
       SYSDATE

EX:
SQL> SELECT SYSDATE FROM DUAL;

SYSDATE
---------
13-OCT-22

SQL> SELECT SYSDATE CURRENT_DATE,SYSDATE+5 NEW_DATE FROM DUAL;

CURRENT_DATE    NEW_DATE
-------------------------   --------------------
13-OCT-22     18-OCT-22

SQL> SELECT SYSDATE CURRENT_DATE,SYSDATE-5 NEW_DATE FROM DUAL;

CURRENT_DATE    NEW_DATE
-------------------------  ------------------
13-OCT-22     08-OCT-22

ADD_MONTHS():
=============
       - TO ADD / SUBTRACT NO.OF MONTHS FROM THE GIVE DATE EXPRESSION.

SYNTAX:
=======
       ADD_MONTHS(DATE,<NO.OF MONTHS>)
```

EX:

```
SQL> SELECT SYSDATE,ADD_MONTHS(SYSDATE,3) FROM DUAL;

SYSDATE        ADD_MONTH
---------              ---------
13-OCT-22      13-JAN-23

SQL> SELECT SYSDATE,ADD_MONTHS(SYSDATE,-3) FROM DUAL;

SYSDATE        ADD_MONTH
---------              ---------
13-OCT-22      13-JUL-22
```

EX:

```
SQL> CREATE TABLE PRODUCT
  2  (
  3  PCODE INT,
  4  MFG DATE,
  5  EXP DATE
  6  );
Table created.

SQL> INSERT INTO PRODUCT(PCODE,MFG)
  2  VALUES(1021,'14-FEB-2021');

SQL> INSERT INTO PRODUCT(PCODE,MFG)
  2  VALUES(1022,'25-JUN-2022');


SQL> SELECT * FROM PRODUCT;

     PCODE     MFG                    EXP
----------             ---------              ---------
      1021             14-FEB-21
      1022             25-JUN-22

SQL> UPDATE PRODUCT SET EXP=ADD_MONTHS(MFG,24);
2 rows updated.
SQL> SELECT * FROM PRODUCT;

     PCODE     MFG                    EXP
----------             ---------              ---------
      1021             14-FEB-21      14-FEB-23
      1022             25-JUN-22      25-JUN-24
```

LAST_DAY():

==========

- IT RETURN THE LAST DAY FROM THE GIVEN MONTH.

SYNTAX:

========

LAST_DAY(DATE)

EX:

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

SYSDATE        LAST_DAY(
---------             ---------
14-OCT-22      31-OCT-22

MONTHS_BETWEEN():
==================
       - IT RETURN THE NO.OF MONTHS BETWEEN THE GIVEN TWO DATE
EXPRESSIONS.

SYNTAX:
========
       MONTHS_BETWEEN(DATE1,DATE2)

NOTE:
======
       - DATE1 IS ALWAYS GREATER THAN TO DATE2 OTHERWISE IT RETURNS (-ve)
SIGN VALUE.

EX:
SQL> SELECT MONTHS_BETWEEN('05-JAN-81','05-JAN-82')FROM DUAL;

MONTHS_BETWEEN('05-JAN-81','05-JAN-82')
---------------------------------------
                                    -12

SQL> SELECT MONTHS_BETWEEN('05-JAN-81','05-JAN-80')FROM DUAL;

MONTHS_BETWEEN('05-JAN-81','05-JAN-80')
---------------------------------------
                                     12

CONVERSION FUNCTIONS:
======================
       i) TO_CHAR()
       ii) TO_DATE()

i) TO_CHAR():
===========
       - TO CONVERT DATE TYPE INTO CHARACTER TYPE AND ALSO DISPLAY DATE
IN DIFFERENT FORMATS.

SYNTAX:
=======
       TO_CHAR(SYSDATE,<INTERVAL>)

YEAR FORMAT INTERVALS:
=======================
YYYY    - RETURN IN FOUR DIGITS FORMAT(2022)
YY      - RETURN IN THE LAST TWO DIGITS(22)
YEAR    - Twenty Twenty-Two
CC      - Centuary 21
AD / BC     - Ad YEAR / Bc YEAR
```

EX:
SQL> SELECT TO_CHAR(SYSDATE,'YYYY YY YEAR CC BC') FROM DUAL;

```
TO_CHAR(SYSDATE,'YYYYYYYEARCCBC')
---------------------------------------------------
2022 22 TWENTY TWENTY-TWO 21 AD
```

MONTH INTERVALS:
==========
MM - Month IN Number Format
MON - First Three Char From Month Spelling
MONTH - Full Name Of Month

EX:
SQL> SELECT TO_CHAR(SYSDATE,'MM MON MONTH') FROM DUAL;

```
TO_CHAR(SYSDATE,'MMMONMONTH')
-----------------------------------------------------
10 OCT OCTOBER
```

DAY INTERVALS:
==========
DDD - Day Of The Year.
DD - Day Of The Month.
D - Day Of The Week
Sun - 1
Mon - 2
Tue - 3
Wen - 4
Thu - 5
Fri - 6
Sat - 7
DAY - Full Name Of The Day
DY - First Three Char's Of Day Spelling

EX:
SQL> SELECT TO_CHAR(SYSDATE,'DDD DD D DAY DY') FROM DUAL;

```
TO_CHAR(SYSDATE,'DDDDDDDAYDY')
-----------------------------------------------------------
287 14 6 FRIDAY    FRI
```

Quater Format Intervals:

Q - One Digit Quater Of The Year.

1 - Jan - Mar
2 - Apr - Jun
3 - Jul - Sep
4 - Oct - Dec

EX:
SQL> SELECT TO_CHAR(SYSDATE,'Q') FROM DUAL;

```
T
-
4

Week Format Intervals:
-----------------------------------
WW      - Week Of The Year
W       - Week Of Month

EX:
SQL> SELECT TO_CHAR(SYSDATE,'WW W') FROM DUAL;

TO_C
----
41 2

ii) TO_DATE():
============
       - TO CONVERT CHARACTER TYPE TO ORACLE DEFAULT DATE TYPE.

SYNTAX:
========
       TO_DATE('STRING')

EX:
SQL> SELECT TO_DATE('14/OCTOBER/2022') FROM DUAL;

TO_DATE('
---------
14-OCT-22

SQL> SELECT TO_DATE('14/OCTOBER/2022')+5 FROM DUAL;

TO_DATE('
---------
19-OCT-22
================================================================================
==
EX:
WAQ TO DISPLY EMPLOYEES WHO ARE JOINED IN 1981 BY USING TO_CHAR()?
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'YYYY')='1981';

EX:
WAQ TO DISPLAY EMPLOYEES WHO ARE JOINED 1980,1982,1983 BY USING TO_CHAR()?
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'YY') IN(80,82,83);

EX:
WAQ TO DISPLAY EMPLOYEES WHO ARE JOINED IN MONTH OF DECEMBER BY USING
TO_CHAR()?
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'MM')='12';

EX:
WAQ TO DISPLAY EMPLOYEES WHO ARE JOINED IN 2ND QUATER OF THE YEAR?
```

```
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'Q')=2;

EX:
WAQ TO DISPLAY EMPLOYEES WHO ARE JOINED IN 2ND QUATER OF 1981?
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'YYYY')='1981' AND
TO_CHAR(HIREDATE,'Q')=2;

ii) MULTIPLE ROW FUNCTIONS:
==========================
       - THESE FUNCTIONS ARE ALSO CALLED AS "AGGREGATIVE / GROUPING
FUNCTIONS" IN ORACLE DB.
       - THESE FUNCTION ARE "SUM(),AVG(),MIN(),MAX(),COUNT()".

SUM():
======
       - RETURN TOTAL VALUE.

EX:
SQL> SELECT SUM(SAL) TOTAL_SALARY FROM EMP;

TOTAL_SALARY
------------
       29025

SQL> SELECT SUM(SAL) TOTAL_SALARY FROM EMP WHERE JOB='CLERK';

TOTAL_SALARY
------------
        4150

AVG():
=====
       - RETURN AVERAGE OF TOTAL.

EX:
SQL> SELECT AVG(SAL) AVG_SALARY FROM EMP;

AVG_SALARY
----------
2073.21429

SQL> SELECT AVG(SAL) AVG_SALARY FROM EMP WHERE JOB='CLERK';

AVG_SALARY
----------
    1037.5

MIN():
=====
       - RETURN MINIMUM VALUE.

EX:
SQL> SELECT MIN(SAL) FROM EMP;
```

```
  MIN(SAL)
----------
       800

SQL> SELECT MIN(HIREDATE) FROM EMP;

MIN(HIRED
---------
17-DEC-80

SQL> SELECT MIN(SAL) FROM EMP WHERE DEPTNO=10;

  MIN(SAL)
----------
      1300
```

MAX():
=====
- RETURN MAXIMUM VALUE.

EX:
```
SQL> SELECT MAX(SAL) FROM EMP;
```

OUTPUT:
========
5000

COUNT():
========
i) COUNT(*)
ii) COUNT(COLUMN NAME)
iii) COUNT(DISTINCT <COLUMN NAME>)

```
SQL> SELECT * FROM TEST100;

       SNO NAME
---------- ----------
         1 A
         2 B
         3
         4 C
         5 B
         6 A
```

i) COUNT(*):
===========
- COUNTING ALL VALUES INCLUDING DUPLICATES AND NULLS IN A TABLE.

EX:
```
SQL> SELECT COUNT(*) FROM TEST100;----------> 6
```

ii) COUNT(COLUMN NAME):
======================
- COUNTING ALL VALUES INCLUDING DUPLICATE BUT NOT NULLS FROM A COLUMN.

EX:
SELECT COUNT(NAME) FROM TEST100;--------------> 5

iii) COUNT(DISTINCT <COLUMN NAME>):
==========
- COUNTING UNIQUE VALUES(NO DUPLICATES & NO NULLS).

EX:
SQL> SELECT COUNT(DISTINCT NAME) FROM TEST100;----------> 3

CLAUSES:
==========
- CLAUSE IS A STATEMENT WHICH WILL PROVIDE SOME FACILITIES LIKE "FILTERING ROWS,SORTING VALUES,GROUPING SIMILAR DATA,FINDING SUB TOTAL AND GRAND TOTAL "BASED ON COLUMNS AUTOMATICALLY.

> WHERE
> ORDER BY
> GROUP BY
> HAVING
> ROLLUP
> CUBE

WHERE:
========
- FILTERING ROWS BEFORE GROUPING DATA IN A TABLE.
- CAN WORK ON "SELECT,UPDATE,DELETE" COMMANDS ONLY.

SYNTAX:
=======
WHERE <FILTERING CONDITION>;

EX:
SELECT * FROM EMP WHERE EMPNO=7788;
UPDATE EMP SET SAL=56000 WHERE DEPTNO=20;
DELETE FROM EMP WHERE JOB='CLERK';

ORDER BY:
==========
- TO ARRANGE A SPECIFIC COLUMN VALUES IN ASCENDING OR DESCENDING ORDER.
- BY DEFAULT ORDER BY CAN ARRANGE VALUES IN "ASCENDING" ORDER.IF WE WANT TO ARRANGE IN DESCENDING ORDER THEN WE "DESC" KEYWORD.

SYNTAX:
========
SELECT * FROM <TN> ORDER BY <COLUMN NAME1> <ASC/DESC>,<COLUMN NAME2> <ASC/DESC>,.....;

EX:
SQL> SELECT * FROM EMP ORDER BY SAL;

```
SQL> SELECT SAL FROM EMP ORDER BY SAL DESC;
SQL> SELECT ENAME FROM EMP ORDER BY ENAME;
SQL> SELECT HIREDATE FROM EMP ORDER BY HIREDATE DESC;

EX:
WAQ TO DISPLAY EMPLOYEES WHO ARE WORKING UNDER DEPTNO IS 30 AND ARRANGE
THOSE EMPLOYEES SALARIES IN DESCENDING ORDER?
SQL> SELECT * FROM EMP WHERE DEPTNO=30 ORDER BY SAL DESC;

EX:
WAQ TO DISPLAY EMPLOYEES DEPTNO'S IN ASCENDING ORDER AND THEIR SALARIES
IN DESCENDING ORDER FROM EACH DEPTNO WISE?
SQL> SELECT * FROM EMP ORDER BY DEPTNO,SAL DESC;

NOTE:
=====
       - ORDER BY CLAUSE CAN APPLY ON NOT ONLY COLUMN NAMES EVEN THOUGH
IT CAN APPLY ON THE POSITION OF COLUMN IN SELECT QUERY.

EX:
SQL> SELECT * FROM EMP ORDER BY SAL DESC;
SQL> SELECT * FROM EMP ORDER BY 6 DESC;
SQL> SELECT EMPNO,ENAME,SAL FROM EMP ORDER BY 3 DESC;
SQL> SELECT ENAME,SAL FROM EMP ORDER BY 2;
SQL> SELECT SAL FROM EMP ORDER BY 1;

ORDER BY  WITH "NULL CLAUSES":
==============================
       1. NULLS FIRST:
       ===============
              - BY DEFAULT ORDER BY ON ASCENDING ORDER,
                     FIRST : VALUES
                     LATER : NULLS
       EX:
       SQL> SELECT * FROM EMP ORDER BY COMM;

       TO OVERVCOME THIS WE USE "NULLS FIRST" CLAUSE,
                     FRIST : NULLS
                     LATER : VALUES
       EX:
       SQL> SELECT * FROM EMP ORDER BY COMM NULLS FIRST;

       2. NULLS LAST:
       ==============
              - BY DEFAULT ORDER BY ON DESCENDING ORDER,
                     FIRST : NULLS
                     LATER : VALUES
       EX:
       SQL> SELECT * FROM EMP ORDER BY COMM DESC;

       TO OVERVCOME THIS WE USE "NULLS LAST" CLAUSE,
                     FRIST : VALUES
                     LATER : NULLS
       EX:
```

```
        SQL> SELECT * FROM EMP ORDER BY COMM DESC NULLS LAST;

GROUP BY:
=========
        - IS USED TO MAKE GROUPS BASED ON A COLUMNS.
        - WHEN WE USE GROUP BY WE SHOULD USE "GROUPING/AGGREGATIVE"
        FUNCTIONS ARE "SUM(),AVG(),MIN(),MAX(),COUNT()".

        EX: TO FIND OUT NO.OF FEMALES & MALES ?

                        GROUP BY
                               |
        COUNT(*):       GENDER
                               |
                FEMALE         MALE
                FEMALE         MALE
                FEMALE         MALE
                               MALE
                ========              ======
                      3          4
                ========              =======

SYNTAX:
========
SELECT <COLUMN NAME1>,...................,<AGGREGATIVE FUNCTION
NAME1>,..............
FROM <TABLE NAME> GROUP BY <COLUMN NAME1>,<COLUMN
NAME2>,.....................;

TABLE:
SQL> SELECT * FROM EMPLOYEE;

ENAME      GENDER
----------      ---------
SMITH      M
ALLEN      F
WARD       F
JONES     M
SCOTT     M

EX:
WAQ TO FIND OUT NO.OF MALE AND FEMALE EMPLOYEES?
SQL> SELECT GENDER,COUNT(*) FROM EMPLOYEE GROUP BY GENDER;
                        (OR)
SQL> SELECT GENDER,COUNT(GENDER) FROM EMPLOYEE GROUP BY GENDER;

GENDER          COUNT(GENDER)
-------------          -------------
M                      3
F                      2

EX:
WAQ TO TO FIND OUT THE NO.OF EMPLOYEES WORKING IN EACH JOB?
SQL> SELECT JOB,COUNT(JOB) FROM EMP GROUP BY JOB;
```

EX:
WAQ TO FIND OUT NO.OF EMPLOYEES WORKING IN EACH JOB ALONG WITH THEIR DEPTNO
WISE?

```
SQL> SELECT JOB,DEPTNO,COUNT(JOB) FROM EMP
  2  GROUP BY JOB,DEPTNO;
```

EX:
WAQ TO DISPLAY SUM OF SALARY OF EACH DEPTNO WISE?

```
SQL> SELECT DEPTNO,SUM(SAL) FROM EMP GROUP BY DEPTNO
  2  ORDER BY DEPTNO;
```

EX:
WAQ TO FIND OUT NO.OF EMPLOYEE,SUM OF SALARY,AVERAGE SALARY,MINIMUM SALARY AND MAXIMUM SALARY OF EACH DEPTNO?

```
SQL> SELECT DEPTNO,COUNT(DEPTNO) NO_OF_EMPLOYEES,
  2  SUM(SAL) SUM_SAL,AVG(SAL) AVG_SAL,
  3  MIN(SAL) MIN_SAL,MAX(SAL) MAX_SAL FROM EMP
  4  GROUP BY DEPTNO ORDER BY DEPTNO;
```

HAVING:
=======
- FILTERING ROW AFTER GROUPING DATA IN A TABLE.

SYNTAX:
=======

```
SELECT <COLUMN NAME1>,...................,<AGGREGATIVE FUNCTION
NAME1>,.............
FROM <TABLE NAME> GROUP BY <COLUMN NAME1>,<COLUMN NAME2>,.................
HAVING <FILTERING CONDITION>;
```

EX:
WAQ TO DISPLAY JOBS IN WHICH JOB THE NO.OF EMPLOYEES ARE MORE THAN 3?

```
SQL> SELECT JOB,COUNT(JOB) FROM EMP
  2  GROUP BY JOB HAVING COUNT(JOB)>3;

JOB                  COUNT(JOB)
---------            ----------
CLERK                4
SALESMAN             4
```

NOTE:
=====
- WE CANNOT USE "AGGREGATIVE FUNCTIONS" UNDER "WHERE CLAUSE" CONDITION.

EX:
WAQ TO DISPLAY DEPTNO's IN WHICH DEPTNO SUM OF SALARY IS LESS THAN 10000?

```
SQL> SELECT DEPTNO,SUM(SAL) FROM EMP
  2  GROUP BY DEPTNO HAVING SUM(SAL)<10000;
```

ALL CLAUSES IN A SINGLE SELECT QUERY:

```
=====================================
SYNTAX:
========
SELECT <COLUMN NAME1>,.............................,<AGGREGATIVE FUNCTION
NAME1>,.........
FROM <TABLE NAME> [ WHERE <FILTERING CONDITION>
                   GROUP BY <COLUMN NAME1>,...........
                   HAVING <FILTERING CONDITION>
                   ORDER BY <COLUMN NAME1><ASC/DESC>,...........
                   ];

EX:
SQL> SELECT DEPTNO,COUNT(DEPTNO) FROM EMP
  2  WHERE SAL>1000
  3  GROUP BY DEPTNO
  4  HAVING COUNT(DEPTNO)>3
  5  ORDER BY DEPTNO;

    DEPTNO     COUNT(DEPTNO)
----------            -------------
        20                   4
        30                   5

EXECUTION ORDER OF CLAUSES:
============================
       > FROM
              > WHERE
                     > GROUP BY
                                 > HAVING
                                        > ORDER BY

ROLLUP & CUBE:
===============
       - TO FIND OUT SUB TOTAL & GRAND TOTAL.
              ROLLUP : BASED ON SINGLE COLUMN WISE.
              CUBE     : BASED ON MULTIPLE COLUMNS WISE.
       - THESE CLAUSES CAN USE ALONG WITH GROUP BY CLAUSE ONLY.

SYNTAX FOR ROLLUP:
===================
SELECT <COLUMN NAME1>,.................................,<AGGREGATIVE
FUNCTION NAME1>,.......
FROM <TN> GROUP BY ROLLUP(<COLUMN NAME1>,<COLUMN
NAME2>,.........................);

EX:
SQL> SELECT DEPTNO,COUNT(DEPTNO) FROM EMP
  2  GROUP BY ROLLUP(DEPTNO);

    DEPTNO     COUNT(DEPTNO)
----------            -------------
        10                   3
        20                   5
        30                   6
```

```
EX:
SQL> SELECT DEPTNO,JOB,COUNT(DEPTNO) FROM EMP
  2  GROUP BY ROLLUP(DEPTNO,JOB);

EX:
SQL> SELECT JOB,DEPTNO,COUNT(JOB) FROM EMP
  2  GROUP BY ROLLUP(JOB,DEPTNO);

SYNTAX FOR CUBE:
==================
SELECT <COLUMN NAME1>,.................................,<AGGREGATIVE
FUNCTION NAME1>,.......
FROM <TN> GROUP BY CUBE(<COLUMN NAME1>,<COLUMN
NAME2>,.......................);

EX:
SQL> SELECT DEPTNO,COUNT(*) FROM EMP GROUP BY CUBE(DEPTNO)
         ORDER BY DEPTNO;

SQL> SELECT DEPTNO,JOB,COUNT(DEPTNO) FROM EMP
         GROUP BY CUBE(DEPTNO,JOB) ORDER BY DEPTNO;

NOTE:
======
       - ROLLUP AND CUBE CLAUSES ARE MOSTLY USING IN DATAWAREHOUSE
(BIG DATA) FOR DATA SUMMERIZED.

GROUPING_ID():
=============
       - IT IS A PRE-DEFINED FUNCTION IN ORACLE.
       - IS USED TO MORE COMPACT WAY TO IDENTIFY SUB TOTAL ROWS AND
GRAND TOTAL ROW FROM THE RESULT SET.

SYNTAX:
=======
       GROUPING_ID(<COLUMN NAME1>,<COLUMN NAME2>,.................);

       ID NUMBER
       ==========
              1      - THE FIRST GROUPING COLUMN SUB TOTAL ROWS
              2      - THE SECOND GROUPING COLUMN SUB TOTAL ROWS
              3      - GRAND TOTAL ROW


EX:
SQL> SELECT DEPTNO,JOB,COUNT(DEPTNO),GROUPING_ID(DEPTNO,JOB) FROM EMP
          GROUP BY CUBE(DEPTNO,JOB) ORDER BY DEPTNO;
================================================================================
====
```

```
JOINS:
======
EQUI JOIN / INNER JOIN:
=====================
EX:
WAQ TO RETRIEVAL STUDENT AND THEIR CORRESPONDING COURSE DETAILS?
NON-ANSI:
=========
SELECT * FROM STUDENT,COURSE WHERE STUDENT.CID=COURSE.CID;
                      (OR)
SELECT * FROM STUDENT S,COURSE C WHERE S.CID=C.CID;

ANSI:
=====
SELECT * FROM STUDENT INNER JOIN COURSE ON STUDENT.CID=COURSE.CID;
                      (OR)
SELECT * FROM STUDENT S INNER JOIN COURSE C ON S.CID=C.CID;

RULE OF JOINS:
==============
 - A ROW IN A TABLE IS COMPARING WITH ALL ROWS OF ANOTHER TABLE.

EX:
WAQ TO RETRIEVAL STUDENT,COURSE DETAILS WHO ARE SELECTED "ORACLE" COURSE?
ANSI:
SQL> SELECT STID,SNAME,CNAME FROM STUDENT S INNER JOIN COURSE C
  2  ON S.CID=C.CID AND CNAME='ORACLE';

NON-ANSI:
SQL> SELECT STID,SNAME,CNAME FROM STUDENT S , COURSE C
  2   WHERE S.CID=C.CID AND CNAME='ORACLE';

EX:
WAQ TO DISPLAY EMPLOYEES WHO ARE WORKING IN CHICAGO LOCATION?
ANSI:
SQL> SELECT ENAME,E.DEPTNO,D.DEPTNO,LOC FROM EMP E INNER JOIN
  2  DEPT D ON E.DEPTNO=D.DEPTNO AND LOC='CHICAGO';

NON-ANSI:
SQL> SELECT ENAME,E.DEPTNO,D.DEPTNO,LOC FROM EMP E ,
  2  DEPT D  WHERE E.DEPTNO=D.DEPTNO AND LOC='CHICAGO';

EX:
WAQ TO DISPLAY SUM OF SALARY OF EACH DEPARTMENT?
ANSI:
SQL> SELECT DNAME,SUM(SAL) FROM EMP E INNER JOIN DEPT D
  2  ON E.DEPTNO=D.DEPTNO GROUP BY DNAME;

NON-ANSI:
SQL> SELECT DNAME,SUM(SAL) FROM EMP E , DEPT D
```

```
  2  WHERE E.DEPTNO=D.DEPTNO GROUP BY DNAME;

DNAME                SUM(SAL)
--------------               ----------
RESEARCH             10875
SALES                9400
ACCOUNTING          8750
```

EX:
WAQ TO DISPLAY DEPARTMENTS IN WHICH DEPARTMENT SUM OF SALARY IS MORE THAN 10000?
ANSI:

```
SQL> SELECT DNAME,SUM(SAL) FROM EMP E INNER JOIN DEPT D
     ON E.DEPTNO=D.DEPTNO GROUP BY DNAME HAVING SUM(SAL)>10000;
```

NON-ANSI:

```
SQL> SELECT DNAME,SUM(SAL) FROM EMP E , DEPT D
  2  WHERE E.DEPTNO=D.DEPTNO GROUP BY DNAME HAVING SUM(SAL)>10000;

DNAME                SUM(SAL)
--------------               ----------
RESEARCH             10875
```

OUTER JOINS:
============
- THESE OUTER JOINS ARE USED TO RETRIEVAL MATCHING AND ALSO UNMATCHING ROWS FROM A TABLE.
- THESE ARE 3 TYPES.

i) LEFT OUTER JOIN:
==================
- MATCHING ROWS FROM BOTH TABLES BUT UNMATCHING ROWS FROM LEFT SIDE TABLE ONLY.

ANSI:
=====

```
SQL> SELECT * FROM STUDENT S LEFT OUTER JOIN COURSE C ON S.CID=C.CID;
```

NON-ANSI:
=========
- WHEN WE WRITE OUTER JOINS STATEMENTS IN NON-ANSI FORMAT THEN WE SHOULD USE A JOIN OPERATOR IS (+).

```
SQL> SELECT * FROM STUDENT S,COURSE C WHERE S.CID=C.CID(+);
```

ii) RIGHT OUTER JOIN:
==================
- MATCHING ROWS FROM BOTH TABLES BUT UNMATCHING ROWS FROM RIGHT SIDE TABLE ONLY.

ANSI:

```
SQL> SELECT * FROM STUDENT S RIGHT OUTER JOIN COURSE C ON S.CID=C.CID;
```

NON-ANSI:

```
SQL> SELECT * FROM STUDENT S,COURSE C WHERE S.CID(+)=C.CID;

iii) FULL OUTER JOIN:
====================
       - IT IS COMBINATION OF LEFT OUTER AND RIGHT OUTER.
       - MATCHING & UNMATCHING ROWS FROM BOTH TABLES.

ANSI:
SQL> SELECT * FROM STUDENT S FULL OUTER JOIN COURSE C ON S.CID=C.CID;

ANON-ANSI:
SQL> SELECT * FROM STUDENT S,COURSE C WHERE S.CID(+)=C.CID(+) ;
ERROR at line 1:
ORA-01468: a predicate may reference only one outer-joined table

SOLUTION:
SQL> SELECT * FROM STUDENT S,COURSE C WHERE S.CID=C.CID(+)
  2  UNION
  3  SELECT * FROM STUDENT S,COURSE C WHERE S.CID(+)=C.CID;

NON-EQUI JOIN:
==============
       - RETRIEVAL DATA FROM MULTIPLE TABLES BASED ON ANY OPERATOR
EXCEPT AN "=" OPERATOR.
       - IN THIS JOIN WE CAN USE " < , > , <= , >= , != ,
BETWEEN,AND,.......etc".

EX:
NON-ANSI:
SQL> SELECT * FROM TEST1 T1,TEST2 T2 WHERE T1.SNO > T2.SNO;

ANSI:
SQL> SELECT * FROM TEST1 T1 JOIN TEST2 T2 ON T1.SNO > T2.SNO;

EX:
WAQ TO DISPLAY EMPLOYEES WHOSE SALARY IS BETWEEN LOW SALARY AND HIGH
SALARY?

NON-ANSI:
SQL> SELECT ENAME,SAL,LOSAL,HISAL FROM EMP,SALGRADE
  2  WHERE SAL BETWEEN LOSAL AND HISAL;
              (OR)
SQL> SELECT ENAME,SAL,LOSAL,HISAL FROM EMP,SALGRADE
  2  WHERE (SAL>=LOSAL) AND (SAL<=HISAL);

ANSI:
SQL> SELECT ENAME,SAL,LOSAL,HISAL FROM EMP JOIN SALGRADE
  2  ON SAL BETWEEN LOSAL AND HISAL;
              (OR)
SQL> SELECT ENAME,SAL,LOSAL,HISAL FROM EMP JOIN SALGRADE
  2  ON (SAL>=LOSAL) AND (SAL<=HISAL);

CROSS JOIN:
===========
```

- IT IS DEFAULT JOIN. WHICH IS USED TO JOIN TWO OR MORE THAN TWO TABLES WITHOUT ANY CONDITION.

- IN CROSS JOIN, EACH ROW OF THE FIRST TABLE IS COMPARING WITH EACH ROW OF THE SECOND TABLE.FOR EXAMPLE A TABLE IS HAVING (M) NO.OF ROWS AND THE SECOND TABLE IS HAVING (N) NO.OF ROWS THEN THE RESULT IS (M X N) ROWS.

EX:
NON-ANSI:
```
SQL> SELECT * FROM STUDENT CROSS JOIN COURSE;
```

ANSI:
```
SQL> SELECT * FROM STUDENT CROSS JOIN COURSE;
```

EX:
```
SQL> SELECT * FROM ITEMS1;

       SNO     INAME                PRICE
----------     ----------           ----------
         1     PIZZA                150
         2     BURGER               85

SQL> SELECT * FROM ITEMS2;

       SNO     INAME                PRICE
----------     ----------           ----------
       101     PEPSI                25
       102     COCACOLA             20
```

ANSI:
```
SQL> SELECT I1.INAME,I1.PRICE,I2.INAME,I2.PRICE,
  2  I1.PRICE+I2.PRICE TOTAL_AMOUNT FROM
  3  ITEMS1 I1 CROSS JOIN ITEMS2 I2;
```

NON-ANSI:
```
SQL> SELECT I1.INAME,I1.PRICE,I2.INAME,I2.PRICE,
  2  I1.PRICE+I2.PRICE TOTAL_AMOUNT FROM
  3  ITEMS1 I1,ITEMS2 I2;

INAME                PRICE  INAME                PRICE  TOTAL_AMOUNT
----------           ----------    ----------           ----------
------------
PIZZA                150    PEPSI                25            175
```

NATURAL JOIN:
==========

- IT IS SIMILAR TO EQUI JOIN.BECAUSE NATURAL JOIN IS ALSO RETRIEVING
MATCHING ROWS BASED ON AN "=" OPERATOR JUST LIKE EQUI JOIN.

NATURAL JOIN	EQUI JOIN
- COMMON COLUMN NAME IS	- OPTIONAL

MANDATORY.

- JOINING CONDITION IS PREPARED BY SYSTEM IMPLICITLY.
- JOINING CONDITION PREPARED BY USER EXPLICITLY.

- TO AVOID DUPLICATE COLUMNS
- NOT AVOID DUPLICATE COLUMNS.

EX:
```
SQL> SELECT * FROM STUDENT S NATURAL JOIN COURSE C;
```

SELF JOIN:
==========
- JOINING A TABLE BY ITSELF IS CALLED AS "SELF JOIN".
- SELF JOIN CAN BE IMPLEMENTED BASED ON A SINGLE TABLE.
- WHEN WE WANT TO USE SELF JOIN WE SHOULD CREATE ALIAS NAMES ON A TABLE.ONCE WE CREATE ALIAS NAME ON A TABLE INTERNALLY SYSTEM IS PPREPARING VIRTUAL TABLE COPY OF EACH ALIAS NAME.
- WE CAN CREATE ANY NO.OF ALIAS NAMES ON A SINGLE TABLE BUT EACH ALIAS NAME SHOULD BE DIFFERENT.
- WITHOUT ALIAS NAMES WE CANNOT IMPLEMENT SELF JOIN MECHANISM.

PURPOSE OF SELF JOIN:
=====================
CASE-1. COMPARING A SINGLE COLUMN VALUES BY ITSELF WITH IN THE TABLE.
CASE-2. COMPARING TWO DIFFERENT COLUMNS VALUES TO EACH OTHER WITH IN THE TABLE.

EX.ON CASE-1:
=============
WAQ TO DISPLAY EMPLOYEE WHO ARE WORKING IN SAME LOCATION THE EMPLOYEE "SMITH" IS ALSO WORKING?

```
SQL> SELECT * FROM TEST;

ENAME      LOC
----------     ----------
SMITH      HYD
JONES     MUMBAI
WARD       HYD
MILLER    CHENNAI
```

SOLUTION:
==========
```
SQL> SELECT T1.ENAME,T1.LOC FROM TEST T1,TEST T2
         WHERE T1.LOC=T2.LOC AND T2.ENAME='SMITH';

ENAME      LOC
----------     ----------
SMITH      HYD
WARD       HYD
```

EX:
WAQ TO DISPLAY EMPLOYEES WHOSE SALARY IS SAME AS THE EMPLOYEE "SCOTT"
SALARY?
NON-ANSI:
========
```
SQL> SELECT E1.ENAME,E1.SAL FROM EMP E1,EMP E2
    WHERE E1.SAL=E2.SAL AND E2.ENAME='SCOTT';
```

ANSI:
=====
```
SQL> SELECT E1.ENAME,E1.SAL FROM EMP E1 JOIN EMP E2
    ON E1.SAL=E2.SAL AND E2.ENAME='SCOTT';

ENAME                SAL
----------           ----------
SCOTT                3000
FORD                 3000
```

EX.ON CASE-2:
=============
WAQ TO DISPLAY MANAGERS AND THEIR EMPLOYEES?
```
SQL> SELECT M.ENAME MANAGERS,E.ENAME EMPLOYEES
  2  FROM EMP E,EMP M WHERE M.EMPNO=E.MGR;
```

EX:
WAQ TO DISPLAY EMPLOYEES WHO ARE JOINED BEFORE THEIR MANAGER?
```
SQL> SELECT E.ENAME EMPLOYEES,E.HIREDATE E_DOJ,
  2  M.ENAME MANAGER,M.HIREDATE M_DOJ FROM
  3  EMP E,EMP M WHERE M.EMPNO=E.MGR
  4  AND E.HIREDATE < M.HIREDATE;
```

EX:
WAQ TO DISPLAY EMPLOYEES WHOSE SALARY IS MORE THAN THEIR MANAGER SALARY?
```
SQL> SELECT E.ENAME EMPLOYEES,E.SAL E_SALARY,
  2  M.ENAME MANAGERS,M.SAL M_SALARY FROM
  3  EMP E,EMP M WHERE M.EMPNO=E.MGR
  4  AND E.SAL > M.SAL;
```

HOW TO JOIN MORE THAN TWO TABLES:
==================================
SYNTAX FOR NON-ANSI JOINS:
==========================
```
SELECT * FROM <TN1>,<TN2>,<TN3>,<TN4>,.............WHERE <JOINING
CONDITION1> AND
<JOINING CONDITION2> AND <JOINING CONDITION3> AND .................;
```

SYNTAX FOR ANSI JOINS:
======================
```
SELECT * FROM <TN1> <JOIN KEY> <TN2> ON <JOINING CONDITION1>
<JOIN KEY> <TN3> ON <JOINING CONDITION2>
<JOIN KEY> <TN4> ON <JOINING CONDITION3>
.....................................................................
.....................................................................
<JOIN KEY> <TN n> ON <JOINING CONDITION n-1>;
```

EQUI JOIN:
==========
NON-ANSI FORMAT:

```
SQL> SELECT * FROM STUDENT S,COURSE C,REGISTER R
  2  WHERE S.CID=C.CID AND C.CID=R.CID;
```

ANSI FORMAT:

```
SQL> SELECT * FROM STUDENT S INNER JOIN COURSE C
  2  ON S.CID=C.CID INNER JOIN REGISTER R ON C.CID=R.CID;
```

DATAINTEGRITY:
==============

- TO MAINTAIN ACCURATE AND CONSISTENCY DATA IN DB TABLES.
  1. DECLARATIVE INTEGRITY
     > CAN IMPLEMENTING BY USING "CONSTRAINTS"(SQL)
  2. PROCEDURAL INTEGRITY
     > CAN IMPLEMENTING BY USING "TRIGGERS"(PL/SQL)

1. DECLARATIVE INTEGRITY:
=========================

i) ENTITY INTEGRITY
ii) REFERENCIAL INTEGRITY
iii) DOMAIN INTEGRITY

i) ENTITY INTEGRITY:
====================

- IT ENSURE THAT EACH ROW IN A TABLE SHOULD BE UNIQUE IDENTITY.
- IT CAN BE IMPLEMENTED BY USING "UNIQUE , PRIMARY KEY" CONSTRAINTS.

ii) REFERENCIAL INTEGRITY:
==========================

- TO MAKING RELATIONSHIPS BETWEEN TABLES.
- BY USING "FOREIGN KEY" CONSTRAINT.

iii) DOMAIN INTEGRITY:
======================

- TO CHECK VALUES WITH USER DEFINED CONDITION BEFORE ACCEPTING INTO A COLUMN.
- IT CAN BE IMPLEMENTED BY USING "CHECK,NOT NULL,DEFAULT" CONSTRAINTS.

CONSTRAINTS:
============

- ARE USED TO RESTRICTED / ENFORCE UNWANTED DATA FROM DB TABLES.
- THESE ARE 6 TYPES,
  > UNIQUE

```
               > NOT NULL
               > CHECK
               > PRIMARY KEY
               > FOREIGN KEY / REFERENCES
               > DEFAULT
       - ALL DATABASES ARE SUPPORTING TWO TYPES OF METHODS TO DEFINED
CONSTRAINTS ON A TABLE.

i) COLUMN LEVEL:
================
       - TO DEFINE A CONSTRAINT ON EACH COLUMN WISE.

SYNTAX:
=======
CREATE TABLE <TABLE NAME>
(
<COLUMN NAME1> <DATATYPE>[SIZE] <CONSTRAINT TYPE>,
<COLUMN NAME2> <DATATYPE>[SIZE] <CONSTRAINT TYPE>,
.....................................................................
.....................,
.....................................................................
.....................
);

ii) TABLE LEVEL:
==============
       - TO DEFINE A CONSTRAINT AFTER ALL COLUMNS ARE DESIGNED.
       i.e END OF THE TABLE.

SYNTAX:
=======
CREATE TABLE <TABLE NAME>
(
<COLUMN NAME1> <DATATYPE>[SIZE],
<COLUMN NAME2> <DATATYPE>[SIZE],
.....................................................,
.....................................................,
<CONSTRAINT TYPE>(<COLUMN NAME1>,.........)
);

UNIQUE:
========
       - TO RESTRCITED DUPLICATE VALUES.
       - BUT ALLOWED "NULLS".
EX:
COLUMN LEVEL:
==============
SQL> CREATE TABLE TEST1
  2  (
  3  SNO INT UNIQUE,
  4  SNAME VARCHAR2(10) UNIQUE
  5  );

TESTING:
```

```
SQL> INSERT INTO TEST1 VALUES(1,'A');----------------ALLOWED
SQL> INSERT INTO TEST1 VALUES(1,'A');---------------NOT ALLOWED
SQL> INSERT INTO TEST1 VALUES(NULL,NULL);-----ALLOWED
SQL> INSERT INTO TEST1 VALUES(2,'B');-------ALLOWED

TABLE LEVEL:
============
SQL> CREATE TABLE TEST2
  2  (
  3  SNO INT,
  4  SNAME VARCHAR2(10),
  5  UNIQUE(SNO,SNAME) -----------------> COMPOSITE UNIQUE CONSTRAINT
  6  );

TESTING:
=========
SQL> INSERT INTO TEST2 VALUES(1,'A');------ALLOWED
SQL> INSERT INTO TEST2 VALUES(1,'A');------NOT ALLOWED
SQL> INSERT INTO TEST2 VALUES(2,'A');------ALLOWED

NOT NULL:
=========
       - TO RESTRICTED NULLS INTO A COLUMN.
       - IT CAN DEFINED AT COLUMN LEVEL ONLY.
       - BUT ALLOWED DULICATE VALUES.

EX:
SQL> CREATE TABLE TEST3(SNO INT NOT NULL,SNAME VARCHAR2(10) NOT NULL);

TESTING:
SQL> INSERT INTO TEST3 VALUES(1,'A');----------ALLOWED
SQL> INSERT INTO TEST3 VALUES(1,'A');---------ALLOWED
SQL> INSERT INTO TEST3 VALUES(NULL,NULL);----NOT ALLOWED

HOW TO DEFINED MULTIPLE CONSTRAINTS ON A COLUMN:
================================================
EX:
SQL> CREATE TABLE TEST4(EID INT UNIQUE NOT NULL,
ENAME VARCHAR2(10) UNIQUE NOT NULL);

TESTING:
SQL> INSERT INTO TEST4 VALUES(101,'SMITH');----ALLOWED
SQL> INSERT INTO TEST4 VALUES(101,'SMITH');-----NO
SQL> INSERT INTO TEST4 VALUES(NULL,NULL);-----NO

CHECK:
=======
       - TO CHECK VALUES WITH USER DEFINED CONDITION ON A COLUMN.

EX:
COLUMN LEVEL:
==============
SQL> CREATE TABLE TEST5(ENAME VARCHAR2(10),SAL NUMBER(10)
CHECK(SAL>10000));
```

```
TESTING:
SQL> INSERT INTO TEST5 VALUES('SMITH',8500);------NOT ALLOWED
SQL> INSERT INTO TEST5 VALUES('SMITH',12000);----ALLOWED

TABLE LEVEL:
============
SQL> CREATE TABLE TEST6(ENAME VARCHAR2(10),SAL NUMBER(10),
          CHECK(ENAME=LOWER(ENAME) AND SAL>=8000));

TESTING:
SQL> INSERT INTO TEST6 VALUES('ALLEN',10000);----NOT ALLOWED
SQL> INSERT INTO TEST6 VALUES('allen',7500);----NOT ALLOWED
SQL> INSERT INTO TEST6 VALUES('allen',10000);-----ALLOWED

PRIMARY KEY:
============
       - IT IS A COMBINATION OF UNIQUE AND NOT NULL.
       - IT NOT ALLOWED DUPLICATE VALUES AND NULLS.
       - A TABLE IS HAVING ONLY ONE PRIMARY KEY CONSTRAINT.

EX:
COLUMN LEVEL:
==============
SQL> CREATE TABLE PRODUCT(PCODE INT PRIMARY KEY,
          PNAME VARCHAR2(10));

TESTING:
SQL> INSERT INTO PRODUCT VALUES(1021,'C');--------ALLOWED
SQL> INSERT INTO PRODUCT VALUES(1021,'C++');------NO
SQL> INSERT INTO PRODUCT VALUES(NULL,'C++');------NO

TABLE LEVEL:
============
SQL> CREATE TABLE BRANCH(BCODE INT,
  2  BNAME VARCHAR2(10),
  3  BLOC VARCHAR2(10),
  4  PRIMARY KEY(BCODE,BNAME));---------COMPOSITE PRIMARY KEY CONSTRAINT

TESTING:
SQL> INSERT INTO BRANCH VALUES(1021,'SBI','SRNAGAR');------ALLOWED
SQL> INSERT INTO BRANCH VALUES(1021,'SBI','MADHAPUR');-----NO
SQL> INSERT INTO BRANCH VALUES(1022,'SBI','MADHAPUR');----ALLOWED

FOREIGN KEY(REFERENCES)
=========================
       - TO CREATE RELATIONSHIP BETWEEN TABLES.
       - BY USING RELATIONSHIP WE CAN TAKE REFERENCIAL(IDENTITY) DATA FROM
ONE TABLE TO ANOTHER TABLE IN DATABASE.

BASIC RULES:
============
1. COMMON COLUMN NAME(OPTIONAL) IN BOTH TABLES.
```

2. COMMON COLUMN DATATYPES MUST BE MATCH.(MANDATORY)

3. ONE TABLE SHOULD CONTAIN PRIMARY KEY AND ANOTHER TABLE SHOULD CONTAIN FOREIGN KEY BUT
PK AND FK COLUMN MUST BE COMMON COLUMN.

4. A PRIMARY KEY CONSTRAINT TABLE IS CALLED AS "PARENT" TABLE AND A FOREIGN KEY CONSTRAINT
TABLE IS CALLED AS "CHILD" TABLE.

5. A FOREIGN KEY COLUMN IS ALLOWED THE VALUES THOSE VALUES MUST BE IN PRIMARY KEY COLUMN.

```
PARENT          CHILD
TEST1           TEST2
======          ======
SNO(PK)         SNO(FK)
========              ========
1               1
2               1
3               2
                2
                3
                3
                3
                4----------ERROR
                NULL-----ALLOWED(ORPHAN)
```

6. BY DEFAULT FOREIGN KEY IS ALLOWED DUPLICATE AND NULLS.

EX:

```
SQL> CREATE TABLE DEPT1(DEPTNO INT PRIMARY KEY,DNAME
VARCHAR2(10));--PARENT

SQL> INSERT INTO DEPT1 VALUES(10,'SALES');
SQL> INSERT INTO DEPT1 VALUES(20,'PRODUCTION');

SQL> CREATE TABLE EMP1(EID INT UNIQUE NOT NULL,
  2  ENAME VARCHAR2(10),DEPTNO INT REFERENCES
  3  DEPT1(DEPTNO));---CHILD

SQL> INSERT INTO EMP1 VALUES(1021,'SMITH',10);
SQL> INSERT INTO EMP1 VALUES(1022,'ALLEN',10);
SQL> INSERT INTO EMP1 VALUES(1023,'WARD',20);
.....................................................................
.......;
```

NOTE:
=====

- ONCE WE CREATED RELATIONSHIP BETWEEN TABLES THERE ARE TWO RULES ARE COME INTO PICTURE.

RULE-1(INSERTION):

==========

- WE CANNOT INSERT VALUES INTO FOREIGN KEY COLUMN IN CHILD TABLE THOSE VALUES ARE NOT EXISTING IN PRIMARY KEY COLUMN OF PARENT TABLE.

NO PARENT = NO CHILD

EX:

```
SQL> INSERT INTO EMP1 VALUES(1025,'SCOTT',30);
ERROR at line 1:
ORA-02291: integrity constraint (MYDB4PM.SYS_C007550) violated - parent
key not found
```

RULE-2(DELETION)

==========

- WE CANNOT DELETE A ROW FROM PARENT TABLE THOSE ROWS ARE HAVING THE CORRESPONDING CHILD ROWS IN CHILD TABLE WITHOUT ADDRESSING TO THE CHILD.

EX:

```
SQL> DELETE FROM DEPT1 WHERE DEPTNO=10
ERROR at line 1:
ORA-02292: integrity constraint (MYDB4PM.SYS_C007550) violated - child
record found
```

HOW TO ADDRESS TO CHILD TABLE:

==========

> CASCADE RULES :
>
> 1. ON DELETE CASCADE
> 2. ON DELETE SET NULL

1. ON DELETE CASCADE:

==========

- ONCE WE DELETE A ROW FROM PARENT TABLE THEN THE CORRESPONDING CHILD ROWS ARE ALSO DELETED FROM CHILD TABLE AUTOMATICALLY.

EX:

```
SQL> CREATE TABLE DEPT2(DEPTNO INT PRIMARY KEY,DNAME
VARCHAR2(10));--PARENT

SQL> INSERT INTO DEPT2 VALUES(10,'SALES');
SQL> INSERT INTO DEPT2 VALUES(20,'PRODUCTION');

SQL> CREATE TABLE EMP2(EID INT UNIQUE NOT NULL,
    ENAME VARCHAR2(10),DEPTNO INT REFERENCES
    DEPT2(DEPTNO) ON DELETE CASCADE);---CHILD

SQL> INSERT INTO EMP2 VALUES(1021,'SMITH',10);
SQL> INSERT INTO EMP2 VALUES(1022,'ALLEN',10);
SQL> INSERT INTO EMP2 VALUES(1023,'WARD',20);
```

TESTING:

```
SQL> DELETE FROM DEPT2 WHERE DEPTNO=10;
1 row deleted.
```

2. ON DELETE SET NULL:

```
=====================
       - ONCE WE DELETE A ROW FROM PARENT TABLE THEN THE CORRESPONDING
CHILD ROWS VALUES(i.e FOREIGN KEY COLUMN) ARE CONVERTING INTO NULLS
IN CHILD TABLE AUTOMATICALLY.
EX:
SQL> CREATE TABLE DEPT3(DEPTNO INT PRIMARY KEY,DNAME
VARCHAR2(10));--PARENT

SQL> INSERT INTO DEPT3 VALUES(10,'SALES');
SQL> INSERT INTO DEPT3 VALUES(20,'PRODUCTION');

SQL> CREATE TABLE EMP3(EID INT UNIQUE NOT NULL,
    ENAME VARCHAR2(10),DEPTNO INT REFERENCES
    DEPT3(DEPTNO) ON DELETE SET NULL);---CHILD

SQL> INSERT INTO EMP3 VALUES(1021,'SMITH',10);
SQL> INSERT INTO EMP3 VALUES(1022,'ALLEN',10);
SQL> INSERT INTO EMP3 VALUES(1023,'WARD',20);

TESTING:
SQL> DELETE FROM DEPT3 WHERE DEPTNO=10;
1 row deleted.

DATADICTIONARY / READ ONLY TABLES:
==================================
       - WHENEVER WE ARE INSTALLING ORACLE S/W INTERNALLY SYSTEM IS
CREATING  SOME PRE-DEFINED TABLES ARE CALLED AS "DATA DICTIONARIES".
       - DATA DICTIONARIES ARE SAVED THE INFORMATION ABOUT
TABLES,CONSTRAINTS,SYNONYMS,VIEWS,INDEXES,SEQUENCES,.............ETC.
       - ON THESE DATA DICTIONARIES WE CAN PERFORM "SELECT" AND "DESC"
COMMANDS ONLY.SO THAT DATADICTIONARIES ARE CALLED AS "READ ONLY TABLES"
IN ORACLE DB.
       - IF WE WANT TO VIEW ALL DATADICTIONARIES IN ORACLE DATABASE THEN
WE
FOLLOW THE FOLLOWING SYNTAX IS:

SYNTAX:
========
       SELECT * FROM DICT; ----------(DICT IS MAIN DICTIONARY)

NOTE:
======
       - IF WE WANT TO VIEW COLUMN NAME AND CONSTRAINT NAME OF A
PARTICULAR TABLE IN ORACLE DB THEN WE USE A DATADICTIONARY IS CALLED AS
"USER_CONS_COLUMNS".

EX:
SQL> DESC USER_CONS_COLUMNS;
SQL> SELECT COLUMN_NAME,CONSTRAINT_NAME FROM USER_CONS_COLUMNS
          WHERE TABLE_NAME='TEST1';

COLUMN_NAME                          CONSTRAINT_NAME
```

```
-----------------------------------------------
--------------------------------------------------------------------------------
------
SNO                                     SYS_C007523
SNAME                                   SYS_C007524

HOW TO CREATE USER DEFINED CONSTRAINT NAME:
===========================================
SYNTAX:
========
CREATE TABLE <TABLE NAME>
(<COLUMN NAME1> <DATATYPE>[SIZE] CONSTRAINT <UD CONSTRAINT NAME>
<CONSTRAINT TYPE>,<COLUMN NAME2> <DATATYPE>[SIZE] CONSTRAINT <UD
CONSTRAINT NAME> <CONSTRAINT
TYPE>,.....................................);

EX:
SQL> CREATE TABLE TEST6
        (
          EID INT CONSTRAINT PK_EID PRIMARY KEY,
          ENAME VARCHAR2(10) CONSTRAINT  UQ_ENAME UNIQUE
        );
Table created.

SQL> SELECT COLUMN_NAME,CONSTRAINT_NAME FROM USER_CONS_COLUMNS
         WHERE TABLE_NAME='TEST6';

COLUMN_NAME                  CONSTRAINT_NAME
-----------------------------------------------
--------------------------------------------------------------------------------
------
EID                          PK_EID
ENAME                        UQ_ENAME

HOW ADD / APPLY CONSTRAINT ON EXISTING TABLE:
=============================================
SYNTAX:
=======
ALTER TABLE <TN> ADD <CONSTRAINT> <CONSTRAINT KEY NAME>
<CONSTRAINT TYPE>(COLUMN NAME);

ADDING PRIMARY KEY:
===================
EX:
SQL> CREATE TABLE PARENT(EID INT,ENAME VARCHAR2(10), SAL NUMBER(10));
Table created.

SQL> ALTER TABLE PARENT ADD CONSTRAINT EID_PK PRIMARY KEY(EID);

ADDING UNIQUE,CHECK CONSTRAINT:
===============================
SQL> ALTER TABLE PARENT ADD CONSTRAINT ENAME_UQ UNIQUE(ENAME);
SQL> ALTER TABLE PARENT ADD CONSTRAINT SAL_CHK CHECK(SAL>=10000);
```

NOTE:
======
- IF WE WANT TO VIEW A CHECK CONSTRAINT CONDITIONAL VALUE THEN WE USE A DATADICTIONARY IS "USER_CONSTRAINTS".

EX:
```
SQL> DESC USER_CONSTRAINTS;
SQL> SELECT SEARCH_CONDITION FROM USER_CONSTRAINTS
          WHERE TABLE_NAME='PARENT';

SEARCH_CONDITION
--------------------------------------------------------------------------------
------
SAL>=10000
```

HOW TO APPLY "NOT NULL" CONSTRAINT:
==========
SYNTAX:
=======
ALTER TABLE <TN> MODIFY <COLUMN NAME> <CONSTRAINT> <CONSTRAINT KEY NAME> NOT NULL;

EX:
```
SQL> ALTER TABLE PARENT MODIFY ENAME CONSTRAINT ENAME_NN NOT NULL;
```

HOW TO ADD A FOREIGN KEY CONSTRAINT TO AN EXISTING TABLE:
==========
SYNTAX:
=======
ALTER TABLE <TN> ADD CONSTRAINT <CONSTRAINT KEY NAME>
FOREIGN KEY(COMMON COLUMN OF CHILD TABLE) REFERENCES
<PARENT TABLE NAME>(COMMON COLUMN OF PARENT TABLE)
ON DELETE CASCADE / ON DELETE SET NULL;

EX:
```
SQL> CREATE TABLE CHILD(DNAME VARCHAR2(10),EID INT);
Table created.

SQL> ALTER TABLE CHILD ADD CONSTRAINT EID_FK
  2  FOREIGN KEY(EID) REFERENCES PARENT(EID)
  3  ON DELETE CASCADE;
```

HOW TO DROP CONSTRAINT FROM AN EXISTING TABLE:
==========
SYNTAX:
========
ALTER TABLE <TN> DROP CONSTRAINT <CONSTRAINT KEY NAME>;

TO DROP A PRIMARY KEY:
==========
CASE-1 : WITHOUT RELATIONSHIP:
==========
```
SQL> ALTER TABLE PARENT DROP CONSTRAINT EID_PK;
```

CASE-2: WITH RELATIONSHIP:
==========
SQL> ALTER TABLE PARENT DROP CONSTRAINT EID_PK CASCADE;

TO DROP UNIQUE,CHECK,NOT NULL CONSTRAINT:
==========
EX:
SQL> ALTER TABLE PARENT DROP CONSTRAINT ENAME_UQ;
SQL> ALTER TABLE PARENT DROP CONSTRAINT SAL_CHK;
SQL> ALTER TABLE PARENT DROP CONSTRAINT ENAME_NN;

HOW TO RENAME A CONSTRAINT NAME:
==========
SYNTAX:
========
ALTER TABLE <TN> RENAME <CONSTRAINT> <OLD CONSTRAINT KEY NAME> TO
<NEW CONSTRAINT KEY NAME>;

EX:
SQL> CREATE TABLE TEST9(SNO INT PRIMARY KEY);
Table created.

SQL> SELECT COLUMN_NAME,CONSTRAINT_NAME FROM
         USER_CONS_COLUMNS WHERE TABLE_NAME='TEST9';

COLUMN_NAME   CONSTRAINT_NAME
---------- ----------
SNO   SYS_C007566

SQL> ALTER TABLE TEST9 RENAME CONSTRAINT SYS_C007566 TO SNO_PK;

COLUMN_NAME   CONSTRAINT_NAME
---------- ----------
SNO   SNO_PK

HOW TO DISABLE / ENABLE A CONSTRAINT:
==========
SYNTAX:
========
ALTER TABLE <TN> DISABLE / ENABLE CONSTRAINT <CONSTRAINT NAME>;

NOTE:
=====
       - BY DEFAULT ALL CONSTRAINTS ARE ENABLE(WORKING) MODE.IF WE WANT
TO DISABLE A CONSTRAINT TEMPORARLY THEN WE USE "DISABLE" KEYWORD.

DISABLE A CONSTRAINT:
==========
EX:
SQL> CREATE TABLE TEST10(ENAME VARCHAR2(10),

```
  2  SAL NUMBER(10) CHECK(SAL>8000));

TESTING:
SQL> INSERT INTO TEST10 VALUES('ALLEN',8001);----ALLOWED

SQL> INSERT INTO TEST10 VALUES('ALLEN',7000);---NOT ALLOWED.
ERROR at line 1:
ORA-02290: check constraint (MYDB4PM.SYS_C007567) violated

       - IF WE WANT TO INSERT SALARY WHICH IS LESS THAN TO 8000 THEN WE
MUST
DISBALE A CHECK CONSTRAINT TEMPORARLY.

EX:
ALTER TABLE TEST10 DISABLE CONSTRAINT SYS_C007567;

TESTING:
SQL> INSERT INTO TEST10 VALUES('ALLEN',7000);------ALLOWED.

ENABLE A CONSTRAINT:
====================
EX:
SQL> ALTER TABLE TEST10 ENABLE CONSTRAINT SYS_C007567
ERROR at line 1:
ORA-02293: cannot validate (MYDB4PM.SYS_C007567) - check constraint
violated

       - TO OVERCOME THE ABOVE PROBLEM AND ENABLE A CHECK CONSTRAINT
ON SALARY COLUMN THEN WE USE A KEY WORD IS "NOVALIDATE".WHEN WE USE
"NOVALIDATE" KEYWORD ORACLE SERVER IS NOT CHECKING AN EXISTING VALAUES
OF A SALARY COLUMN IN TABLE BUT CHECKING NEWLY INSERTING  DATA/VALUES INTO
A SALARY COLUMN.

EX:
SQL> ALTER TABLE TEST10 ENABLE NOVALIDATE CONSTRAINT SYS_C007567;

TESTING:
SQL> INSERT INTO TEST10 VALUES('ADAMS',5500);----NOT ALLOWED
SQL> INSERT INTO TEST10 VALUES('ADAMS',9500);----ALLOWED

DEFAULT CONSTRAINT:
====================
       - IT IS SPECIAL TYPE OF CONSTRAINT WHICH IS USED TO ASSIGN  USER
DEFINED
VALUE TO A PARTICULAR COLUMN.

EX:
SQL> CREATE TABLE TEST11(ENAME VARCHAR2(10),SAL NUMBER(10) DEFAULT 8000);
Table created.

TESTING:
SQL> INSERT INTO TEST11 VALUES('ALLEN',25000);
SQL> INSERT INTO TEST11(ENAME)VALUES('SCOTT');
```

```
SQL> SELECT * FROM TEST11;

ENAME                SAL
----------           ----------
ALLEN                25000
SCOTT                8000

HOW TO ADD A DEFAULT VALUE TO AN EXISTING TABLE COLUMN:
=======================================================
SYNTAX:
========
ALTER TABLE <TN> MODIFY <COLUMN NAME> DEFAULT <VALUE>;

EX:
SQL> CREATE TABLE TEST12(SNO INT,SFEE NUMBER(10));
SQL> ALTER TABLE TEST12 MODIFY SFEE DEFAULT 500;

TESTING:
SQL> INSERT INTO TEST12 VALUES(1,15000);
SQL> INSERT INTO TEST12(SNO) VALUES(2);
SQL> SELECT * FROM TEST12;

NOTE:
=====
       TO VIEW A DEFAULT VALUE OF A COLUMN OF PARTICULAR TABLE THEN WE USE
A DATADICTIONARY IS CALLED AS "USER_TAB_COLUMNS".

EX:
SQL> DESC USER_TAB_COLUMNS;
SQL> SELECT COLUMN_NAME,DATA_DEFAULT FROM USER_TAB_COLUMNS
  2  WHERE TABLE_NAME='TEST12';

COLUMN_NAME          DATA_DEFAULT
-----------------------------------------
-------------------------------------
SFEE                 500

HOW TO REMOVE DEFAULT VALUE FROM A COLUMN:
==========================================
SYNTAX:
========
ALTER TABLE <TN> MODIFY <COLUMN NAME> DEFAULT <VALUE>;

EX:
SQL> ALTER TABLE TEST12 MODIFY SFEE DEFAULT NULL;

TESTING:
SQL> INSERT INTO TEST12(SNO)VALUES(3);

COLUMN_NAME          DATA_DEFAULT
-----------------------------------------
-------------------------------------
SFEE                 NULL
```

TRANSACTION CONTROL LANGUAGE(TCL):
==========
1) COMMIT
2) ROLLBACK
3) SAVEPOINT

1) COMMIT:
==========
- TO MAKE A TRANSACTION IS PERMANENT.
- ORACLE DB SUPPORTING THE FOLLOWING TWO TYPES OF COMMIT TRANSACTIONS THOSE ARE,
I) IMPLICIT COMMIT TRANSACTIONS
II) EXPLICIT COMMIT TRANSACTIONS

I) IMPLICIT COMMIT TRANSACTIONS:
==========
- THESE TRANSACTIONS ARE COMMITTED BY SYSTEM AUTOMATICALLY.
EX: DDL COMMANDS

II) EXPLICIT COMMIT TRANSACTIONS:
==========
- THESE TRANSACTIONS ARE COMMITTED BY USER AS PER THEIR REQUIREMENT.
- WHEN A USER WANT TO COMMIT A TRANSACTION THEN WE SHOULD USE "COMMIT" COMMAND.
EX: DML COMMANDS

SYNTAX:
========
COMMIT;

EX:
```
SQL> CREATE TABLE BRANCH(BCODE INT,BNAME VARCHAR2(10),BLOC VARCHAR2(10));

SQL> INSERT INTO BRANCH VALUES(1021,'SBI','HYD');
SQL> COMMIT;

SQL> UPDATE BRANCH SET BLOC='PUNE' WHERE BCODE=1021;
SQL> COMMIT;

SQL> DELETE FROM BRANCH WHERE BCODE=1021;
SQL> COMMIT;
```
(OR)
```
SQL> INSERT INTO BRANCH VALUES(1021,'SBI','HYD');
SQL> UPDATE BRANCH SET BLOC='PUNE' WHERE BCODE=1021;
SQL> DELETE FROM BRANCH WHERE BCODE=1021;
SQL> COMMIT;
```

II) ROLLBACK:
==========
- IS USED TO CANCEL A TRANSACTION.
- IT IS SIMILAR TO "UNDO" OPERATION IN FILE.

- ONCE WE COMMIT A TRANSACTION THEN WE CANNOT ROLLBACK.

SYNTAX:
========

```
ROLLBACK;
```

EX:

```
SQL> INSERT INTO BRANCH VALUES(1022,'ICICI','MUMBAI');
SQL> ROLLBACK;

SQL> UPDATE BRANCH SET BLOC='PUNE' WHERE BCODE=1022;
SQL> ROLLBACK;

SQL> DELETE FROM BRANCH WHERE BCODE=1022;
SQL> ROLLBACK;
```

(OR)

```
SQL> INSERT INTO BRANCH VALUES(1022,'ICICI','MUMBAI');
SQL> UPDATE BRANCH SET BLOC='PUNE' WHERE BCODE=1022;
SQL> DELETE FROM BRANCH WHERE BCODE=1022;
SQL> ROLLBACK;
```

SAVEPOINT:
===========

- WHEN WE CREATE A SAVE POINTER INTERNALLY SYSTEM IS ALLOCATING SOME SPECIAL MEMORY TO A POINTER FOR STORING ROW / ROWS WHICH WE WANT TO ROLLBACK IN THE FEATURE.

SYNTAX TO CREATE A SAVEPOINTER:
================================

```
SAVEPOINT <POINTER NAME>;
```

SYNTAX TO ROLLBACK A SAVEPOINT:
================================

```
ROLLBACK TO <POINTER NAME>;
```

EX1:

```
SQL> DELETE FROM BRANCH WHERE BCODE=1021;
SQL> DELETE FROM BRANCH WHERE BCODE=1025;

SQL> SAVEPOINT S1;
Savepoint created.
SQL> DELETE FROM BRANCH WHERE BCODE=1023;
```

CASE-1:
======

```
SQL> ROLLBACK TO S1;--------1023 ROW ONLY CANCELED
```

CASE-2:
=======

```
SQL> COMMIT / ROLLBACK;
```

EX2:

```
SQL> DELETE FROM BRANCH WHERE BCODE=1021;

SQL> SAVEPOINT S1;
```

Savepoint created.
SQL> DELETE FROM BRANCH WHERE BCODE IN(1023,1025);

CASE-1:
=======
ROLLBACK TO S1;--------> 1023,1025 ROWS CANCELLED

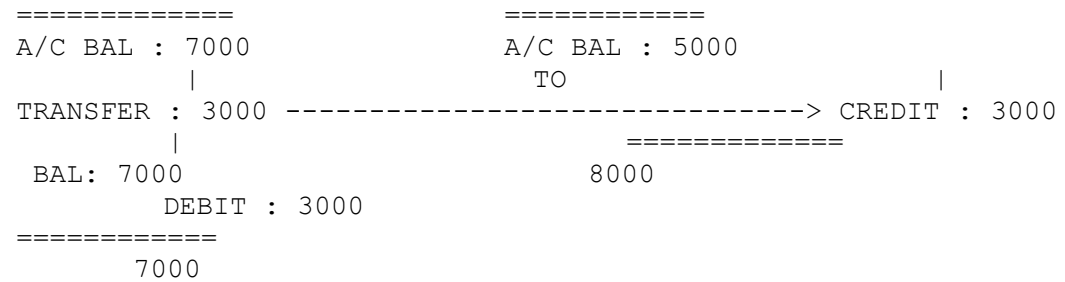
CASE-2:
=======
COMMIT / ROLLBACK;

NOTE:
=====
- ALL DATABASE MANAGEMENT SYSTEMS ARE MAINTAIN "ACID" PROPERTIES BY DEFAULT.BY USING THESE "ACID" PROPERTIES DATABASE WILL MAINTAIN "ACCURATE
AND CONSISTENCY" DATA / INFORMATION OF A PARTICULAR ORGANIZATION.

A - AUTOMICITY
C - CONSISTENCY
I - ISOLATION
D - DURABILITY

AUTOMICITY:
===========
- AUTOMIC - SINGLE
WITHDRAW TRANSACTION:
=======================
> INSERT ATM
> SELECT LANG
> SELECT BANKING
> CLICK WITHDRAW
> ENTER AMOUNT :4000
> SELECT CURR/SAVE
> ENTER PIN
> YES / NO

CONSISTENCY:
==============
- PROVIDE DATA ACCURATE.



ISOLATION:
=========

- EVERY TRANSACTION IS INDEPENDENT.

DURABILITY:
==========
- ONCE A TRANSACTION IS COMMIT.THEN WE CANNOT ROLLBACK.

SUBQUERY:
==========
- A QUERY INSIDE ANOTHER QUERY IS CALLED AS "SUBQUERY".
- SUBQUERY IS ALSO CALLED AS "NESTED QUERY".
- A SUBQUERY STATEMENT IS HAVING TWO MORE QUERIES THOSE ARE,
  I) INNER QUERY / CHILD QUERY / SUB QUERY
  II) OUTER QUERY / PARENT QUERY / MAIN QUERY
- AS PER THE EXECUTION PROCESS SUBQUERY STATEMENT IS CLASSIFIED INTO
  TWO WAYS,
  1. NON-CORELATED SUBQUERY:
  ==========
  FIRST : INNER QUERY
  LATER : OUTER QUERY

  2. CO-RELATED SUBQUERY:
  ==========
  FIRST : OUTER QUERY
  LATER : INNER QUERY

SYNTAX:
========
SELECT * FROM <TN> WHERE <CONDITION>(SELECT * FROM......(SELECT * FROM ....));

1. NON-CORELATED SUBQUERY:
==========
- i) SINGLE ROW SUBQUERY
- ii) MULTIPLE ROW SUBQUERY
- iii) MULTIPLE COLUMN SUBQUERY
- iv) INLINE VIEW

i) SINGLE ROW SUBQUERY:
==========
- WHENEVER A SUBQUERY RETURN A SINGLE VALUE.
- CAN USE THE FOLLOWING OPERATORS ARE :
  " = , < , > , <= , >= , !="

EX:
WAQ T DISPLAY EMPLOYEES DETAILS WHO ARE EARNING THE FIRST HIGHEST SALARY?
==========
||SUBQUERY = (OUTER QUERY + INNER QUERY); ||
==========

```
STEP1: INNER QUERY:
===================
SQL> SELECT MAX(SAL) FROM EMP;

STEP2: OUTER QUERY:
====================
SQL> SELECT * FROM EMP WHERE SAL=(INNER QEURY);

STEP3: SUBQUERY =(OUTER QUERY + INNER QUERY)
=============================================
SQL> SELECT * FROM EMP WHERE SAL=(SELECT MAX(SAL) FROM EMP);

EX:
WAQ TO DISPLAY THE SENIOR MOST EMPLOYEE DETAILS FROM EMP TABLE?
SQL> SELECT * FROM EMP WHERE HIREDATE=(SELECT MIN(HIREDATE) FROM EMP);

EX:
WAQ TO DISPLAY EMPLOYEES WHOSE JOB IS SAME AS THE EMPLOYEE "SMITH" JOB?
SQL> SELECT * FROM EMP WHERE JOB=(SELECT JOB FROM EMP WHERE
ENAME='SMITH');

EX:
WAQ TO DISPLAY EMPLOYEES WHOSE SALARY IS MORE THAN THE MAXIMUM
SALARY OF "SALESMAN"?
SQL> SELECT * FROM EMP WHERE SAL >(SELECT MAX(SAL) FROM EMP
         WHERE JOB='SALESMAN');

EX:
WAQ TO FIND OUT THE SECOND HIGHEST SALARY FROM EMP TABLE?
SQL> SELECT MAX(SAL) FROM EMP WHERE SAL<(SELECT MAX(SAL) FROM EMP);

EX:
WAQ TO DISPLAY EMPLOYEES WHO ARE EARNING THE SECOND HIGHEST SALARY FROM
EMP TABLE?
SQL> SELECT * FROM EMP WHERE SAL=
(SELECT MAX(SAL) FROM EMP WHERE SAL<
(SELECT MAX(SAL) FROM EMP));

EX:
WAQ TO DISPLAY EMPLOYEES WHO ARE EARNING THE 3RD HIGHEST SALARY FROM EMP
TABLE?
SQL> SELECT * FROM EMP WHERE SAL=
  2  (SELECT MAX(SAL) FROM EMP WHERE SAL<
  3  (SELECT MAX(SAL) FROM EMP WHERE SAL<
  4  (SELECT MAX(SAL) FROM EMP)));

              Nth    N+1
              ===    ===
              1ST    2Q
              2ND    3Q
              3RD    4Q
```

30TH 31Q

150TH 151Q

HOW TO OVERCOME THE ABOVE PROBLEM?
==========
MULTIPLE ROW SUBQUERY:
==========
- WHEN A SUBQUERY RETURN MORE THAN ONE VALUE IS CALLED AS "MRSQ".
- WE CAN USE THE FOLLOWING OPERATORS ARE "IN,ANY,ALL".

EX:
WAQ TO DISPLAY EMPLOYEES WHOSE EMPLOYEE JOB IS SAME AS THE EMPLOYEE "SMITH" OR "MARTIN" JOBS?

```
SQL> SELECT * FROM EMP WHERE JOB IN(SELECT JOB FROM EMP
  2  WHERE ENAME='SMITH' OR ENAME='MARTIN');
                    (OR)
SQL> SELECT * FROM EMP WHERE JOB IN(SELECT JOB FROM EMP
  2  WHERE ENAME IN('SMITH','MARTIN'));
```

EX:
WAQ TO DISPLAY EMPLOYEES WHO ARE EARNING MINIMUM AND MAXIMUM SALARY FROM EMP TABLE?

```
SQL> SELECT * FROM EMP WHERE SAL IN
  2  (SELECT MIN(SAL) FROM EMP
  3  UNION
  4  SELECT MAX(SAL) FROM EMP
  5  );
```

EX:
WAQ TO DISPLAY EMPLOYEES WHO ARE GETTING MAXIMUM SALARY FROM EACH JOB WISE?

```
SQL> SELECT * FROM EMP WHERE SAL IN(SELECT MAX(SAL) FROM EMP
         GROUP BY JOB);
```

EX:
WAQ TO DISPLAY THE SENIOR MOST EMPLOYEES FROM EACH DEPTNO WISE?

```
SQL> SELECT * FROM EMP WHERE HIREDATE IN(SELECT MIN(HIREDATE)
  2  FROM EMP GROUP BY DEPTNO);
```

WORKING WITH "ANY" , "ALL" OPERATORS:
==========
ANY :
=====
- IT RETURNS "TRUE" IF ANY ONE VALUE IS SATISFIED IN THE LIST WITH THE GIVEN VALUE.

EX:
i) X > ANY(10,20,30)
IF X=40 ------> TRUE

```
              IF X=08 ------> FALSE
              IF X=25 ------> TRUE

ALL:
====
       - IT RETURNS "TRUE" IF ALL VALUES ARE SATISFIED IN THE LIST WITH
THE GIVEN VALUE.

       EX:
       i) X > ALL(10,20,30)
              IF X=40 ------> TRUE
              IF X=08 ------> FALSE
              IF X=25 ------> FALSE

EX:
WAQ TO DISPLAY EMPLOYEES WHOSE SALARY IS MORE THAN ANY ONE "SALESMAN"
SALARY?
SQL> SELECT * FROM EMP WHERE SAL >ANY(SELECT SAL FROM EMP
          WHERE JOB='SALESMAN');

EX:
WAQ TO DISPLAY EMPLOYEES WHOSE SALARY IS MORE THAN ALL "SALESMAN"
SALARIES?
SQL> SELECT * FROM EMP WHERE SAL >ALL(SELECT SAL FROM EMP
          WHERE JOB='SALESMAN');

UPDATING DATA IN EMP TABLE:
===========================
SQL> UPDATE EMP SET SAL=1300 WHERE EMPNO=7902;

EX:
WAQ TO DISPLAY EMPLOYEES WHO ARE GETTING MAXIMUM SALARY FROM EACH JOB
WISE?
SQL> SELECT * FROM EMP WHERE SAL IN(SELECT MAX(SAL) FROM EMP
         GROUP BY JOB);

OUTPUT:
=======
ENAME          JOB                  SAL
---------- -          --------             ----------
ALLEN          SALESMAN             1600
JONES          MANAGER              2975
SCOTT          ANALYST              3000
KING                  PRESIDENT             6000
FORD                  ANALYST               1300
MILLER         CLERK                1300

NOTE:
=====
       - WHEN WE ARE COMPARING THE GORUP OF VALUES BY USING MULTIPLE ROW
SUBQUERY THEN ORACLE RETURNS THE WRONG RESULT.TO OVERCOME THE ABOVE
PROBLEM WE USE "MULTIPLE COLUMN SUBQUERY" MECHANISM.

MULTIPLE COLUMN SUBQUERY:
```

==========

- COMPARING MULTIPLE COLUMNS VALUES OF INNER QUERY WITH MULTIPLE COLUMNS VALUES OF OUTER QUERY IS CALLED AS "MCSQ".

SYNTAX:
========

```
SELECT * FROM <TN> WHERE (<COLUMN NAME1>,<COLUMN
NAME2>,...................)
IN(SELECT <COLUMN NAME1>,<COLUMN NAME2>,.................FROM
<TN>..........);
```

EX:
WAQ TO DISPLAY EMPLOYEES WHO ARE GETTING MAXIMUM SALARY FROM EACH JOB WISE?

SOLUTION:

```
SQL> SELECT * FROM EMP WHERE(JOB,SAL)
  2  IN(SELECT JOB,MAX(SAL) FROM EMP
  3  GROUP BY JOB);
```

OUTPUT:
=======

```
ENAME          JOB                  SAL
-----------          --------              ----------
ALLEN          SALESMAN             1600
JONES          MANAGER              2975
SCOTT          ANALYST              3000
KING                  PRESIDENT            6000
MILLER         CLERK                1300
```

EX:
WAQ TO DISPLAY EMPLOYEES WHOSE EMPLOYEE JOB,MGR ARE SAME AS THE JOB,MGR OF THE EMPLOYEE "SCOTT"?

```
SQL> SELECT * FROM EMP WHERE(JOB,MGR) IN(SELECT JOB,MGR
  2  FROM EMP WHERE ENAME='SCOTT');
```

PSEUDO COLUMNS:
==========

- THESE COLUMNS ARE WORKING JUST LIKE A TABLE COLUMNS.
  1) ROWID
  2) ROWNUM

1) ROWID:
========

- WHENEVER WE INSERT A NEW ROW INTO A TANLE INTERNALLY SYSTEM WILL GENERATE A "UNIQUE ROW IDENTIFICATION ADDRESS" FOR EACH ROW WISE.
- THESE ROWID's ARE STORED IN DATABASE SO THAT THESE ARE PERMANENT ID's IN ORACLE DB.

EX:

```
SQL> SELECT ROWID,ENAME FROM EMP;
SQL> SELECT ROWID,ENAME,JOB FROM EMP WHERE JOB='MANAGER';
```

```
SQL> SELECT MIN(ROWID) FROM EMP;
SQL> SELECT MAX(ROWID) FROM EMP;
```

HOW TO DELETE MULTIPLE DUPLICATE ROWS EXCEPT ONE DUPLICATE ROW FROM A TABLE:

EX:

```
SQL> SELECT * FROM TEST;

       SNO NAME
---------- ----------
         1 A
         1 A
         1 A
         2 B
         3 C
         3 C
         4 D
         4 D
         4 D
         5 E
         5 E
```

SOLUTION:

```
SQL> DELETE FROM TEST WHERE ROWID NOT IN(SELECT MAX(ROWID)
          FROM TEST GROUP BY SNO);
```

OUTPUT:

```
SQL> SELECT * FROM TEST;

       SNO NAME
---------- ----------
         1 A
         2 B
         3 C
         4 D
         5 E
```

2) ROWNUM:

- IT IS USED TO GENERATE ROW NUMBERS FOR EACH ROW WISE (OR) FOR EACH GROUP OF ROWS WISE AUTOMATICALLY.
- THESE ROW NUMBERS ARE NOT SAVED IN DB.SO THAT THESE ARE TEMPORARY NUMBERS.
- BY USING ROWNUM PSEUDO COLUMN WE CAN PERFORM "TOP n" AND "Nth" ROW OPERATIONS ON TABLE DATA.

EX:

```
SELECT ROWNUM,ENAME FROM EMP;
SELECT ROWNUM,ENAME,JOB FROM EMP WHERE JOB='MANAGER';
```

EX:
WAQ TO FETCH THE 1ST ROW DETAILS FROM EMP TABLE BY USING ROWNUM?
SQL> SELECT * FROM EMP WHERE ROWNUM=1;

EX:
WAQ TO FETCH THE 2ND ROW DETAILS FROM EMP TABLE BY USING ROWNUM?
SQL> SELECT * FROM EMP WHERE ROWNUM=2;
no rows selected

NOTE:
======
- GENERALLY ROWNUM IS ALWAYS STARTS WITH "1" FOR EVERY SELECTED ROW FROM A TABLE.SO TO OVERCOME THIS PROBLEM WE SHOULD USE THE FOLLOWING OPERATORS ARE "< , < = ".

SOLUTION:
==========
```
SQL> SELECT * FROM EMP WHERE ROWNUM<=2
  2  MINUS
  3  SELECT * FROM EMP WHERE ROWNUM=1;
```

EX:
WAQ TO FETCH TOP 5 ROWS FROM EMP TABLE BY USING ROWNUM?
```
SQL> SELECT * FROM EMP WHERE ROWNUM<=5;
```

EX:
WAQ TO FETCH 5TH POSITION ROW FROM EMP TABLE BY USING ROWNUM?
```
SQL> SELECT * FROM EMP WHERE ROWNUM<=5
  2  MINUS
  3  SELECT * FROM EMP WHERE ROWNUM<=4;
```

EX:
WAQ TO FETCH FROM 3RD POSITION ROW TO 10TH POSITION ROW FROM EMP TABLE BY USING ROWNUM?
```
SQL> SELECT * FROM EMP WHERE ROWNUM<=10
  2  MINUS
  3  SELECT * FROM EMP WHERE ROWNUM<3;
```

EX:
WAQ TO FETCH THE LAST TWO ROWS FROM EMP TABLE BY USING ROWNUM?
```
SQL> SELECT * FROM EMP WHERE ROWNUM<=14
  2  MINUS
  3  SELECT * FROM EMP WHERE ROWNUM<=12;
```

(OR)

```
SQL> SELECT * FROM EMP
  2  MINUS
  3  SELECT * FROM EMP WHERE ROWNUM<=(SELECT COUNT(*)-2 FROM EMP);
```

INLINE VIEW:
==========

- PROVIDING A SELECT QUERY INPLACE OF A TABLE NAME IN SELECT STATEMENT IS CALLED AS "INLINE VIEW".

(OR)

- PROVIDING A SELECT QUERY IN FROM CLAUSE IN SELECT STATEMENT IS CALLED AS "INLINE VIEW".

SYNTAX:
========
SELECT * FROM (<SELECT QUERY>);------INLINE VIEW

NOTE:
=====
- IN THIS INLINE VIEW MECHANISM THE RESULT OF INNER QUERY WILL ACT AS
A TABLE FOR OUTER QUERY.

CASE1:
======
- GENERALLY SUBQUERY IS NOT ALLOWED "ORDER BY" CLAUSE.SO THAT TO OVERCOME THIS WE NEED "INLINE VIEW".

CASE-2:
======
- GENERALLY COLUMN ALIAS NAMES ARE NOT ALLOWED UNDER "WHERE CLAUSE" CONDITION.SO TO OVERCOME THIS PROBLEM WE NEED "INLINE VIEW".

EX:
WAQ TO DISPLAY EMPLOYEES WHOSE ANNUAL SALARY IS MORE THAN 25000?
```
SQL> SELECT * FROM(SELECT EMPNO,ENAME,SAL,SAL*12 ANNSAL FROM EMP)
          WHERE ANNSAL>25000;
```

HOW TO USE "ROWNUM" ALIAS NAME UNDER WHERE CLAUSE:
==========
EX:
WAQ TO FETCH 5TH POSITION ROW FROM EMP TABLE BY USING ROWNUM ALIAS NAME ALONG WITH INLINE VIEW?
```
SQL> SELECT * FROM(SELECT ROWNUM R,EMPNO,ENAME FROM EMP) WHERE R=5;
```
(OR)
```
SQL> SELECT * FROM(SELECT ROWNUM R,EMP.* FROM EMP) WHERE R=5;
```

EX:
WAQ TO FETCH 3RD,6TH,9TH,12TH ROWS FROM EMP TABLE BY USING ROWNUM ALIAS NAME ALONG WITH INLINE VIEW?
```
SQL> SELECT * FROM(SELECT ROWNUM R,EMP.* FROM EMP) WHERE R IN(3,6,9,12);
```

EX:
WAQ TO FETCH EVEN POSITION ROWS FROM EMP TABLE BY USING ROWNUM ALIAS NAME ALONG WITH INLINE VIEW?
```
SQL> SELECT * FROM(SELECT ROWNUM R,EMP.* FROM EMP) WHERE MOD(R,2)=0;
```

EX:
WAQ TO FETCH THE 1ST AND THE LAST ROW FROM EMP TABLE BY USING ROWNUM ALIAS NAME ALONG WITH INLINE VIEW?
```
SQL> SELECT * FROM(SELECT ROWNUM R,EMP.* FROM EMP) WHERE R=1 OR R=14;
```

```
                        (OR)
SQL> SELECT * FROM(SELECT ROWNUM R,EMP.* FROM EMP) WHERE R IN(1,14);
                        (OR)
SQL> SELECT * FROM(SELECT ROWNUM R,EMP.* FROM EMP)
         WHERE R=1 OR R=(SELECT COUNT(*) FROM EMP);
                        (OR)
SQL> SELECT * FROM(SELECT ROWNUM R,EMP.* FROM EMP)
         WHERE R IN(1,(SELECT COUNT(*) FROM EMP));
```

USING "ORDER BY" CLAUSE IN SUBQUERY:
==========

EX:
WAQ TO FETCH THE FIRST FIVE HIGHEST SALARIES EMPLOYEE ROWS FROM EMP TABLE BY USING ROWNUM ALONG WITH INLINE VIEW?

```
SQL> SELECT * FROM(SELECT * FROM EMP ORDER BY SAL DESC) WHERE ROWNUM<=5;
```

EX:
WAQ TO FETCH THE 5TH HIGHEST SALARY ROW FROM EMP TABLE BY USING ROWNUM ALONG WITH INLINE VIEW?

```
SQL> SELECT * FROM(SELECT * FROM EMP ORDER BY SAL DESC) WHERE ROWNUM<=5
          MINUS
          SELECT * FROM(SELECT * FROM EMP ORDER BY SAL DESC) WHERE
ROWNUM<=4;
```

ANALYTICAL FUNCTIONS:
==========

1. ROW_NUMBER()
2. RANK()
3. DENSE_RANK()

- THE ABOVE FUNCTIONS ARE USED TO GENERATE RANKING NUMBERS FOR EACH ROW / FOR EACH GROUP OF ROWS WISE AUTOMATICALLY EXCEPT ROW_NUMBER(). THIS ROW_NUMBER() IS USED TO GENERATE SEQUENCE NUMBERS TO EACH ROW / TO EACH GROUP OF ROWS WISE.

- RANK(),DENSE_RANK() ARE ASSIGNING SAME RANK NUMBERS TO SAME VALUE BUT RANK() WILL SKIP THE NEXT RANK NUMBER IN THE ORDER WHEREAS DENSE_RANK()
WILL NOT SKIP THE NEXT RANK NUMBER IN THE ORDER.

EX:

ENAME	SALARY	ROW_NUMBER()	RANK()	DENSE_RANK()
A	85000	1	1	1
B	72000	2	2	2
C	72000	3	2	2
D	68000	4	4	3
E	55000	5	5	4
F	46000	6	6	5

SYNTAX:
==========

ANALYTICAL FUNCTION NAME() OVER([PARTITION BY <COLUMN NAME>] ORDER BY <COLUMN NAME> <ASC/DESC>)

Here,
PARTITION BY CLAUSE ----------> OPTIONAL
ORDER BY CLAUSE ----------> MANDATORY

WITHOUT PARTITION BY CLAUSE:
==========
EX:
```
SQL> SELECT EMPNO,ENAME,SAL,ROW_NUMBER()
        OVER(ORDER BY SAL DESC) ROWNUMBERS FROM EMP;

SQL> SELECT EMPNO,ENAME,SAL,RANK()
         OVER(ORDER BY SAL DESC) ROWNUMBERS FROM EMP;

SQL> SELECT EMPNO,ENAME,SAL,DENSE_RANK()
        OVER(ORDER BY SAL DESC) ROWNUMBERS FROM EMP;
```

WITH PARTITION BY CLAUSE:
==========
EX:
```
SQL> SELECT EMPNO,ENAME,DEPTNO,SAL,
  2  ROW_NUMBER()OVER(PARTITION BY DEPTNO ORDER BY SAL DESC)
  3  ROWNUMBERS FROM EMP;

SQL> SELECT EMPNO,ENAME,DEPTNO,SAL,
  2  RANK()OVER(PARTITION BY DEPTNO ORDER BY SAL DESC)
  3  RANKS FROM EMP;

SQL> SELECT EMPNO,ENAME,DEPTNO,SAL,
         DENSE_RANK()OVER(PARTITION BY DEPTNO ORDER BY SAL DESC)
         RANKS FROM EMP
```

EX:
WAQ TO DISPLAY 4TH HIGHEST SALARY EMPLOYEE FROM EACH DEPTNO WISE BY USING DENSE_RANK() ALONG INLINE VIEW?
```
SQL> SELECT * FROM(SELECT EMPNO,ENAME,DEPTNO,SAL,
  2           DENSE_RANK()OVER(PARTITION BY DEPTNO ORDER BY SAL DESC)
  3           RANKS FROM EMP) WHERE RANKS=4;
```

EX:
WAQ TO DISPLAY 3RD SENIOR MOST EMPLOYEE FROM EACH JOB WISE BY USING DENSE_RANK() ALONG INLINE VIEW?
```
SQL> SELECT * FROM(SELECT EMPNO,ENAME,JOB,HIREDATE,
  2  DENSE_RANK()OVER(PARTITION BY JOB ORDER BY HIREDATE)
  3  RANKS FROM EMP) WHERE RANKS=3;
```

2. CO-RELATED SUBQUERY:
==========
- IN THIS MECHANISM FIRST OUTER QUERY IS EXECUTED AND LATER INNER QUERY WILL EXECUTE.

SYNTAX TO FIND OUT "Nth" HIGH / LOW SALARY :
==========
```
SELECT * FROM <TABLE NAME> <TABLE ALIAS NAME1> WHERE N-1=(SELECT
COUNT(DISTINCT <COLUMN NAME>) FROM <TABLE NAME> <TABLE ALIAS NAME2>
```

```
WHERE <TABLE ALIAS NAME2>.<COLUMN NAME> < / > <TABLE ALIAS NAME1>.<COLUMN
NAME>);

              < ------------ FINDING LOWEST SALARY
              > ------------ FINDING HIGHEST SALARY

EX:
WAQ TO FIND OUT THE FIRST HIGHEST SALARY EMPLOYEE DETAILS?
SQL> SELECT * FROM EMPLOYEE E1 WHERE 0=(SELECT
         COUNT(DISTINCT SAL) FROM EMPLOYEE E2
         WHERE E2.SAL > E1.SAL);

SOL:
====
       IF N=1
              N-1 ====> 1-1 ====> 0

EX:
WAQ TO FIND OUT THE 4TH HIGHEST SALARY EMPLOYEE DETAILS?
SQL> SELECT * FROM EMPLOYEE E1 WHERE 3=(SELECT
         COUNT(DISTINCT SAL) FROM EMPLOYEE E2
         WHERE E2.SAL > E1.SAL);

SOL:
====
       IF N=4
              N-1 ====> 4-1 ====> 3


EX:
WAQ TO FIND OUT THE FIRST LOWEST SALARY EMPLOYEE DETAILS?
SQL> SELECT * FROM EMPLOYEE E1 WHERE 0=(SELECT
         COUNT(DISTINCT SAL) FROM EMPLOYEE E2
         WHERE E2.SAL < E1.SAL);

SOL:
====
       IF N=1
              N-1 ====> 1-1 ====> 0


SYNTAX TO DISPLAY "TOP n" HIGH / LOW SALARIES :
==============================================
SELECT * FROM <TABLE NAME> <TABLE ALIAS NAME1> WHERE N>(SELECT
COUNT(DISTINCT <COLUMN NAME>) FROM <TABLE NAME> <TABLE ALIAS NAME2>
WHERE <TABLE ALIAS NAME2>.<COLUMN NAME> < / > <TABLE ALIAS NAME1>.<COLUMN
NAME>);

EX:
WAQ TO DISPLAY TOP 3 HIGHEST SALARIES EMPLOYEE DETAILS FROM EMP TABLE?
SQL> SELECT * FROM EMPLOYEE E1 WHERE 3>(SELECT
  2  COUNT(DISTINCT SAL) FROM EMPLOYEE E2
  3  WHERE E2.SAL > E1.SAL);
```

EX:
WAQ TO DISPLAY TOP 3 LOWEST SALARIES EMPLOYEE DETAILS FROM EMP TABLE?

```
SQL> SELECT * FROM EMPLOYEE E1 WHERE 3>(SELECT
    COUNT(DISTINCT SAL) FROM EMPLOYEE E2
    WHERE E2.SAL < E1.SAL);
```

NOTE:
======
> TO FIND OUT "Nth" HIGH / LOW SALARY --------------> N-1
> TO DISPLAY "TOP n" HIGH / LOW SALARIES ----------> N>

EXISTS OPERATOR:
==========

- IT IS A SPECIAL TYPE OF OPERATOR WHICH IS USED IN CO-RELATED SUBQUERY ONLY. BY USING "EXISTS" OPERATOR WE CAN CHECK A REQUIRED ROW/ROWS ARE EXISTING IN A TABLE OR NOT.
- IF A ROW IS EXISTS IN TABLE THEN RETURNS "TRUE".
- IF A ROW IS NOT EXISTS IN TABLE THEN RETURNS "FALSE".

SYNTAX:
=======

```
WHERE EXISTS(<SELECT QUERY>);
```

EX:
WAQ TO DISPLAY DEPARTMENT DETAILS IN WHICH DEPARTMENT THE EMPLOYEES ARE WORKING?

```
SQL> SELECT * FROM DEPT D WHERE EXISTS(SELECT DEPTNO FROM EMP
  2  E WHERE E.DEPTNO=D.DEPTNO);
```

EX:
WAQ TO DISPLAY DEPARTMENT DETAILS IN WHICH DEPARTMENT THE EMPLOYEES ARE NOT WORKING?

```
SQL> SELECT * FROM DEPT D WHERE NOT EXISTS(SELECT DEPTNO FROM EMP
       E WHERE E.DEPTNO=D.DEPTNO);
```

SCALAR SUBQUERY:
==========

- PROVIDING SUBQUERY INPLACE OF COLUMNS IN A SELECT STATEMENT.
  (OR)
- PROVIDING SUBQUERIES IN A SELECT CLAUSE IS CALLED AS "SCALAR SUBQUERY".
- EVERY SCALAR SUBQUERY RESULT WILL SHOW AS A COLUMN.

SYNTAX:
========

```
SELECT  (<SUBQUERY1>),(<SUBQUERY2>),.....  FROM <TN>;
```

EX:

```
SQL> SELECT(SELECT COUNT(*) FROM DEPT) NO_OF_ROWS,(SELECT COUNT(*) FROM
EMP)          NO_OF_ROWS FROM DUAL;

NO_OF_ROWS     NO_OF_ROWS
----------          ----------
```

```
         4           14

EX:
SQL> SELECT (SELECT SUM(SAL) FROM EMP WHERE DEPTNO=10) "10",
  2  (SELECT SUM(SAL) FROM EMP WHERE DEPTNO=20) "20",
  3  (SELECT SUM(SAL) FROM EMP WHERE DEPTNO=30) "30" FROM DUAL;

        10           20           30
----------   ----------   ----------
      9750        10875         9400
```

DB SECURITY:
============

- ALL DATABASES ARE SUPPORTING THE FOLLOWING TWO SECURITY MECHANISMS THOSE ARE,
  1. AUTHENTICATION
  2. AUTHORIZATION

1. AUTHENTICATION:
==================

- TO VERIFY USER CREDENTIAL(USERNAME & PASSWORD) BEFORE LOGIN INTO SYSTEM / ORACLE SERVER.
- THESE USER CREDENTIALS ARE CREATED BY "DBA" ONLY.

SYNTAX TO CREATE NEW USERNAME & PASSWORD:
==

```
CREATE USER <USER NAME> IDENTIFIED BY <PASSWORD>;
```

EX:

```
CREATE USER U1 IDENTIFIED BY U1;
```

2. AUTHORIZATION:
=================

- TO GIVE PERMISSIONS TO USERS TO PERFORM OPERATIONS OVER ORACLE DB.
- THESE PERMISSIONS ARE GIVING BY "DBA" ONLY.
- BY USING "DCL" COMMANDS DBA WILL GIVE AUTHORIZATION PERMISSIONS TO USERS.

DATA CONTROL LANGUAGE(DCL):
============================

- GRANT:
  - > GIVING PERMISSIONS TO USER.

  SYNTAX:
  ========

  ```
  GRANT <PRIVILEGE NAME> TO <USER NAME>;
  ```

- REVOKE:
  - > TO CANCEL PERMISSIONS OF USER.

  SYNTAX:
  ========

  ```
  REVOKE <PRIVILEGE NAME> FROM <USER NAME>;
  ```

PRIVILEGE:
==========
- IT IS A RIGHT / PERMISSION GIVING TO USERS.
- ORACLE SUPPORTS THE FOLLOWING TWO TYPES OF PRIVILEGES ARE,
1) SYSTEM PRIVILEGES
2) OBJECT PRIVILEGES

1) SYSTEM PRIVILEGES:
====================
- THESE PRIVILEGES ARE GIVING BY "DBA" ONLY.
- ARE CONNECT,CREATE TABLE,UNLIMITED TABLESPACE,CREATE VIEW, CREATE SYNONYM,CREATE MATERILIZED VIEW,CREATE SEQUENCE,CREATE INDEX,....etc

SYNTAX:
=======
GRANT <SYSTEM PRIVILEGE NAME> TO <USERNAME>;

EX:
```
SQL> CONN SYSTEM/TIGER
SQL> CREATE USER U1 IDENTIFIED BY  U1;

SQL> CONN U1/U1
ERROR.

SQL> CONN SYSTEM/TIGER
SQL> GRANT CONNECT TO U1;

SQL> CONN U1/U1
SQL> CREATE TABLE TEST1(SNO INT);
ERROR.

SQL> CONN SYSTEM/TIGER
SQL> GRANT CREATE TABLE TO U1;

SQL> CONN U1/U1
SQL> CREATE TABLE TEST1(SNO INT);
TABLE CREATED.
SQL> INSERT INTO TEST1 VALUES(1);
ERROR

SQL> CONN SYSTEM/TIGER
SQL> GRANT UNLIMITED TABLESPACE TO U1;

SQL> CONN U1/U1
SQL> INSERT INTO TEST1 VALUES(1);----ALLOWED

SQL> CREATE SYNONYM S1 FOR TEST1;
ERROR
SQL> CREATE VIEW V1 AS SELECT * FROM TEST1;
ERROR

SQL> CONN SYSTEM/TIGER
SQL> GRANT CREATE SYNONYM,CREATE VIEW TO U1;
```

```
SQL> CONN U1/U1
SQL> CREATE SYNONYM S1 FOR TEST1;-----ALLOWED
SQL> CREATE VIEW V1 AS SELECT * FROM TEST1;------ALLOWED

HOW TO CANCEL "CONNECT" PERMISSION OF A USER:
==============================================
SYNTAX:
========
REVOKE <SYSTEM PRIVILEGE NAME> FROM <USERNAME>;

EX:
SQL> CONN SYSTEM/TIGER
SQL> REVOKE CONNECT FROM U1;

SQL> CONN U1/U1
ERROR.

2) OBJECT PRIVILEGES:
======================
       - THESE PRIVILEGES ARE GIVING BY "DBA" AND ALSO "USER".
       - THESE PRIVILEGE ARE 4 TYPES :
              - SELECT , INSERT ,UPDATE , DELETE (OR) "ALL" KEYWORD.

SYNTAX:
=======
GRANT <OBJECT PRIVILEGE NAME> ON <OBJECT NAME> TO <USERNAME>;

CASE-1: DBA  TO  USER:
======================
EX:
SQL> CONN U1/U1
SQL> SELECT * FROM DEPT;------------------------ERROR
SQL> SELECT * FROM SYSTEM.DEPT;---------ERROR

SQL> CONN SYSTEM/TIGER
SQL> GRANT SELECT ON DEPT TO U1;

SQL> CONN U1/U1
SQL> SELECT * FROM DEPT;---------------ERROR
SQL> SELECT * FROM SYSTEM.DEPT;-------ALLOWED

SQL> INSERT INTO SYSTEM.DEPT VALUES(50,'DBA','HYD');-------ERROR
SQL> UPDATE SYSTEM.DEPT SET LOC='PUNE' WHERE DEPTNO=30;--------ERROR
SQL> DELETE FROM SYSTEM.DEPT WHERE DEPTNO=10;--------ERROR

SQL> CONN SYSTEM/TIGER
SQL> GRANT INSERT,UPDATE,DELETE ON DEPT TO U1;

SQL> CONN U1/U1
SQL> INSERT INTO SYSTEM.DEPT VALUES(50,'DBA','HYD');-------ALLOWED
SQL> UPDATE SYSTEM.DEPT SET LOC='PUNE' WHERE DEPTNO=30;--------ALLOWED
SQL> DELETE FROM SYSTEM.DEPT WHERE DEPTNO=10;--------ALLOWED
```

HOW TO CANCEL OBJECT PRIVILEGES OF A USER:
==========

SYNTAX:
========

REVOKE <OBJECT PRIVILEGE NAME> ON <OBJECT NAME> FROM <USERNAME>;

EX:

```
SQL> CONN
Enter user-name: system/tiger

SQL> REVOKE SELECT,INSERT,UPDATE,DELETE ON DEPT FROM U1;
               (OR)
SQL> REVOKE ALL ON DEPT FROM U1;
```

CASE-2 : USER TO USER:
==========

EX:

```
SQL> CONN SYSTEM/TIGER
SQL> GRANT SELECT ON DEPT TO U1;

SQL> CONN U1/U1
SQL> SELECT * FROM SYSTEM.DEPT;------ALLOWED

SQL> GRANT SELECT ON SYSTEM.DEPT TO U2;
ERROR at line 1:
ORA-01031: insufficient privileges
```

NOTE:
=====

- WHEN A USER(U1) WANT TO GIVE OBJECT PRIVILEGE PERMISSIONS TO ANOTHER USER(U2) THEN USER U1 MUST BE TAKE PERMISSIONS FROM DBA WITH "WITH GRANT OPTION" STATEMENT THEN ONLY USER U1 CAN GIVE PERMISSION TO USER U2.

EX:

```
SQL> CONN SYSTEM/TIGER
SQL> GRANT SELECT ON DEPT TO U1 WITH GRANT OPTION;

SQL> CONN U1/U1
SQL> SELECT * FROM SYSTEM.DEPT;-----------ALLOWED
SQL> GRANT SELECT ON SYSTEM.DEPT TO U2;------ALLOWE

SQL> CONN U2/U2
SQL> SELECT * FROM SYSTEM.DEPT;------- ALLOWED
```

ROLE:
=====

- IS NOTHING BUT "SET OF PRIVILEGES" GIVING TO USERS.
- THESE ROLES ARE CREATED BY DBA ONLY.

WHY ROLE:
=========

- WHEN WE ARE WORKING ON SAME PROJECT THEN THE GROUP OF

USER REQUIRED SAME SET OF PRIVILEGES SO THAT DBA WILL CREATE A ROLE TO ASSIGN
SET OF PRIVILEGES TO USERS.
	- WHEN WE WANT TO CREATE A ROLE THEN WE FOLLOW THE FOLLOWING 3 STEPS ARE,

STEP1: CREATE A ROLE:
=====================
SYNTAX:
========
CREATE ROLE <ROLE NAME>;

STEP2: ASSIGNING SET PRIVILEGES TO A ROLE:
==
SYNTAX:
========
GRANT <PRIVILEGES> TO <ROLE NAME>;

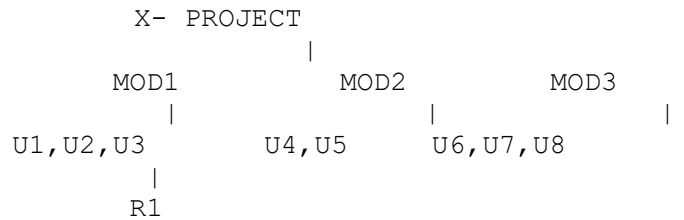
STEP3: TO ASSIGN A ROLE TO MULTIPLE USERS:
==
SYNTAX:
=======
GRANT <ROLE NAME> TO <USERS>;

EX:
SQL> CONN SYSTEM/TIGER

SQL> CREATE ROLE R1;
SQL> GRANT CONNECT,CREATE TABLE TO R1;
SQL> GRANT R1 TO U11,U12;

SQL> CONN U11/U11
CONNECTED.
SQL> CREATE TABLE T1(SNO INT);
TABLE CREATED.

SQL> CONN U12/U12
CONNECTED.
SQL> CREATE TABLE T1(SNO INT);
TABLE CREATED.
==
====
SQL:
====
	> DDL -------- WORK ON STRUCTURE OF TABLE
	> DML -------- WORK ON DATA ------------> ANY DEVELOPERS
	> DQL/DRL ------ READING DATA FROM SOURCE
	> TCL -------- MANAGING ONLINE TRANSACTIONS
	==================================
	> DCL -------- WORKS ON DB SECURITY ------------> DBA ONLY



SYNONYMS:
==========
- IT IS A DB OBJECT WHICH IS USED TO CREATE PERMANENT ALIAS NAMES FOR TBALE / DB OBJECT.

PURPOSE OF SYNONYMS:
=====================
1. TO REDUCE LENGTHY TABLE NAME.
EX:
COLLEGE_ENROLLMENT_DETAILS(TABLE NAME)

```
SQL> INSERT INTO COLLEGE_ENROLLMENT_DETAILS VALUES(......);
SQL> UPDATE COLLEGE_ENROLLMENT_DETAILS SET
..................;
SQL> DELETE FROM COLLEGE_ENROLLMENT_DETAILS.............;
SQL> SELECT * FROM COLLEGE_ENROLLMENT_DETAILS;
```

SOLUTION:
==========

```
CREATE SYNONYM S1 FOR COLLEGE_ENROLLMENT_DETAILS;

SQL> INSERT INTO S1 VALUES(......);
SQL> UPDATE S1 SET ..................;
SQL> DELETE FROM S1.............;
SQL> SELECT * FROM S1;
```

2. TO HIDE OWNERNAME AND TBALE OBJECT NAME(SECURITY)
EX:

```
SQL> CONN SYSTEM/TIGER
SQL> GRANT SELECT ON DEPT TO U1;

SQL> CONN U1/U1
SQL> SELECT * FROM SYSTEM.DEPT;
       OWNER NAME : SYSTEM
       TABLE NAME : DEPT
```

SOLUTION:
==========

```
SQL> CONN SYSTEM/TIGER
SQL> CREATE SYNONYM S2 FOR SYSTEM.DEPT;
SQL> GRANT SELECT ON S2  TO U1;
```

```
SQL> CONN U1/U1
SQL> SELECT * FROM S2;
```

TYPES OF SYNONYMS:
==========
1. PRIVATE SYNONYMS
2. PUBLIC SYNONYMS

1. PRIVATE SYNONYMS:
==========
- THESE ARE CREATED BY USERS WHO ARE HAVING PERMISSION.

SYNTAX:
=======
CREATE SYNONYM <SYNONYM NAME> FOR [OWNER NAME].<TABLE NAME>;

EX:
```
SQL> CONN SYSTEM/TIGER
SQL> CREATE USER U22 IDENTIFIED BY U22;
SQL> GRANT CONNECT,CREATE TABLE,UNLIMITED TABLESPACE TO U22;

SQL> CONN U22/U22
SQL> CREATE TABLE COLLEGE_ENROLLMENT_DETAILS(STID INT,SNAME VARCHAR2(10),
BNAME VARCHAR2(10));

SQL> INSERT INTO COLLEGE_ENROLLMENT_DETAILS VALUES(1,'SMITH','CSE');
SQL> UPDATE COLLEGE_ENROLLMENT_DETAILS SET STID=1021 WHERE STID=1;
SQL> DELETE FROM COLLEGE_ENROLLMENT_DETAILS WHERE STID=1021;

SQL> CREATE SYNONYM S1 FOR COLLEGE_ENROLLMENT_DETAILS;
ERROR at line 1:
ORA-01031: insufficient privileges

SQL> CONN SYSTEM/TIGER
SQL> GRANT CREATE SYNONYM TO U22;

SQL> CONN U22/U22
SQL> CREATE SYNONYM S1 FOR COLLEGE_ENROLLMENT_DETAILS;----ALLOWED

SQL> SELECT * FROM S1;
SQL> INSERT INTO S1 VALUES(1,'SMITH','CSE');
SQL> UPDATE S1 SET STID=101 WHERE STID=1;
SQL> DELETE FROM S1 WHERE STID=101;
```

NOTE:
======
- TO VIEW PRIVATE SYNONYMS ALONG WITH TABLE NAME,OWNER OF THE TABLE THEN WE USE A DATADICTIONARY IS "USER_SYONYMS".

EX:
```
SQL> DESC USER_SYNONYMS;
SQL> SELECT TABLE_OWNER,TABLE_NAME,SYNONYM_NAME
  2  FROM USER_SYNONYMS;
```

```
TABLE_OWNER    TABLE_NAME                    SYNONYM_NAME
-------------------------- ------------------------------
       ----------------------
U22            COLLEGE_ENROLLMENT_DETAILS    S1

HOW TO DROP A PRIVATE SYNONYM:
================================
SYNTAX:
=======
DROP SYNONYM <SYNONYM NAME>;

EX:
DROP SYONYM S1;

2. PUBLIC SYNONYMS:
====================
       - THESE ARE CREATED BY DBA ONLY.
       - TO HIDE OWNER NAME AND TABLE NAME FROM USERS.

SYNTAX:
=======
CREATE PUBLIC SYONYM <SYNONYM NAME> FOR [OWNER NAME].<TABLE NAME>;

EX:
SQL> CONN SYSTEM/TIGER
SQL> CREATE USER U33 IDENTIFIED BY U33;
SQL> CREATE USER U44 IDENTIFIED BY U44;
SQL> GRANT CONNECT TO U33,U44;
SQL> CREATE PUBLIC SYNONYM PS1 FOR SYSTEM.DEPT;
SQL> GRANT SELECT ON PS1 TO U33,U44;

SQL> CONN U33/U33
SQL> SELECT * FROM PS1;-----ALLOWED

SQL> CONN U44/U44
SQL> SELECT * FROM PS1;------ALLOWED

NOTE:
=====
       - TO VIEW PUBLIC SYNONYMS IN ORACLE DB THEN WE USE A DATADICTIONARY
IS CALLED AS "ALL_SYNONYMS".

EX:
SQL> DESC ALL_SYONYMS;
SQL> SELECT TABLE_OWNER,TABLE_NAME,SYNONYM_NAME FROM
          ALL_SYNONYMS WHERE TABLE_NAME='DEPT';

TABLE_OWNER           TABLE_NAME            SYNONYM_NAME
-------------------------------    ------------------------
-------------------------
SYSTEM                DEPT                  PS1

HOW TO DROP A PUBLIC SYNONYM:
================================
```

SYNTAX:
========
DROP PUBLIC SYONYM <SYNONYM NAME>;

EX:
SQL> DROP PUBLIC SYNONYM PS1;
==========
===
VIEWS:
======
- IT IS A VIRTUAL OBJECT WHICH WILL CREATE BASED ON BASE TABLE(MAIN TABLE). VIEW DOES NOT STORE DATA / INFORMATION.IT CAN ACCESS THE REQUIRED DATA
(OR) INFORMATION FROM BASE TABLE.

PURPOSE VIEWS:
==========
1. SECURITY:
> COLUMN LEVEL SECURITY
> ROW LEVEL SECURITY

2. TO ACCESS / ENTER DATA INTO / FORM A BASE TABLE THROUGH A VIEW INTERNALLY
DB SERVER WILL CHECK DATA INTEGRITY RULES WHICH WAS GIVEN ON BASE TABLE.

3. TO CONVERT A COMPLEX QUERY INTO SIMPLE QUERY.(VIEW CAN SAVE QUERY STATEMENT).

TYPES OF VIEWS:
==========
1. SIMPLE VIEWS
2. COMPLEX VIEWS

1. SIMPLE VIEWS:
==========
- WHEN WE ACCESS THE REQUIRED DATA FROM A SINGLE BASE TABLE.
- THROUGH SIMPLE VIEW WE CAN PERFORM INSERT,UPDATE AND DELETE OPERATIONS ON BASE TABLE.

SYNTAX:
=======
CREATE VIEW <VIEW NAME> AS <SELECT QUERY>;

EX:
CREATE A VIEW TO ACCESS DATA FROM DEPT TABLE?

SQL> CREATE VIEW V1 AS SELECT * FROM DEPT;
TESTING:
========
SQL> SELECT * FROM V1;
SQL> INSERT INTO V1 VALUES(50,'SAP','HYD');
SQL> UPDATE V1 SET LOC='PUNE' WHERE DEPTNO=50;
SQL> DELETE FROM V1 WHERE DEPTNO=50;

EX:
CREATE A VIEW TO ACCESS EMPNO,ENAME,SALARY DETAILS FROM EMP TABLE?
SQL> CREATE VIEW V2 AS SELECT EMPNO,ENAME,SAL FROM EMP;

TESTING:
SQL> INSERT INTO V2 VALUES(1122,'YUVIN',5500);-----ALLOWED

EX:
CREATE A VIEW TO DISPLAY EMPLOYEES DETAILS WHO ARE WORKING UNDER
DEPTNO IS 20?
SQL> CREATE VIEW V3 AS SELECT * FROM EMP WHERE DEPTNO=20;

VIEW OPTIONS:
==============
1. WITH CHECK OPTION
2. WITH READ ONLY

1. WITH CHECK OPTION:
=====================
- THIS STATEMENT IS USED TO RESTRICTED DATA ON BASE TABLE THROUGH
A VIEW OBJECT.

EX:
CREATE A VIEW TO DISPLAY AND ACCEPT EMPLOYEE DETAILS WHOSE SALARY IS 3000?
SQL> CREATE VIEW V5 AS SELECT * FROM EMP WHERE SAL=3000 WITH CHECK OPTION;

TESTING:
INSERT INTO V5
VALUES(1121,'BHUVIN1','HR',1524,'12-JAN-21',6000,NULL,NULL);---NO
INSERT INTO V5
VALUES(1122,'BHUVIN2','HR',1525,'10-JAN-21',2000,NULL,NULL);---NO
INSERT INTO V5
VALUES(1123,'BHUVIN3','HR',1526,'25-JAN-21',3000,NULL,NULL);---YES

2. WITH READ ONLY:
==================
- TO RESTRICTED DML OPERATIONS ON BASE TABLE THROUGH A VIEW.
EX:
SQL> CREATE VIEW V6 AS SELECT * FROM DEPT WITH READ ONLY;

TESTING:
SQL> SELECT * FROM V6;------ALLOWED

SQL> INSERT INTO V6 VALUES(50,'DBA','PUNE');
SQL> UPDATE V6 SET LOC='HYD' WHERE DEPTNO=30;
SQL> DELETE FROM V6 WHERE DEPTNO=10;
ERROR at line 1:
ORA-42399: cannot perform a DML operation on a read-only view

2. COMPLEX VIEWS:
================
- WHEN WE CREATED A VIEW BASED ON :
> MULTIPLE TABLES
> BY USING GROUP BY CLAUSE

> BY USING AGGREGATIVE FUNCTIONS
> BY USING HAVING CLAUSE
> BY USING SET OPERATORS
> BY USING DISTINCT KEYWORD
> BY USING JOINS
> BY USING SUBQUERY

- BY DEFAULT COMPLEX VIEWS ARE NOT ALLOWED DML OPERATIONS BUT IT ALLOWED "SELECT" COMMAND.

SYNTAX:
========
CREATE VIEW <VIEW NAME> AS <SELECT QUERY>;

EX:
```
SQL> CREATE VIEW V7 AS
  2  SELECT * FROM EMP_HYD
  3  UNION
  4  SELECT * FROM EMP_CHENNAI;
```

TESTING:
```
SQL> INSERT INTO V7 VALUES(1026,'ADAMS',63000);
SQL> UPDATE V7 SET SAL=48000 WHERE EID=1024;
SQL> DELETE FROM V7 WHERE EID=1021;
ERROR at line 1:
ORA-01732: data manipulation operation not legal on this view
```

EX:
```
SQL> CREATE VIEW V8
  2  AS
  3  SELECT DEPTNO,SUM(SAL)SUM_SAL FROM EMP
  4  GROUP BY DEPTNO;
```

TESTING:
```
SQL> SELECT * FROM V8;
```

FORCE VIEWS:
=============

- GENERALLY VIEWS ARE CREATED BASED ON BASE TABLES WHEREAS FORCE VIEWS ARE CREATED WITHOUT BASE TABLE.

SYNTAX:
========
CREATE FORCE VIEW <VIEW NAME> AS <SELECT QUERY>;

EX:
```
SQL> CREATE FORCE VIEW FV1 AS SELECT * FROM TEST50;
Warning: View created with compilation errors.

SQL> SELECT * FROM FV1;
ERROR at line 1:
ORA-04063: view "MYDB4PM.FV1" has errors
```

- TO ACTIVATE A FORCE VIEW THEN WE MUST CREATE A BASE TABLE WITH NAME

OF "TEST50".

EX:
SQL> CREATE TABLE TEST50(SNO INT);
SQL> INSERT INTO TEST50 VALUES(1021);
SQL> SELECT * FROM FV1;-----WORKING

NOTE:
=====
- TO SEE ALL VIEWS IN ORACLE DB THEN WE USE A DATADICTIONARY IS CALLED AS "USER_VIEWS".

EX:
SQL>DESC USER_VIEWS;
SQL> SELECT VIEW_NAME FROM USER_VIEWS;
SQL> SELECT TEXT FROM USER_VIEWS WHERE VIEW_NAME='V1';(SEE QUERY IN VIEW)

HOW TO DROP A VIEW:
==========
SYNTAX:
========
DROP VIEW <VIEW NAME>;

EX:
SQL> DROP VIEW V1;
SQL> DROP VIEW V8;
SQL> DROP VIEW FV1;
==========
MATERIALIZED VIEWS:
==========
- MVIEWS ARE JUST A VIEW CREATED BASED ON BASED TABLE.

VIEWS	MVIEWS
1. DOES NOT STORE DATA.	1. STORING DATA.
2. IT IS VIRTUAL / LOGICAL TABLE OF A BASE TABLE.	2. IT IS A TABLE OF A BASE TABLE. (BACKUP TABLES).
3. IT IS A DEPENDENT OBJECT.	3. IT IS A INDEPENDENT OBJECT.
4. WHEN WE DROP A BASE TABLE THEN WE CANNOT ACCESS A VIEW.	4. WHEN WE DROP A BASE TABLE EVEN THOUGH WE CAN ACCESS A MVIEW.
5. VIEWS SUPPORTING DML OPERATIONS.	5. MVIEWS ARE CANNOT ALLOWED DML OPERATIONS.

SYNTAX:
========
CREATE MATERIALIZED VIEW <VIEW NAME> AS <SELECT QUERY>;

EX:

```
SQL> CREATE TABLE TEST51(EID INT,ENAME VARCHAR2(10));
Table created.

SQL> CREATE VIEW V51 AS SELECT * FROM TEST51;
View created.

SQL> CREATE MATERIALIZED VIEW MV51 AS SELECT * FROM TEST51;
Materialized view created.

TESTING:
SQL> INSERT INTO TEST51 VALUES(1,'SMITH');

       - WHEN WE INSERT A ROW INTO A BASE TABLE(TEST51) THEN THAT ROW CAN
SEE UNDER VIEW TABLE(V51) BUT WE CANNOT SEE IN MVIEW TABLE(MV51).IF WE
WANT
TO VIEW DATA IN MVIEW(MV51) THEN WE SHOULD REFRESH A MVIEW BY USING THE
FOLLOWING TWO METHODS ARE:
              I) ON DEMAND
              II) ON COMMIT

I) ON DEMAND:
=============
       - IT IS A DEFAULT REFRESH METHOD OF MVIEW.
SYNTAX:
========
       EXECUTE DBMS_MVIEW.REFRESH('MVIEW NAME');
```

```
SEQUENCE:
==========
       - IT IS A DB OBJECT WHICH IS USED TO GENERATE SEQUENCE NUMBERS
ON A PARTICULAR COLUMN AUTOMATICALLY.
       - IT WILL PROVIDE  "AUTO INCREMENTAL VALUE" FACILITY ON A TABLE.

SYNTAX:
========
CREATE SEQUENCE <SEQUENCE NAME>
[START WITH n]
[MINVALUE n]
[INCREMENT BY n]
[MAXVALUE n]
[NO CYCLE / CYCLE]
[NO CACHE / CACHE n];

START WITH n:
=============
       - IT SPECIFY THE STARTING VALUE OF A SEQUENCE.
```

- "n" ---------- NUMBER

MINVALUE n:

==========

- IT SHOWS MINIMUM VALUE IN THE SEQUENCE.
- "n" --------- NUMBER

INCREMENT BY n:

==========

- IT SPECIFY INCRMENTAL VALUE IN BETWEEN SEQUENCE NUMBERS.
- "n" --------- NUMBER

MAXVALUE n:

==========

- IT SHOWS THE MAXIMUM VALUE IN THE SEQUENCE.
- "n" --------- NUMBER

NO CYCLE:

=========

- IT DEFAULT ATTRIBUTE OF SEQUENCE OBJECT.
- WHEN WE CREATED A SEQUENCE WITH "NO CYCLE" ATTRIBUTE THEN THE SET OF SEQUENCE NUMBERS ARE NOT REPEATED.

EX:

```
START WITH 1
MINVALUE 1
INCREMENT BY 1
MAXVALUE 3;
```

OUTPUT:

========

1
2
3 ------- ONE CYCLE COMPLETED.

CYCLE:

======

- WHEN WE CREATED A SEQUENCE WITH "CYCLE" ATTRIBUTE THEN THE SET OF SEQUENCE NUMBERS ARE REPEATED AGAIN AND AGAIN.

EX:

```
START WITH 1
MINVALUE 1
INCREMENT BY 1
MAXVALUE 3
CYCLE;
```

OUTPUT:

========

1
2
3
1
2
3
1
2

3

NO CACHE:
=========
- IT IS DEFAULT ATTRIBUTE OF SEQUENCE OBJECT.
- CACHE IS NOTHING BUT TEMPORARY MEMORY.
- WHEN WE CREATED A SEQUENCE OBJECT WITHOUT "CACHE" THEN THE SET SEQUENCE NUMBERS ARE SAVED IN DATABASE MEMORY DIRECTLY.SO THAT WHENEVER A USER WANT ACCESS DATA FROM A TABLE BASED ON SEQUENCE NUMBER THEN EVERY REQUEST WILL GO TO DATABASE TO FIND A ROW AND RETURN TO A USER APPLICATION THIS MAY BE INCRESE THE BURDON ON DATABASE AND REDUCE THE PERFORMANCE OF DATABASE.

CACHE n:
========
WHEN WE CREATED A SEQUENCE OBJECT WITH "CACHE" THEN THE SET SEQUENCE NUMBERS ARE SAVED IN DATABASE MEMORY AND COPY DATA IS SAVED IN CACHE MEMORY.SO THAT WHENEVER A USER WANT ACCESS DATA FROM A TABLE BASED ON SEQUENCE NUMBER THEN EVERY USER REQUEST WILL GO TO CACHE MEMORY INSTEAD OF DATABASE.TO FIND A ROW IN CACHE AND RETURN TO A USER APPLICATION THIS MAY BE REDUCE THE BURDON ON DATABASE AND IMPROVE THE PERFORMANCE OF DATABASE.HERE "n" THE SIZE CACH FILE. MINIMUM FILE SIZE IS 2KB AND MAXIMUM IS 2O KB.

NOTE:
======
WHEN WE WORK ON SEQUENCE THEN WE USE THE FOLLOWING PSEUDO COLUMNS ARE,
i) NEXTVAL : TO GENERATE THE NEXT BY NEXT SEQUECNE NUMBER.
ii) CURRVAL : TO SHOWS THE CURRENT VALUE OF SEQUENCE.

SNO
=====
1
2
3 ---------- NEXTVAL : 4 CURRVAL =3

EX1:

```
SQL> CREATE SEQUENCE SQ1
  2  START WITH 1
  3  MINVALUE 1
  4  INCREMENT BY 1
  5  MAXVALUE 3;
Sequence created.
```

TESTING:

```
SQL> CREATE TABLE TEST1(SNO INT,NAME VARCHAR2(10));

SQL> INSERT INTO TEST1 VALUES(SQ1.NEXTVAL,'&NAME');
Enter value for name: A
/
Enter value for name: B
/
Enter value for name: C
```

```
/
Enter value for name: D
ERROR at line 1:
ORA-08004: sequence SQ1.NEXTVAL exceeds MAXVALUE and cannot be
instantiated

SQL> SELECT * FROM TEST1;

       SNO NAME
---------- ----------
         1 A
         2 B
         3 C

ALTERING A SEQUENCE:
=====================
SYNTAX:
=======
ALTER SEQUENCE <SEQUENCE NAME> <ATTRIBUTE NAME> n;

EX:
SQL> ALTER SEQUENCE SQ1 MAXVALUE 5;
Sequence altered.

SQL> INSERT INTO TEST1 VALUES(SQ1.NEXTVAL,'&NAME');
Enter value for name: D
/
Enter value for name: E

SQL> SELECT * FROM TEST1;

       SNO NAME
---------- ----------
         1 A
         2 B
         3 C
         4 D
         5 E

EX2:
SQL> CREATE SEQUENCE SQ2
  2  START WITH 1
  3  MINVALUE 1
  4  INCREMENT BY 1
  5  MAXVALUE 3
  6  CYCLE
  7  CACHE 2;

TESTING:
SQL> CREATE TABLE TEST2(SNO INT,NAME VARCHAR2(10));
SQL> INSERT INTO TEST2 VALUES(SQ2.NEXTVAL,'&NAME');
Enter value for name: A
/
.....................................
```

```
/
.....................................

EX3:
SQL> CREATE SEQUENCE SQ3
  2  START WITH 3
  3  MINVALUE 1
  4  INCREMENT BY 1
  5  MAXVALUE 5
  6  CYCLE
  7  CACHE 2;

TESTING
SQL> CREATE TABLE TEST3(SNO INT,NAME VARCHAR2(10));

SQL> INSERT INTO TEST3 VALUES(SQ3.NEXTVAL,'&NAME');
Enter value for name: A
/
.................................
/
.................................

NOTE:
======
       - TO VIEW ALL SEQUENCE OBJECTS IN ORACLE DB THEN WE USE A
DATADICTIONARY "USER_SEQUENCES".

EX:
SQL> DESC USER_SEQUENCES;
SQL> SELECT SEQUENCE_NAME FROM USER_SEQUENCES;

HOW TO DROP A SEQUENCE:
=========================
SYNTAX:
========
DROP SEQUENCE <SEQUENCE NAME>;

EX:
DROP SEQUENCE SQ1;
=============================================================================
=
PARTITION TABLE:
================
       I) RANGE PARTITION
       II) LIST PARTITION
       III) HASH PARTITION

I) RANGE PARTITION:
===================
       - CREATED A PARTITION TABLE BASED ON A PARTICULAR RANGE VALUE.

SYNTAX:
=======
```

```
CREATE TABLE <TABLE NAME>(<COLUMN NAME1>
<DT>[SIZE],.................................)
PARTITION BY RANGE(KEY COLUMN NAME)
(PARTITION <PARTITION NAME1> VALUES LESS THAN(VALUE),
PARTITION <PARTITION NAME2> VALUES LESS THAN(VALUE),
.....................................................................
.....................
);

EX:
SQL> CREATE TABLE TEST41(EID INT,ENAME VARCHAR2(10),
  2  SAL NUMBER)PARTITION BY RANGE(SAL)
  3  (PARTITION P1 VALUES LESS THAN(500),
  4  PARTITION P2 VALUES LESS THAN(2000),
  5  PARTITION P3 VALUES LESS THAN(2500)
  6  );
Table created.

SQL> INSERT INTO TEST41 VALUES(1,'SMITH',1500);
SQL> INSERT INTO TEST41 VALUES(2,'ALLEN',450);

HOW TO CALL A PARTICULAR PARTITION:
===================================
SYNTAX:
========
SELECT * FROM <TN> PARTITION (PARTITION NAME);

EX:
SELECT * FROM TEST41 PARTITION(P1);

LIST PARTITION:
===============
       - CREATED A PARTITION TABLE BASED ON LIST OF VALUES.

SYNTAX:
=======
CREATE TABLE <TABLE NAME>(<COLUMN NAME1>
<DT>[SIZE],.................................)
PARTITION BY LIST(KEY COLUMN NAME)
(PARTITION <PARTITION NAME1> VALUES (V1,V2,V3,......),
PARTITION <PARTITION NAME2> VALUES (V1,V2,V3,......),
.....................................................
PARTITION OTHERS VALUES(DEFAULT));

EX:
SQL> CREATE TABLE TEST2(CID INT,CNAME VARCHAR2(10))
  2  PARTITION BY LIST(CNAME)
  3  (PARTITION P1 VALUES('ORACLE','MYSQL','MSSQL'),
  4  PARTITION P2 VALUES('JAVA','PHP','.NET'),
  5  PARTITION OTHERS VALUES(DEFAULT));
Table created.

SQL> INSERT INTO TEST2 VALUES(1,'C');
SQL> INSERT INTO TEST2 VALUES(2,'ORACLE');
```

.....;

CALLING A PARTITION:
====================
SQL> SELECT * FROM TEST2 PARTITION(P1);

HASH PARTITION:
===============
- CREATED PARTITION TABLE BY SYSTEM BY DEFAULT.

SYNTAX:
=======
CREATE TABLE <TN>(<COL1> <DT>[SIZE],.............)
PARTITION BY HASH(KEY COLUMN NAME) PARTITIONS <number>;

EX:
SQL> CREATE TABLE TEST3(ENAME VARCHAR2(10),SAL NUMBER(10))
  2 PARTITION BY HASH(SAL) PARTITIONS 5;

NOTE:
=====
- TO VIEW ALL PARTITIONS OF A PARTICULAR TABLE IN ORACLE DB THEN USE A DATADICTIONARY IS "USER_TAB_PARTITIONS".

EX:
SQL> DESC USER_TAB_PARTITIONS;
SQL> SELECT PARTITION_NAME FROM USER_TAB_PARTITIONS
     WHERE TABLE_NAME='TEST3';

HOW TO ADD A NEW PARTITION TO AN EXISTING TABLE:
==
SYNTAX FOR LIST PARTITION:
==========================
ALTER TABLE <TN> ADD PARTITION <PARTITION NAME> VALUES(V1,V2,....);

SYNTAX FOR RANGE PARTITION:
===========================
ALTER TABLE <TN> ADD PARTITION <PARTITION NAME> VALUES
LESS THAN(VALUE);

EX FOR LIST PARTITION:
======================
SQL>ALTER TABLE TEST2 ADD PARTITION P3VALUES('PYTHON','SAP');

ERROR at line 1:
ORA-14323: cannot add partition when DEFAULT partition exists

- TO ADD ANY NEW PARTITION TO LIST PARTITION TABLE THEN WE REMOVE "OTHERS" PARTITION THEN ONLY WE CAN ADD A NEW PARTITION.

SYNTAX TO DROP A PARTITION FROM AN EXISTING TABLE:
==
ALTER TABLE <TN> DROP PARTITION <PARTITION NAME>;

EX:
SQL> ALTER TABLE TEST2 DROP PARTITION OTHERS;
SQL> ALTER TABLE TEST2 ADD PARTITION P3 VALUES('PYTHON','SAP');

NOTE:
=====
- TO CHECK A TABLE IS PARTITIONED OR NOT THEN WE USE A DATADICTIONARY IS "USER_TABLES".

EX:
SQL> DESC USER_TABLES;
SQL> SELECT PARTITIONED FROM USER_TABLES
WHERE TABLE_NAME='DEPT';

PAR

NO

SQL> SELECT PARTITIONED FROM USER_TABLES
WHERE TABLE_NAME='TEST2';

PAR

YES
==========
LOCKS:
======
- IT IS A TECHNIQUE TO PREVENT UNAUTHORIZED ACCESS OF OUR RESOURCE.

1) ROW LEVEL LOCKING
2) TABLE LEVEL LOCKING

1) ROW LEVEL LOCKING:
=====================
i) SINGLE ROW LOCKING
ii) MULTIPLE ROWS LOCKING

i) SINGLE ROW LOCKING:
======================
- IN THIS LEVEL WE CAN LOCK A SINGLE ROW.

EX:

USER-1	USER-2
=======	=======
SQL> CONN SYSTEM/TIGER	SQL> CONN MYDB4PM/MYDB4PM
SQL> UPDATE MYDB4PM.EMP SET SAL=1100 WHERE EMPNO=7369;	SQL> UPDATE EMP SET SAL=2200 WHERE EMPNO=7369;

[ROW IS LOCKED] [CANNOT PERFORM UPDATE
- OPERATION]

SQL> COMMIT / ROLLBACK; SQL> 1 row updated.
[FOR RELEASING LOCK]

ii) MULTIPLE ROWS LOCKING:
==========

- WHEN WE LOCK MULTIPLE ROWS THEN WE SHOULD USE "FOR UPDATE" CLAUSE IN SELECT QUERY.

EX:

USER-1	USER-2
=======	=======
SQL> CONN SYSTEM/TIGER	SQL> CONN MYDB4PM/MYDB4PM
SQL> SELECT * FROM MYDB4PM.EMP WHERE DEPTNO=10 FOR UPDATE;	SQL> UPDATE EMP SET SAL=3300 WHERE DEPTNO=10;
[DEPTNO 10 ROWS ARE LOCKED]	[CANNOT PERFORM UPDATE]
SQL> COMMIT / ROLLBACK; [FOR RELEASING LOCK]	SQL> 3 rows updated.

DEAD LOCK:
==========

- BOTH USERS CAN LOCK RESOURCE TO EACH OTHER.

EX:

USER-1	USER-2
=======	========
SQL> CONN SYSTEM/TIGER	SQL> CONN MYDB4PM/MYDB4PM
SQL> UPDATE MYDB4PM.EMP SET SAL=4400 WHERE EMPNO=7369;	SQL> UPDATE EMP SET SAL=6600 WHERE EMPNO=7900;
[ROW IS LOCKED]	[ROW IS LOCKED]
SQL> UPDATE MYDB4PM.EMP SET SAL=5500 WHERE EMPNO=7900;	sql> UPDATE EMP SET SAL=7700 WHERE EMPNO=7369;
ERROR at line 1: ORA-00060: deadlock detected while - waiting for resource	

SQL> COMMIT / ROLLBACK ; SQL> 1 row updated.
[DEAD LOCK IS RELEASED]

2) TABLE LEVEL LOCK:
====================
- WE CAN LOCK THE ENTIRE TABLE(ALL ROWS)
i) SHARE LOCK
ii) EXCLUSIVE LOCK

i) SHARE LOCK
==============
- BOTH USERS CAN LOCK A TABLE.

SYNTAX:
=======
LOCK TABLE <TN> IN SHARE MODE;

EX:

	USER-1	USER-2
	SQL> CONN SYSTEM/TIGER	SQL> CONN MYDB4PM/MYDB4PM
	SQL> LOCK TABLE MYDB4PM.EMP IN SHARE MODE;	SQL> LOCK TABLE EMP IN SHARE MODE;
	Table(s) locked.	Table(s) locked.
	SQL> COMMIT / ROLLBACK; [FOR RELEASING LOCK]	SQL> COMMIT / ROLLBACK; [FOR RELEASING LOCK]

ii) EXCLUSIVE LOCK:
====================
- ANY ONE USER CAN LOCK A TABLE.

SYNTAX:
=======
LOCK TABLE <TN> IN EXCLUSIVE MODE;

EX:

	USER-1	USER-2
	SQL> CONN SYSTEM/TIGER	SQL> CONN MYDB4PM/MYDB4PM
	SQL> LOCK TABLE MYDB4PM.EMP IN EXCLUSIVE MODE;	SQL> LOCK TABLE EMP IN EXCLUSIVE MODE;

Table(s) locked. [CANNOR PERFORM EXCLUSIVE

- LOCK]

SQL> COMMIT / ROLLBACK; SQL> Table(s) locked.
[FOR RELEASING LOCK]

INDEXES:
========
- Index is a db object which is used to retrival the required row from a table fastly.
- it is similar to book index page in text books.
by using book index how we can take the required topic from a text book fastly same as by using db index object we can retrive a particular row from a table fastly.
- on which column we created an index object that column is called as "indexed key column" and this column only use under where clause condition to activate indexes on table.
- all databases are supporting the following two types of searching mechanisms those are,
1. Table scan(default)
2. Index scan

1. Table scan:
==============
- in this mechanism oracle server is scanning the entire table for required row data.

EX:
SELECT * FROM EMP WHERE SAL=3000;-----NO INDEX OBJECT

SOLUTION:
=========

SAL
800
1600
1250
2975
1250
2850
2450

```
      3000                                  WHERE SAL=3000;
      5000
      1500
      1100
       950
      3000
      1300
```

2. Index scan:
==========
- In this scan oracle server is scanning based on an indexed key column for required data.

i) implicit indexes
ii) explicit indexes

i) implicit indexes:
==========
- these indexes are created by system when we create a table along with "unique" (or) "primary key" constraint.

Ex:
```
SQL> CREATE TABLE TEST1(SNO INT UNIQUE,NAME VARCHAR2(10));
SQL> CREATE TABLE TEST2(EID INT PRIMARY KEY,SAL NUMBER(10));
```

NOTE:
=====
- if we want to view column name along with index name of a particular table then we use a datadictionaty is "user_ind_columns".

Ex:
```
sql> desc user_ind_columns;
SQL> SELECT COLUMN_NAME,INDEX_NAME FROM USER_IND_COLUMNS WHERE
  2  TABLE_NAME='TEST1';

COLUMN_NAME
INDEX_NAME
----------------------          --------------------------------
SNO
              SYS_C007528

SQL> SELECT COLUMN_NAME,INDEX_NAME FROM USER_IND_COLUMNS WHERE
  2  TABLE_NAME='TEST2';

COLUMN_NAME
INDEX_NAME
-------------------
---------------------------------
EID
              SYS_C007529
```

ii) Explicit indexes:
==========
- these indexes are created by the user in two types of indexes are,

1) B-tree index
> simple index
> composite index
> unique index
> functional based index

2) Bitmap index

simple index:
==========
- when we created an index object based on a single column.

syntax:
========
create index <index name> on <tn>(column name);

Ex:
SQL> CREATE INDEX I1 ON EMP(SAL);
SQL> SELECT * FROM EMP WHERE SAL=3000;

B-tree INDEX(I1)

|

<
(LP)|3000|(RP) >=

| |
<(LP)|2975|(RP)>=
<(LP)|5000|(RP)>=
|
|
2850|*,2450|*,1600|* ,
3000|*,* (* IS ROWID)
1500|*,1300|*,1250|*,*,
1100|*,950|*,800|*

ii) COMPOSITE INDEX:
==========
- when we created an index object based on multiple columns.

syntax:
=======
create index <index name> on <tn>(column name1,column name2,....);

Ex:
SQL> CREATE INDEX I2 ON EMP(DEPTNO,JOB);

EX:
SQL> SELECT * FROM EMP WHERE DEPTNO=10;-----INDEX SCAN
SQL> SELECT * FROM EMP WHERE JOB='MANAGER';----TABLE SCAN
SQL> SELECT * FROM EMP WHERE DEPTNO=10 AND JOB='MANAGER';---INDEX SCAN

NOTE:
=====
IN COMPOSITE INDEX MECHANISM INDEXES ARE ACTIVATED BASED ON LEADING

COLUMN ONLY.

iii) UNIQUE INDEX:
==========
- when we created an index object with unique constraint.

syntax:
========
create unique index <index name> on <tn>(column name);

EX:
SQL> CREATE UNIQUE INDEX UI ON DEPT(DNAME);

TESTING:
SQL> INSERT INTO DEPT VALUES(50,'SALES','HYD');
ERROR at line 1:
ORA-00001: unique constraint (MYDB4PM.UI) violated

SQL> INSERT INTO DEPT VALUES(50,'DBA','HYD');
1 row inserted.

iv) FUNCTIONAL BASED INDEX:
==========
- when we created an index object based on a function name.

syntax:
=======
create index <index name> on <tn>(<function name>(column name));

EX:
SQL> CREATE INDEX FI ON EMP(UPPER(ENAME));

TESTING:
SQL> SELECT * FROM EMP WHERE UPPER(ENAME)='smith';---NOT ALLOWED
SQL> SELECT * FROM EMP WHERE UPPER(ENAME)='SMITH';---ALLOWED

2) BITMAP INDEX:
==========
- are created on "low cardinality" columns in a table to improve the
performance of database.

Cardinality:
==========
- it refer uniqueness(distinct) values of a column.

formula:
========
cardinality of column = no.of distinct values / no.of rows in a table

EX:
cardinality of empno = 14 / 14 =====> 1 (high cardinality)---> btree index

cardinality of job = 5 / 14 ====> 0.35 (low cardinality)---> bitmap index

syntax:
=======
create bitmap index <index name> on <tn>(column name);

Ex:
SQL> CREATE BITMAP INDEX BIT ON EMP(JOB);
SQL> SELECT * FROM EMP WHERE JOB='CLERK';

- When we created a bitmap index object internally system will create
a bitmap indexed table with bit numbers "1" and "0".here "1" is represent condition true whereas "0" is represent condition is false.

bitmap indexed table(bit) (1 = ROWID)

JOB	1	2	3	4	5	6	7	8	9	10	11	12	13	14
CLERK	1	0	0	0	0	0	0	0	0	0	1	1	0	1
PRESIDNT	0	0	0	0	0	0	0	1	0	0	0	0	0	0
MANAGER	0	0	0	1	0	1	1	0	0	0	0	0	0	0
ANALYST	0	0	0	0	0	0	1	0	0	0	0	1	0	0
SALESMAN	0	1	1	0	0	0	0	0	1	0	0	0	0	0

NOTE:
=====
- TO VIEW INDEX NAME AND INDEX TYPE ON PARTICULAR TABLE IN ORACLE DB
THEN WE USE A DATADICTIONARY "USER_INDEXES".

EX:
```
SQL> DESC USER_INDEXES;
SQL> SELECT INDEX_NAME,INDEX_TYPE FROM USER_INDEXES
  2  WHERE TABLE_NAME='EMP';

INDEX_NAME                          INDEX_TYPE
------------------------------------------------------------------------------
-------------------------------------------------------
---------------------------
I1                                   NORMAL(btree index)
I2                                   NORMAL(btree index)
FI                                   FUNCTION-BASED NORMAL(btree index)
BIT                                  BITMAP
```

HOW TO DROP AN INDEX:
=====================
SYNTAX:
=======
```
DROP INDEX <INDEX NAME>;
```

EX:
```
SQL> DROP INDEX I1;

Index dropped.

SQL> DROP INDEX FI;

Index dropped.

SQL> DROP INDEX BIT;
```
==
CLUSTER:
========
- IT IS A COLLECTION OF TABLES TOGEHTER SAVED IN SAME DATABLOCK MEMORY.
- WE CREATE CLUSTER FOR IMPROVING THE PERFORMANCE OF JOINS.
- CLUSTER CAN BE CREATED AT THE TIME OF CREATING TABLES.
- WHENEVER WE CREATE A CLUSTER WE SHOULD HAVE A COMMON COLUMN IN TABLES
OTHERWISE WE CANNOT CREATE A CLUSTER.
- TO CREATE A CLUSTER MEMORY THEN WE FOLLOW THE FOLLOWING 3 STEPS:

STEP1: CREATE A CLUSTER:
========================
SYNTAX:
=======
```
CREATE CLUSTER <CLUSTER NAME>(<COMMON COLUMN NAME> <DT>[SIZE]);
```

STEP2: CREATE AN INDEX OBJECT ON CLUSTER MEMORY:
==========
SYNTAX:
=======
CREATE INDEX <INDEX NAME> ON CLUSTER <CLUSTER NAME>;

STEP3: CREATE CLUSTER TABLES:
==========
SYNTAX:
=======
CREATE TABLE <TN>(<COLUMN NAME1> <DT>[SIZE],.....)
CLUSTER <CLUSTER NAME>(COMMON COLUMN NAME);

EX:
```
SQL> CREATE CLUSTER EMP_DEPT(DEPTNO INT);
Cluster created.

SQL> CREATE INDEX IND1 ON CLUSTER EMP_DEPT;
Index created.

SQL> CREATE TABLE EMP1(EID INT,ENAME VARCHAR2(10),DEPTNO INT)
     CLUSTER EMP_DEPT(DEPTNO);
Table created.

SQL> INSERT INTO EMP1 VALUES(1021,'SMITH',10);
SQL> INSERT INTO EMP1 VALUES(1022,'WARD',20);
SQL> COMMIT;

SQL> CREATE TABLE DEPT1(DEPTNO INT,DNAME VARCHAR2(10))
  2  CLUSTER EMP_DEPT(DEPTNO);
Table created.

SQL> INSERT INTO DEPT1 VALUES(10,'D1');
SQL> INSERT INTO DEPT1 VALUES(20,'D2');
SQL> COMMIT;
```

NOTE:
=====
WHEN WE WANT TO KNOW THESE TABLES ARE IN CLUSTER MEMORY OR NOT THEN CHECK ROWID's OF TABLES LIKE BELOW,

```
SQL> SELECT ROWID FROM EMP1;

ROWID
------------------
AAASUNAAHAAAAJGAAA
AAASUNAAHAAAAJHAAA

SQL> SELECT ROWID FROM DEPT1;

ROWID
------------------
AAASUNAAHAAAAJGAAA
```

AAASUNAAHAAAAJHAAA

NOTE:
=====
- TO VIEW ALL CLUSTERS IN ORACLE DB THEN USE A DATADICTIONARY IS "USER_CLUSTERS".

EX:
```
SQL> DESC USER_CLUSTERS;
SQL> SELECT CLUSTER_NAME FROM USER_CLUSTERS;

CLUSTER_NAME
--------------------
EMP_DEPT
```

NOTE:
=====
- TO VIEW CLUSTER TABLES IN ORACLE DB THEN USE A DATADICTIONARY IS "USER_TABLES".

EX:
```
SQL> DESC USER_TABLES;
SQL> SELECT TABLE_NAME FROM USER_TABLES WHERE CLUSTER_NAME='EMP_DEPT';

TABLE_NAME
----------------------
EMP1
DEPT1
```

HOW TO DROP A CLUSTER:
======================
SYNTAX:
========
```
DROP CLSUTER <CLUSTER NAME>;
```

EX:
```
SQL> DROP CLUSTER EMP_DEPT
ERROR at line 1:
ORA-00951: cluster not empty
```

- IF WE WANT TO DROP A CLUSTER THEN USE "INCLUDING TABLES" STATEMENT.

SYNTAX:
========
```
DROP CLSUTER <CLUSTER NAME> INCLUDING TABLES;
```

EX:
```
SQL> DROP CLUSTER EMP_DEPT INCLUDING TABLES;
```

CURSORS:
========

-CURSOR IS A TEMPORARY MEMORY / SQL PRIVATE AREA / WORKSPACE.
i) EXPLICIT CURSOR
ii) IMPLICIT CURSOR

i) EXPLICIT CURSOR(static cursor):
==========
- these cursors are created by user for fetching multiple rows in row by row manner from a cursor memory table.
- explicit cursor can hold multiple rows but it will access only one row at a time.
- to create an explicit cursor memory we need to follow the following 4 steps are:

step1: declare cursor variable:
==========
syntax:
=======
declare cursor <cursor name> is <select query>;

step2: open cursor connection:
==========
syntax:
=======
open <cursor name>;

step3: fetching rows from a cursor memory:
==========
syntax:
=======
fetch <cursor name> into <variables>;

step4: close cursor connection:
==========
syntax:
========
close <cursor name>;

ATTRIBUTES OF AN EXPLICIT CURSORS:
==========
- these attributes are used tro check the status of cursor.

i) %isopen:
==========
- it is a default attribute.
- when cursor connection is successfully open then it returs true otherwise false.
- boolean type.

ii) %notfound:
==========
- when there is no data in cursor then return true otherwise false.
- boolean type.

iii) %found:
=============
- when there is a data in cursor then return true otherwise false.
- boolean type.

iv) %rowcount:
==============
- it returns no.of executed fetch statements.
- number type.

syntax:
=======
<cursor name>%<attribute name>;

Ex:
a cursor program to fetch a single row from a table?

```
SQL> DECLARE CURSOR C1 IS SELECT ENAME,SAL FROM EMP;
    v_ENAME VARCHAR2(10);
    v_SAL NUMBER(10);
    BEGIN
    OPEN C1;
    FETCH C1 INTO v_ENAME,v_SAL;
    DBMS_OUTPUT.PUT_LINE(v_ENAME||','||v_SAL);
    CLOSE C1;
    END;
   /
SMITH,800

PL/SQL procedure successfully completed.
```

Ex:
a cursor program to fetch multiple rows from a table?

```
SQL> DECLARE CURSOR C1 IS SELECT ENAME,SAL FROM EMP;
        v_ENAME VARCHAR2(10);
        v_SAL NUMBER(10);
        BEGIN
        OPEN C1;
        FETCH C1 INTO v_ENAME,v_SAL;
        DBMS_OUTPUT.PUT_LINE(v_ENAME||','||v_SAL);
         FETCH C1 INTO v_ENAME,v_SAL;
        DBMS_OUTPUT.PUT_LINE(v_ENAME||','||v_SAL);
         FETCH C1 INTO v_ENAME,v_SAL;
        DBMS_OUTPUT.PUT_LINE(v_ENAME||','||v_SAL);
                CLOSE C1;
                END;
                     /
```

NOTE:
=====
- In the above example we used no.of fetch statements to fetch multiple rows from a table.so to avoid this problem we use "looping statements" like below,

i) by using "smiple loop":

```
==========================
EX:
SQL> DECLARE CURSOR C1 IS SELECT ENAME,SAL FROM EMP;
  2  v_ENAME VARCHAR2(10);
  3  v_SAL NUMBER(10);
  4  BEGIN
  5  OPEN C1;
  6  LOOP
  7  FETCH C1 INTO v_ENAME,v_SAL;
  8  EXIT WHEN C1%NOTFOUND;
  9  DBMS_OUTPUT.PUT_LINE(v_ENAME||','||v_SAL);
 10  END LOOP;
 11  CLOSE C1;
 12  END;
 13  /
SMITH,800
ALLEN,1600
WARD,1250
JONES,2975
MARTIN,1250
BLAKE,2850
CLARK,2450
SCOTT,3000
KING,5000
TURNER,1500
ADAMS,1100
JAMES,950
FORD,3000
MILLER,1300

ii) by using "while loop":
==========================
EX:
SQL> DECLARE CURSOR C1 IS SELECT ENAME,SAL FROM EMP;
  2  v_ENAME VARCHAR2(10);
  3  v_SAL NUMBER(10);
  4  BEGIN
  5  OPEN C1;
  6  FETCH C1 INTO v_ENAME,v_SAL;----fetch starts from 1st row
  7  WHILE(C1%FOUND)
  8  LOOP
  9  DBMS_OUTPUT.PUT_LINE(v_ENAME||','||v_SAL);
 10  FETCH C1 INTO v_ENAME,v_SAL;----fetching upto last row
 11  END LOOP;
 12  CLOSE C1;
 13  END;
 14  /
SMITH,800
ALLEN,1600
WARD,1250
JONES,2975
MARTIN,1250
BLAKE,2850
CLARK,2450
```

```
SCOTT,3000
KING,5000
TURNER,1500
ADAMS,1100
JAMES,950
FORD,3000
MILLER,1300
```

iii) BY USING "FOR LOOP":
==========

```
SQL> DECLARE CURSOR C1 IS SELECT ENAME,SAL FROM EMP;
  2  BEGIN
  3  FOR i IN C1
  4  LOOP
  5  DBMS_OUTPUT.PUT_LINE(i.ENAME||','||i.SAL);
  6  END LOOP;
  7  END;
  8  /
SMITH,800
ALLEN,1600
WARD,1250
JONES,2975
MARTIN,1250
BLAKE,2850
CLARK,2450
SCOTT,3000
KING,5000
TURNER,1500
ADAMS,1100
JAMES,950
FORD,3000
MILLER,1300
```

2) IMPLICIT CURSORS:
==========

- these cursors are declared by oracle server by default. oracle server created implicit cursor memory when we perform dml operations on a particular table in database.
- implicit cursors are used to check the status of last dml query is executed successfully or not.
- implicit cursor variable name is "SQL".

ATTRIBUTES OF IMPLICIT CURSORS:
==========

i) %ISOPEN:
==========

- it is default attribute.
- it returns true when implicit cursor connection is open successfully otherwise false.

ii) %notfound:
==========

- it returns true when the last dml command is fail otherwise returns false.

iii) %found:
============
- it returns true when the last dml command is successfully executed otherwise false.

iv) %rowcount:
==============
- it returns the no.of rows affected by dml command.

syntax:
=======
SQL%<attribute name>

Ex:
```
SQL> DECLARE
  2  v_EMPNO NUMBER(10);
  3  BEGIN
  4  v_EMPNO:=&v_EMPNO;
  5  DELETE FROM EMP WHERE EMPNO=v_EMPNO;
  6  IF SQL%FOUND THEN
  7  DBMS_OUTPUT.PUT_LINE('RECORD IS FOUND AND DELETED');
  8  ELSE
  9  DBMS_OUTPUT.PUT_LINE('RECORD IS NOT FOUND');
 10  END IF;
 11  END;
 12  /
Enter value for v_empno: 7900
RECORD IS FOUND AND DELETED
PL/SQL procedure successfully completed.

SQL> /
Enter value for v_empno: 1122
RECORD IS NOT FOUND
PL/SQL procedure successfully completed.
```

REF CURSORS:
============
- When we assign a "select statement" at time of opening a cursor is called as "ref cursor" / "dynamic cursor".
1) weak ref cursor
2) strong ref cursor

weak ref cursor	strong ref cursor
1. there is no "return type".	1.delcare with "return type".
2. is having its own pre-defined datatype to declare weak ref cursor variable.	2. there is no pre-defined datatype for strong ref cursor so that we need to

create a user defined DT.

3. can access rows(data) from any table(more than one)

3. can access rows(data) from a specific table only

(one table).

Ex.on weak ref cursor with a single table:
==========

```
SQL> DECLARE
  2  C1 SYS_REFCURSOR;
  3  i EMP%ROWTYPE;
  4  BEGIN
  5  OPEN C1 FOR SELECT * FROM EMP WHERE DEPTNO=10;
  6  LOOP
  7  FETCH C1 INTO i;
  8  EXIT WHEN C1%NOTFOUND;
  9  DBMS_OUTPUT.PUT_LINE(i.ENAME||','||i.SAL||','||i.DEPTNO);
 10  END LOOP;
 11  CLOSE C1;
 12  END;
 13  /
MILLER,1300,10
CLARK,2450,10
KING,5000,10
```

Ex.on strong ref cursor with a single table:
==========

creating a user defined DT for strong ref cursor:
==========

syntax:
=======

```
type <type name> is ref cursor return <type>;
```

EX:

```
SQL> DECLARE
  2  TYPE UD_REFCURSOR IS REF CURSOR RETURN EMP%ROWTYPE;
  3  C1 UD_REFCURSOR;
  4  i EMP%ROWTYPE;
  5  BEGIN
  6  OPEN C1 FOR SELECT * FROM EMP WHERE DEPTNO=20;
  7  LOOP
  8  FETCH C1 INTO i;
  9  EXIT WHEN C1%NOTFOUND;
 10  DBMS_OUTPUT.PUT_LINE(i.ENAME||','||i.DEPTNO);
 11  END LOOP;
 12  CLOSE C1;
 13  END;
 14  /
SCOTT,20
FORD,20
SMITH,20
```

```
ADAMS,20
JONES,20

Ex.on weak ref cursor with multiple tables:
==========================================
SQL> DECLARE
  2  C1 SYS_REFCURSOR;
  3  i EMP%ROWTYPE;
  4  j DEPT%ROWTYPE;
  5  v_DEPTNO NUMBER(10):=&v_DEPTNO;
  6  BEGIN
  7  IF v_DEPTNO = 10 THEN
  8  OPEN C1 FOR SELECT * FROM EMP WHERE DEPTNO=10;
  9  LOOP
 10  FETCH C1 INTO i;
 11  EXIT WHEN C1%NOTFOUND;
 12  DBMS_OUTPUT.PUT_LINE(i.ENAME||','||i.DEPTNO);
 13  END LOOP;
 14  ELSIF v_DEPTNO = 20 THEN
 15  OPEN C1 FOR SELECT * FROM DEPT WHERE DEPTNO=20;
 16  LOOP
 17  FETCH C1 INTO j;
 18  EXIT WHEN C1%NOTFOUND;
 19  DBMS_OUTPUT.PUT_LINE(j.DEPTNO||','||j.DNAME||','||j.LOC);
 20  END LOOP;
 21  CLOSE C1;
 22  END IF;
 23  END;
 24  /
Enter value for v_deptno: 10
MILLER,10
CLARK,10
KING,10
PL/SQL procedure successfully completed.

SQL> /
Enter value for v_deptno: 20
20,RESEARCH,DALLAS
PL/SQL procedure successfully completed.

Ex.on strong ref cursor with multiple tables:
============================================
SQL> DECLARE
        TYPE UD_REFCURSOR IS REF CURSOR RETURN EMP%ROWTYPE;
    C1 UD_REFCURSOR;
    i EMP%ROWTYPE;
    j DEPT%ROWTYPE;
    v_DEPTNO NUMBER(10):=&v_DEPTNO;
    BEGIN
    IF v_DEPTNO = 10 THEN
    OPEN C1 FOR SELECT * FROM EMP WHERE DEPTNO=10;
    LOOP
   FETCH C1 INTO i;
   EXIT WHEN C1%NOTFOUND;
```

```
   DBMS_OUTPUT.PUT_LINE(i.ENAME||','||i.DEPTNO);
   END LOOP;
   ELSIF v_DEPTNO = 20 THEN
   OPEN C1 FOR SELECT * FROM DEPT WHERE DEPTNO=20;
   LOOP
   FETCH C1 INTO j;
   EXIT WHEN C1%NOTFOUND;
   DBMS_OUTPUT.PUT_LINE(j.DEPTNO||','||j.DNAME||','||j.LOC);
   END LOOP;
   CLOSE C1;
   END IF;
   END;
   /
ERROR:
PLS-00394: wrong number of values in the INTO list of a FETCH statement
```

```
EXCEPTION HANDLING:
===================
       What is an Exception?
                      - it is a runtime / execution error.

       What is an Exception Handling?
                      - to avoid abnormal terimination of a program
execution
problem.

       - In pl/sql there are two types of exceptions,
                              1. pre-defined exceptions
                              2. user-defined exceptions

1. pre-defined exceptions:
==========================
               - these are inbuilt exceptions in oracle.
                              > no_data_found
                              > too_many_rows
                              > zero_divide
                              > invalid_cursor
                              > cursor_already_open .........etc

no_data_found:
==============
       - when we try to fetch data from a table by using select...
into statement and that row is not found in a table then oracle
server raise an exception is called as "no data found".
Ex:
SQL> DECLARE
  2  v_ENAME VARCHAR2(10);
  3  BEGIN
  4  SELECT ENAME INTO v_ENAME FROM EMP WHERE EMPNO=&EMPNO;
  5  DBMS_OUTPUT.PUT_LINE(v_ENAME);
```

```
  6  END;
  7  /
Enter value for empno: 7788
SCOTT
PL/SQL procedure successfully completed.

SQL> /
Enter value for empno: 1122
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 4
```

- to overcome the an exception oracle server provide a pre-defined exception name is "no_data_found".

solution:

```
SQL> DECLARE
  2  v_ENAME VARCHAR2(10);
  3  BEGIN
  4  SELECT ENAME INTO v_ENAME FROM EMP WHERE EMPNO=&EMPNO;
  5  DBMS_OUTPUT.PUT_LINE(v_ENAME);
  6  EXCEPTION
  7  WHEN NO_DATA_FOUND THEN
  8  DBMS_OUTPUT.PUT_LINE('RECORD IS NOT FOUND');
  9  END;
 10  /
Enter value for empno: 7900
RECORD IS NOT FOUND
PL/SQL procedure successfully completed.
```

II) TOO_MANY_ROWS:

- when we fetch more than one row from a table by using select.....into statement then oracle server raise an exception is "exact fetch returns more than requested number of rows".

Ex:

```
SQL> SELECT * FROM TEST1;

       SNO        SAL
---------- ----------
         1      24000
         2      34000

SQL> DECLARE
  2  v_SAL NUMBER(10);
  3  BEGIN
  4  SELECT SAL INTO v_SAL FROM TEST1;
  5  DBMS_OUTPUT.PUT_LINE(v_SAL);
  6  END;
  7  /
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
```

```
ORA-06512: at line 4
```

- to handle the above exception oracle server provide a pre-defined exception name is "too_many_rows".

solution:

```
=========
SQL> DECLARE
  2  v_SAL NUMBER(10);
  3  BEGIN
  4  SELECT SAL INTO v_SAL FROM TEST1;
  5  DBMS_OUTPUT.PUT_LINE(v_SAL);
  6  EXCEPTION
  7  WHEN TOO_MANY_ROWS THEN
  8  DBMS_OUTPUT.PUT_LINE('A TABLE IS HAVING MORE THAN ONE ROW');
  9  END;
 10  /
A TABLE IS HAVING MORE THAN ONE ROW
PL/SQL procedure successfully completed.
```

III)ZERO_DIVIDE:

```
================
```

- when we perform division by zero then oracle returns an exception is called as " divisor is equal to zero".

Ex:

```
SQL> DECLARE
  2  X NUMBER(10);
  3  Y NUMBER(10);
  4  Z NUMBER(10);
  5  BEGIN
  6  X:=&X;
  7  Y:=&Y;
  8  Z:=X/Y;
  9  DBMS_OUTPUT.PUT_LINE(Z);
 10  END;
 11  /
Enter value for x: 10
Enter value for y: 2
5
PL/SQL procedure successfully completed.

SQL> /
Enter value for x: 10
Enter value for y: 0
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at line 8
```

- to handle the above exception oracle provide a pre-defined exception name is " zero_divide ".

SOLUTION:

```
==========
```

```
SQL> DECLARE
  2  X NUMBER(10);
  3  Y NUMBER(10);
  4  Z NUMBER(10);
  5  BEGIN
  6  X:=&X;
  7  Y:=&Y;
  8  Z:=X/Y;
  9  DBMS_OUTPUT.PUT_LINE(Z);
 10  EXCEPTION
 11  WHEN ZERO_DIVIDE THEN
 12  DBMS_OUTPUT.PUT_LINE('SECOND NUMBER SHOULD NOT BE ZERO');
 13  END;
 14  /

Enter value for x: 10
Enter value for y: 0
SECOND NUMBER SHOULD NOT BE ZERO
PL/SQL procedure successfully completed.
```

SQLCODE & SQLERRM:
==========

- these are pre-defined properties which are used to handle exceptions which was raised in a pl/sql program automatically and return the information of an exception.
- when we use these properties we should use "others" exception name.

SQLCODE : returns exception number / code.
SQLERRM : returns error message.

EX:

```
SQL> DECLARE
  2  X NUMBER(10);
  3  Y NUMBER(10);
  4  Z NUMBER(10);
  5  BEGIN
  6  X:=&X;
  7  Y:=&Y;
  8  Z:=X/Y;
  9  DBMS_OUTPUT.PUT_LINE(Z);
 10  EXCEPTION
 11  WHEN OTHERS THEN
 12  DBMS_OUTPUT.PUT_LINE(SQLCODE);
 13  DBMS_OUTPUT.PUT_LINE(SQLERRM);
 14  END;
 15  /
Enter value for x: 10
Enter value for y: 2
5
PL/SQL procedure successfully completed.

SQL> /
Enter value for x: 10
Enter value for y: 0
```

-1476
ORA-01476: divisor is equal to zero
PL/SQL procedure successfully completed.

2) USER DEFINED EXCEPTION NAMES:
==========
- when we create an exception name to raise and handling exception in pl/sql program is called as "user defined exception name".
- to create a user defined exception name then we follow the following 3 steps are:

step1: declare user defined exeption name:
==========
syntax:
=======
<UD exception name> exception;

step2: to raise a UD exception name:
==========
method-1:
=========
syntax:
=======
raise <UD exception name>;

- "raise" statement raised an excption and also handled an exception in pl/sql program.

method-2:
=========
syntax:
========
raise_application_error(number,message);

- this statement will raise an exception but not handled an exception in a pl/sql program.
- thise statement is having two arguments those are

number : it return user defined exception number.
UD exception number must be form -20000
to -20999.
message : it return user defined error message.

step3: Handling exceptions by using UD exception name:
==========
syntax:
=======
when <UD excpetion name> then
<statements>;
end;

```
       /

EX:
i) BY USING "RAISE" STATEMENT:
==============================
SQL> DECLARE
  2  X NUMBER(10);
  3  Y NUMBER(10);
  4  Z NUMBER(10);
  5  EX EXCEPTION;------(1)
  6  BEGIN
  7  X:=&X;
  8  Y:=&Y;
  9  IF Y=0 THEN
 10  RAISE EX;-------(2)
 11  ELSE
 12  Z:=X/Y;
 13  DBMS_OUTPUT.PUT_LINE(Z);
 14  END IF;
 15  EXCEPTION
 16  WHEN EX THEN----(3)
 17  DBMS_OUTPUT.PUT_LINE('SECOND NUMBER NOT BE ZERO');
 18  END;
 19  /
Enter value for x: 10
Enter value for y: 2
5
PL/SQL procedure successfully completed.

SQL> /
Enter value for x: 10
Enter value for y: 0
SECOND NUMBER NOT BE ZERO
PL/SQL procedure successfully completed.

ii) BY USING "RAISE_APPLICATION_ERROR()":
=========================================
SQL> DECLARE
  2  X NUMBER(10);
  3  Y NUMBER(10);
  4  Z NUMBER(10);
  5  EX EXCEPTION;
  6  BEGIN
  7  X:=&X;
  8  Y:=&Y;
  9  IF Y=0 THEN
 10  RAISE EX;
 11  ELSE
 12  Z:=X/Y;
 13  DBMS_OUTPUT.PUT_LINE(Z);
 14  END IF;
 15  EXCEPTION
 16  WHEN EX THEN
 17  RAISE_APPLICATION_ERROR(-20478,'SECOND NUMBER NOT ZERO');
```

```
 18  END;
 19  /
Enter value for x: 10
Enter value for y: 2
5
PL/SQL procedure successfully completed.

SQL> /
Enter value for x: 10
Enter value for y: 0
ERROR at line 1:
ORA-20478: SECOND NUMBER NOT ZERO
ORA-06512: at line 17
```

EXCEPTION PROPAGATION:
==========

- Exception block can handled exceptions which was raise in execution block but not in declarion block.so to overcome this problem we use a mechanism which can handle exceptions which was raised in declaration block.this mechanism is called as "exception propagation".
- by using exception propagation mechanism we can handle exceptions which was raised in declaration block in pl/sql program.

Ex:

```
SQL> DECLARE
  2  X VARCHAR2(3):='WARD';
  3  BEGIN
  4  DBMS_OUTPUT.PUT_LINE(X);
  5  EXCEPTION
  6  WHEN VALUE_ERROR THEN
  7  DBMS_OUTPUT.PUT_LINE('INVALID STRING LENGTH');
  8  END;
  9  /
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character string buffer too
small
ORA-06512: at line 2
```

- to handle the above exception then we use "exception propagation". it can be implemented by using "nested pl/sql block".whereas in nested pl/sql block outer block exception block only can handle exceptions which was raised in declaration block.

SOLUTION:
==========

```
SQL> BEGIN
  2  DECLARE
  3  X VARCHAR2(3):='WARD';
  4  BEGIN
  5  DBMS_OUTPUT.PUT_LINE(X);
  6  EXCEPTION
  7  WHEN VALUE_ERROR THEN
```

```
  8  DBMS_OUTPUT.PUT_LINE('INVALID STRING LENGTH');
  9  END;
 10  EXCEPTION
 11  WHEN VALUE_ERROR THEN
 12  DBMS_OUTPUT.PUT_LINE('STRING LENGTH IS MORE THAN THE SIZE OF
DATATYPE');
 13  END;
 14  /

STRING LENGTH IS MORE THAN THE SIZE OF DATATYPE
PL/SQL procedure successfully completed.
```

---

sub blocks:
==========
- are named blocks which can store a program source code in database automatically.
- sub block objects are 4 types those are,
  1. stored procedures
  2. stored functions
  3. packages
  4. triggers

1. stored procedures:
====================
- it is a named object which contains pre-compiled code / query.
- it is a block code to perform some operation and it may be or may not be return a value.
- generally stored procedures are never returns a value.if a procedure want to return a value then we use "OUT" parameters in stored procedure.

syntax:
========
```
create [or replace] procedure <pname>(<parameter name1> [mode type]
<datatype>,.........)
is / as
<declare variables>;
begin
<procedure body / statements>;
end;
/
```

How to call a stored procedure:
==============================
syntax1:
========

```
execute <pname>(value/(s));
              (or)
exec <pname>(value / (s));

syntax2: anonymous block:
=========================
begin
<pname>(value / (s));
end;
/

Types of parameters modes:
==========================
       - in pl/sql a stored procedure is supporting the following
three types of parameters modes those are,
                              I) IN mode
                              II) OUT mode
                              III) IN OUT mode

I) IN mode:
===========
       - these parameters are storing input values which was given
by user at execution time.
       - these are default parameters of a stored procedure.

II) OUT mode:
=============
       -these parameters are used to return output value from a
stored procedure.


III) IN OUT mode:
=================
       - a prameter which can store and also which can return a
value.

Ex.on IN mode:
==============
create a SP to accept empno as IN parameter and display that
employee name and salary details from emp table?
SQL> CREATE OR REPLACE PROCEDURE SP1(p_EMPNO IN NUMBER)
  2  IS
  3  v_ENAME VARCHAR2(10);
  4  v_SAL NUMBER(10);
  5  BEGIN
  6  SELECT ENAME,SAL INTO v_ENAME,v_SAL FROM EMP
  7  WHERE EMPNO=p_EMPNO;
  8  DBMS_OUTPUT.PUT_LINE(v_ENAME||','||v_SAL);
  9  END;
 10  /

Procedure created.

CALLING A SP:
```

```
=============
SQL> EXECUTE SP1(7788);
SCOTT,3000
                      (OR)
SQL> EXEC SP1(7788);
SCOTT,3000
                      (OR)
SQL> BEGIN
  2  SP1(7788);
  3  END;
  4  /
SCOTT,3000
```

NOTE:
=====
- To view all sub blocks objects(SP,SF,package,trigger) in oracle db then use a datadictionary "user_objects".

EX:

```
SQL> DESC USER_OBJECTS;
SQL> SELECT OBJECT_NAME FROM USER_OBJECTS WHERE OBJECT_TYPE='PROCEDURE';

OBJECT_NAME
-----------------
SP1
```

NOTE:
=====
- To view the source code of a sub block object(SP/SF/Package/Trigger) then use a datadictionary is "user_source".

EX:

```
SQL> DESC USER_SOURCE;
SQL> SELECT TEXT FROM USER_SOURCE WHERE NAME='SP1';
```

EX.ON "OUT" PARAMETERS:
=======================

```
SQL> CREATE OR REPLACE PROCEDURE SP2(X IN NUMBER,Y OUT NUMBER)
  2  IS
  3  BEGIN
  4  Y:=X*X;
  5  END;
  6  /
Procedure created.

SQL> EXECUTE SP2(5);
ERROR at line 1:
ORA-06550: line 1, column 7:
PLS-00306: wrong number or types of arguments in call to 'SP2'
```

- To overcome the above problem we should follow the following 3 steps procedure,

step1: declare bind / referenced variables for "OUT" parameters:
==========
syntax:
=======
var[iable] <bind variable name> <datatype>[size];

step2: to add bind / referenced variables to a stored procedure:
==========
syntax:
========
execute <pname>(value1,value2,.....,:<bind variable name1>,
:<bind variable name2>,.............);

step3: print bind / referenced variables:
==========
syntax:
=======
print <bind variable name>;

OUTPUT:
=======
```
SQL> VAR A NUMBER;
SQL> EXECUTE SP2(5,:A);

PL/SQL procedure successfully completed.

SQL> PRINT A;

         A
----------
        25
```

EX:
create a SP to input empno as a IN parameter and returns that employee provident fund,professional tax at 5%,10% on basic salary of employee by using "OUT" parameters?

```
SQL> CREATE OR REPLACE PROCEDURE SP3(p_EMPNO IN NUMBER,PF OUT NUMBER,PT
OUT NUMBER)
  2  IS
  3  v_BSAL NUMBER(10);
  4  BEGIN
  5  SELECT SAL INTO v_BSAL FROM EMP WHERE EMPNO=p_EMPNO;
  6  PF:=v_BSAL*0.05;
  7  PT:=v_BSAL*0.1;
  8  END;
  9  /

Procedure created.

SQL> VAR rPF NUMBER;
SQL> VAR rPT NUMBER;
SQL> EXECUTE SP3(7788,:rPF,:rPT);
PL/SQL procedure successfully completed.
```

```
SQL> PRINT rPF rPT;

       RPF
----------
       150


       RPT
----------
       300
```

EX.ON "IN OUT" PARAMETER:
==========

```
SQL> CREATE OR REPLACE PROCEDURE SP4(X IN OUT NUMBER)
  2  IS
  3  BEGIN
  4  X:=X*X*X;
  5  END;
  6  /

Procedure created.

SQL> EXECUTE SP4(5);
ERROR at line 1:
ORA-06550: line 1, column 11:
PLS-00363: expression '5' cannot be used as an assignment target
```

- To overcome the above problem we need to follow the following 4 steps procedure,

step1: declare bind / referenced variable for "OUT" parameter:
==========

syntax:
========

```
var[iable] <bind variable name> <datatype>[size];
```

step2: to assign a value to bind variable:
==========

syntax:
=======

```
execute :<bind variable name> := <value>;
```

step3: to add bind variable to a stored procedure:
==========

syntax:
=======

```
execute <pname>(:<bind variable name>);
```

step4: to print bind variables:
==========

syntax:
=======

```
print <bind variable name>;
```

```
OUTPUT:
=======
SQL> VAR A NUMBER;
SQL> EXECUTE :A:=5;

PL/SQL procedure successfully completed.

SQL> EXECUTE SP4(:A);

PL/SQL procedure successfully completed.

SQL> PRINT A;

         A
----------
       125

HOW TO DROP A STORED PROCEDURE:
===============================
SYNTAX:
=======
DROP PROCEDURE <PNAME>;

EX:
DROP PROCEDURE SP1;
================================================================
2) STORED FUNCTIONS:
====================
       - a function is block of code to perform some task and it
must return a value.
       - these functions are created by user as per requirement.
so that stored functions are also called as "user defined functions"
in oracle.

syntax:
=======
create [or replace] function <fname>(<parameter name1>
<datatype>,.........)
return <return VARIABLE DATATYPE>
is / as
<declare variables>;
begin
<function body / statements>;
return <return VARIABLE NAME>;
end;
/

How to call a stored function:
==============================
syntax:
=======
select <fname>(value/(s)) from dual;
```

EX:
create a SF to input empno and return that employee name
from emp table?

```
SQL> CREATE OR REPLACE FUNCTION SF1(p_EMPNO NUMBER)
  2  RETURN VARCHAR2
  3  AS
  4  v_ENAME VARCHAR2(10);
  5  BEGIN
  6  SELECT ENAME INTO v_ENAME FROM EMP WHERE EMPNO=p_EMPNO;
  7  RETURN v_ENAME;
  8  END;
  9  /

Function created.
```

OUTPUT:
=======

```
SQL> SELECT SF1(7369) FROM DUAL;

SF1(7369)
---------
SMITH
```

EX:
create a SF to return the sum of salaries of given department name?

```
SQL> CREATE OR REPLACE FUNCTION SF2(p_DNAME VARCHAR2)
  2  RETURN NUMBER
  3  AS
  4  v_SUMSAL NUMBER(10);
  5  BEGIN
  6  SELECT SUM(SAL) INTO v_SUMSAL FROM EMP E INNER JOIN DEPT D
  7  ON E.DEPTNO=D.DEPTNO AND DNAME=p_DNAME;
  8  RETURN v_SUMSAL;
  9  END;
 10  /

Function created.

SQL> SELECT SF2('SALES') FROM DUAL;

SF2('SALES')
------------
        8450
```

EX:
create a SF to return the no.of employees are joined in between
the given two date expressions?

```
SQL> CREATE OR REPLACE FUNCTION SF3(SD DATE,ED DATE)
  2  RETURN NUMBER
  3  AS
  4  v_NUMEMP NUMBER(10);
  5  BEGIN
  6  SELECT COUNT(*) INTO v_NUMEMP FROM EMP
  7  WHERE HIREDATE BETWEEN SD AND ED;
  8  RETURN v_NUMEMP;
```

```
  9  END;
 10  /

Function created.

SQL> SELECT SF3('01-JAN-81','31-DEC-81') FROM DUAL;

SF3('01-JAN-81','31-DEC-81')
----------------------------
                           9
```

EX:

create a SF to input empno and return that employee gross salary based on the following conditions are,

i) HRA ------ 10%
ii) DA ------ 20%
iii) PF ----- 10%

on basic salary ?

empno : 7788

> bsal : 3000

HRA = bsal*10%
= 3000*0.1
= 300

DA = 3000*0.2
= 600

PF = 3000*0.1
= 300

> gross = bsal+hra+da+pf
= 3000+300+600+300
= 4200

```
SQL> CREATE OR REPLACE FUNCTION SF4(p_EMPNO NUMBER)
  2  RETURN NUMBER
  3  AS
  4  v_BSAL NUMBER(10);
  5  v_HRA NUMBER(10);
  6  v_DA NUMBER(10);
  7  v_PF NUMBER(10);
  8  v_GROSS NUMBER(10);
  9  BEGIN
 10  SELECT SAL INTO v_BSAL FROM EMP WHERE EMPNO=p_EMPNO;
 11  v_HRA:=v_BSAL*0.1;
 12  v_DA:=v_BSAL*0.2;
 13  v_PF:=v_BSAL*0.1;
 14  v_GROSS:=v_BSAL+v_HRA+v_DA+v_PF;
 15  RETURN v_GROSS;
 16  END;
 17  /

Function created.
```

```
SQL> SELECT SF4(7788) FROM DUAL;

 SF4(7788)
----------
      4200
```

NOTE:
=====
```
SQL> SELECT OBJECT_NAME FROM USER_OBJECTS WHERE OBJECT_TYPE='FUNCTION';
SQL> SELECT TEXT FROM USER_SOURCE WHERE NAME='SF1';
```

HOW TO DROP A STORED FUNCTION:
==============================
SYNTAX:
=======
```
DROP FUNCTION <FNAME>;
```

EX:
```
DROP FUNCTION SF1;
```

---

3. packages:
============
- it is a collection of variables,stored procedures and stored functions are stored in a single unit of memory is called as "package".
- by using package we can share the required sub programs(procedures and functions) to client application and importing that package into client application for reusable of those sub programs.
- by using packages we are implementing encapsulating and function overloading mechanisms.
- to create a package in pl/sql then we should create the following two blocks those are
    1. package specification block
    2. package implementation block(body)

1. package specification block:
==============================
- in this block we can declare variables,procedures and functions.

syntax:
=======
```
create [or replace] package <package name>
is / as
<declare variables>;
```

```
<declare stored procedures>;
<declare stored functions>;
end;
/

2. Package implementation block(body):
=======================================
       - in this block we can implement the logical
code of procedure and function which was declared
in package specification block.

syntax:
=======
create [or replace] package body <package name>
is / as
<implementing logical code>;
end;
/

How to call a SP from a package:
================================
syntax:
=======
execute <package name>.<pname>(values);

How to call a SF from a package:
================================
syntax:
=======
select <package name>.<fname>(value/(s)) from dual;

Ex:
create a package to bind multiple stored procedures
in a single pl/sql block.
SQL> CREATE OR REPLACE PACKAGE PK1
  2  IS
  3  PROCEDURE SP1;
  4  PROCEDURE SP2;
  5  END;
  6  /

Package created.

SQL> CREATE OR REPLACE PACKAGE BODY PK1
  2  IS
  3  PROCEDURE SP1
  4  AS
  5  BEGIN
  6  DBMS_OUTPUT.PUT_LINE('I AM PROCEDURE-1');
  7  END SP1;
  8  PROCEDURE SP2
  9  AS
 10  BEGIN
 11  DBMS_OUTPUT.PUT_LINE('I AM PROCEDURE-2');
```

```
 12  END SP2;
 13  END;
 14  /

Package body created.

OUTPUT:
=======
SQL> EXECUTE PK1.SP1;
I AM PROCEDURE-1
PL/SQL procedure successfully completed.

SQL> EXECUTE PK1.SP2;
I AM PROCEDURE-2
PL/SQL procedure successfully completed.

EX:
create a package to bind variable,stored procedure
and stored function in a single unit of memory?

SQL> CREATE OR REPLACE PACKAGE PK2
  2  AS
  3  X NUMBER(10):=1000;
  4  PROCEDURE SP1;
  5  FUNCTION SF1(A NUMBER) RETURN NUMBER;
  6  END;
  7  /

Package created.

SQL> CREATE OR REPLACE PACKAGE BODY PK2
  2  AS
  3  PROCEDURE SP1
  4  IS
  5  A NUMBER(10);
  6  BEGIN
  7  A:=X/2;
  8  DBMS_OUTPUT.PUT_LINE(A);
  9  END SP1;
 10  FUNCTION SF1(A NUMBER)
 11  RETURN NUMBER
 12  AS
 13  BEGIN
 14  RETURN A*X;
 15  END SF1;
 16  END;
 17  /

Package body created.

SQL> SELECT PK2.SF1(5) FROM DUAL;

PK2.SF1(5)
----------
```

```
      5000

SQL> EXECUTE PK2.SP1;
500
PL/SQL procedure successfully completed.
```

IMPLEMENTING FUNCTION OVERLOADING MECHANISM:
==========

- TO CHANGE NO.OF ARGUMENTS TO THE SAME FUNCTION IS CALLED AS "FUNCTION OVERLOADING".

EX: FUNCTION F1(X NUMBER,Y NUMBER);
FUNCTION F1(A NUMBER,B NUMBER,C NUMBER);

EX:

```
SQL> CREATE OR REPLACE PACKAGE PK3
  2  IS
  3  FUNCTION F1(X NUMBER,Y NUMBER) RETURN NUMBER;
  4  FUNCTION F1(A NUMBER,B NUMBER,C NUMBER) RETURN NUMBER;
  5  END;
  6  /
Package created.

SQL> CREATE OR REPLACE PACKAGE BODY PK3
  2  IS
  3  FUNCTION F1(X NUMBER,Y NUMBER)
  4  RETURN NUMBER
  5  AS
  6  BEGIN
  7  RETURN X*Y;
  8  END F1;
  9  FUNCTION F1(A NUMBER,B NUMBER,C NUMBER)
 10  RETURN NUMBER
 11  AS
 12  BEGIN
 13  RETURN A+B+C;
 14  END F1;
 15  END;
 16  /
Package body created.

SQL> SELECT PK3.F1(10,20,30) FROM DUAL;

PK3.F1(10,20,30)
----------------
              60

SQL> SELECT PK3.F1(10,20) FROM DUAL;

PK3.F1(10,20)
-------------
          200
```

NOTE:

```
SQL> SELECT OBJECT_NAME FROM USER_OBJECTS WHERE OBJECT_TYPE='PACKAGE';
SQL> SELECT TEXT FROM USER_SOURCE WHERE NAME='PK1';

HOW TO DROP A PACKAGE BODY:
===========================
SYNTAX:
=======
DROP PACKAGE BODY <PACKAGE NAME>;

EX:
DROP PACKAGE BODY PK1;

HOW TO DROP A PACKAGE:
======================
SYNTAX:
=======
DROP PACKAGE <PACKAGE NAME>;

EX:
DROP PACKAGE PK1;
================================================================
TRIGGERS:
=========
       - it is named block similar to stored procedure but executed
by system automatically.

purpose of triggers:
====================
1. to raise user defined alerts along with security.
2. to control / restricted dml and ddl operations over database.
3. to implement business logical conditions.
4. to validating data
5. for auditing

Types of triggers:
==================
              1. DML triggers
              2. DDL triggers(DB triggers)

1. DML triggers:
================
       - when we created a trigger object based on dml commands
(insert/update/delete) then we called as "DML triggers".
       - these trigger are executed by system automatically when
user perform dml operations on a particular table in database.

syntax:
=======
create [or replace] trigger <trigger name>
before / after insert or update or delete on <table name>
[for each row ] -----> use in row level triggers only
begin
<trigger body / statements>;
end;
```

/

Trigger events:
==========
Before trigger:
==========
- when we created a trigger object with "before" event then,
first : trigger body executed
later : DML command will execute

After trigger:
==========
- when we created a trigger object with "after" event then,
first : DML command executed
later : trigger body will execute

NOTE: both are providing same result / output.

Levels of triggers:
==========
- trigger object can be created at two levels.
i) row level triggers
ii) statement level triggers

i) row level triggers:
==========
- in this level a trigger body is executing for each row wise in a table so that we should use "for each row" statement.

EX:

```
SQL> SELECT * FROM TEST;

ENAME             SAL
---------- ----------
SMITH           12000
JONES           35000
ALLEN           12000
WARD            68000

SQL> CREATE OR REPLACE TRIGGER TR1
  2  AFTER UPDATE ON TEST
  3  FOR EACH ROW
  4  BEGIN
  5  DBMS_OUTPUT.PUT_LINE('HELLO');
  6  END;
  7  /
Trigger created.
```

TESTING:

```
SQL> UPDATE TEST SET SAL=18000 WHERE SAL=12000;
HELLO
HELLO
2 rows updated.
```

ii) statement level triggers:
==========
- in this level a trigger body is executed only one time for a dml operation.

Ex:
```
SQL>CREATE OR REPLACE TRIGGER TR1
    AFTER UPDATE ON TEST
    BEGIN
    DBMS_OUTPUT.PUT_LINE('HELLO');
    END;
    /
Trigger created.
```

TESTING:
```
SQL> UPDATE TEST SET SAL=10000 WHERE SAL=18000;
HELLO

2 rows updated.
```

BIND VARIABLES:
==========
- these variables are working just like normal variables to store values while inserting,updating,deleting data from a table.these are two types,

i) :NEW :
==========
- when we insert a new row into a table those new values are storing in :new bind variable.

syntax:
==========
:new.<column name>

ii) :OLD:
==========
- when we delete old values from a table those old values are storing in :old bind variable.

syntax:
==========
:old.<column name>

NOTE:
==========
- these bind variables are used in row level triggers only.

To raise user defined alerts along with security:
==========
EX:
```
SQL> CREATE OR REPLACE TRIGGER TR1
  2  AFTER INSERT ON TEST
  3  BEGIN
```

```
  4  DBMS_OUTPUT.PUT_LINE('SOME ONE INSERTED A ROW INTO YOUR TABLE.PLZ
CHECK IT !!!');
  5  END;
  6  /
Trigger created.

TESTING:
SQL> INSERT INTO TEST VALUES(1022,'AALEN',45000);---ALLOWED
SOME ONE INSERTED A ROW INTO YOUR TABLE.PLZ CHECK IT !!!
1 row created.

       - in the above trigger we raised a user defined alert but
not restricted a dml operation on test table.so we need to
restrict a dml operation on test table then we use a
statement is called as"raise_application_erro()".

EX:
SQL> CREATE OR REPLACE TRIGGER TR1
  2  AFTER INSERT ON TEST
  3  BEGIN
  4  RAISE_APPLICATION_ERROR(-20478,'SOME ONE INSERTED A ROW INTO YOUR
TABLE.PLZ CHECK IT !!!');
  5  END;
  6  /

Trigger created.

SQL> INSERT INTO TEST VALUES(1023,'WARD',32000);--NOT ALLOWED
ERROR at line 1:
ORA-20478: SOME ONE INSERTED A ROW INTO YOUR TABLE.PLZ CHECK IT !!!

EX:
SQL> CREATE OR REPLACE TRIGGER TR1
    AFTER UPDATE ON TEST
    BEGIN
    RAISE_APPLICATION_ERROR(-20478,'SOME ONE UPDATED A ROW IN YOUR
TABLE.PLZ CHECK IT !!!');
    END;
    /
SQL> UPDATE TEST SET SAL=28000 WHERE SNO=1021;---NOT ALLOWED
ERROR at line 1:
ORA-20478: SOME ONE UPDATED A ROW IN YOUR TABLE.PLZ CHECK IT !!!

EX:
SQL> CREATE OR REPLACE TRIGGER TR1
    AFTER DELETE ON TEST
    BEGIN
    RAISE_APPLICATION_ERROR(-20478,'SOME ONE DELETED A ROW FROM YOUR
TABLE.PLZ CHECK IT !!!');
    END;
    /

SQL> DELETE FROM TEST WHERE SNO=1022;---NOT ALLOWED
ERROR at line 1:
```

```
ORA-20478: SOME ONE DELETED A ROW FROM YOUR TABLE.PLZ CHECK IT !!!

TRIGGER WITH ALL DML OPERATIONS:
================================
EX:
SQL> CREATE OR REPLACE TRIGGER TR1
    AFTER INSERT OR UPDATE OR DELETE ON TEST
    BEGIN
    RAISE_APPLICATION_ERROR(-20478,'SOME ONE PERFORMING DML OPERATIONS ON
YOUR TABLE.PLZ CHECK IT !!!');
    END;
    /
TESTING:
SQL> INSERT INTO TEST VALUES(1023,'WARD',32000);--NOT ALLOWED
SQL> UPDATE TEST SET SAL=28000 WHERE SNO=1021;---NOT ALLOWED
SQL> DELETE FROM TEST WHERE SNO=1022;---NOT ALLOWED
ERROR at line 1:
ORA-20478: SOME ONE PERFORM DML OPERATIONS ON YOUR TABLE.PLZ CHECK IT !!!

2. to control / restricted dml operations by using
business logical conditions:
============================
Ex:
create a trigger to restrict all dml operations on test table
on every monday?
SQL> CREATE OR REPLACE TRIGGER TRDAY
  2  AFTER INSERT OR UPDATE OR DELETE ON TEST
  3  BEGIN
  4  IF TO_CHAR(SYSDATE,'DY') = 'MON' THEN
  5  RAISE_APPLICATION_ERROR(-20569,'WE CANNOT PERFORM DML OPERATIONS ON
MONDAY');
  6  END IF;
  7  END;
  8  /

Trigger created.

EX:
create a trigger to restricted all dml operations on a table
between 9am to 5pm?
SQL> CREATE OR REPLACE TRIGGER TRTIME
  2  AFTER INSERT OR UPDATE OR DELETE ON TEST
  3  BEGIN
  4  IF TO_CHAR(SYSDATE,'HH24') BETWEEN 9 AND 16 THEN
  5  RAISE_APPLICATION_ERROR(-20657,'INVALID TIME');
  6  END IF;
  7  END;
  8  /

LOGIC:
======
              9AM(9) = 9:00:00 TO 9:59:59 --- UNDER 9 O CLOCK
              5PM(17)= 5:00:00 TO 5:59:59 --- UP TO 6 O CLOCK
              4PM(16)= 4:00:00 TO 4:59:59 --- UP TO 5 O CLOCK
```

to validating data:
==========
EX:
create a trigger to control insert operation on a table if
new salary is less than to 10000?

```
SQL> CREATE OR REPLACE TRIGGER TRSAL
  2  BEFORE INSERT ON TEST
  3  FOR EACH ROW
  4  BEGIN
  5  IF :NEW.SAL < 10000 THEN
  6  RAISE_APPLICATION_ERROR(-20574,'NEW SAL SHOULD NOT BE LESS THAN TO
10000');
  7  END IF;
  8  END;
  9  /

Trigger created.
```

TESTING:

```
SQL> INSERT INTO TEST VALUES(1023,'ADAMS',9500);--NOT ALLOWED
SQL> INSERT INTO TEST VALUES(1023,'ADAMS',12000);--ALLOWED
```

EX:
create a trigger to control delete operation on a table if
we try delete the employee "smith" details?

```
SQL> CREATE OR REPLACE TRIGGER TRNAME
  2  BEFORE DELETE ON TEST
  3  FOR EACH ROW
  4  BEGIN
  5  IF :OLD.NAME='SMITH' THEN
  6  RAISE_APPLICATION_ERROR(-20478,'NOT ALLOWED');
  7  END IF;
  8  END;
  9  /
```

TESTING:

```
SQL> DELETE FROM TEST WHERE NAME='ADAMS';----ALLOWED
SQL> DELETE FROM TEST WHERE NAME='SMITH';---NOT ALLOWED
```

auditing:
==========
- when we perform transactions(i/u/d) on a table those
transactional values are stored in an other table in database
is called as "audit table".

EX:

```
SQL> CREATE TABLE EMP1(EID INT,ENAME VARCHAR2(10),SAL NUMBER(10));
Table created.

SQL> CREATE TABLE EMP1_AUDIT(EID INT,AUDIT_INFO VARCHAR2(100));
Table created.

SQL> SELECT * FROM EMP1;
no rows selected
```

```
SQL> SELECT * FROM EMP1_AUDIT;
no rows selected

SQL> CREATE OR REPLACE TRIGGER TR_AUDIT
  2  BEFORE INSERT ON EMP1
  3  FOR EACH ROW
  4  BEGIN
  5  INSERT INTO EMP1_AUDIT VALUES(:NEW.EID,'SOME ONE INSERTED A NEW ROW
INTO YOUR TABLE ON'
  6  ||' '||TO_CHAR(SYSDATE,'DD-MON-YYYY HH:MI:SS AM'));
  7  END;
  8  /
Trigger created.

TESTING:
SQL> INSERT INTO EMP1 VALUES(1021,'SMITH',58000);
1 row created.

OUTPUT:
SQL> SELECT * FROM EMP1;

       EID ENAME             SAL
---------- ---------- ----------
      1021 SMITH           58000

SQL> SELECT * FROM EMP1_AUDIT;

       EID AUDIT_INFO
----------
--------------------------------------------------------------------------------
--------------------------
      1021 SOME ONE INSERTED A NEW ROW INTO YOUR TABLE ON 13-DEC-2022
05:07:55 PM

EX:
SQL>CREATE OR REPLACE TRIGGER TR_AUDIT
    BEFORE UPDATE ON EMP1
    FOR EACH ROW
    BEGIN
    INSERT INTO EMP1_AUDIT VALUES(:OLD.EID,'SOME ONE UPDATED A ROW INTO
YOUR TABLE ON'
    ||' '||TO_CHAR(SYSDATE,'DD-MON-YYYY HH:MI:SS AM'));
    END;
    /
TESTING:
SQL> UPDATE EMP1 SET SAL=12000 WHERE EID=1021;

EX:
    CREATE OR REPLACE TRIGGER TR_AUDIT
    BEFORE DELETE ON EMP1
    FOR EACH ROW
    BEGIN
```

```
    INSERT INTO EMP1_AUDIT VALUES(:OLD.EID,'SOME ONE DELETED A ROW FROM
YOUR TABLE ON'
    ||' '||TO_CHAR(SYSDATE,'DD-MON-YYYY HH:MI:SS AM'));
    END;
       /

TESTING:
SQL> DELETE FROM EMP1 WHERE EID=1022;

HOW TO DROP A DML TRIGGER:
==========================
SYNTAX:
========
DROP TRIGGER <TRIGGER NAME>;

EX:
DROP TRIGGER TRSAL;

DDLTRIGGERS:
============
       - WHEN WE CREATED A TRIGGER BASED ON DDL COMMANDS(CREATE,
ALTER,RENAME,DROP) THEN WE CALLED AS DDLTRIGGERS.
       - DDL TRIGGERS ARE WORKING ON A PARTICULAR DATABASE.SO THAT
DDLTRIGGER ARE ALSO CALLED AS "DB TRIGGERS".

SYNTAX:
=======
CREATE [OR REPLACE] TRIGGER <TRIGGER NAME>
BEFORE / AFTER  CREATE OR ALTER OR RENAME OR DROP
ON <USERNAME>.SCHEMA
BEGIN
<TRIGGER BODY / STATEMENTS>;
END;
/

EX:
CREATE A DDLTRIGGER TO RESTRICTED CREATE OPERATION ON MYDB4PM
DATABASE?
SQL> CREATE OR REPLACE TRIGGER TRDDL
  2  BEFORE CREATE ON MYDB4PM.SCHEMA
  3  BEGIN
  4  RAISE_APPLICATION_ERROR(-20478,'WE CANNOT PERFORM CREATE COMMAND ON
MYDB4PM');
  5  END;
  6  /

SQL> CREATE TABLE T1(SNO INT);
ERROR at line 1:
ORA-20478: WE CANNOT PERFORM CREATE COMMAND ON MYDB4PM

HOW TO DROP DDL TRIGGER:
========================
SYNTAX:
=======
```

```
DROP TRIGGER <TRIGGER NAME>

EX:
DROP TRIGGER TRDDL;
```
