ORACLE

Oracle content: (2 months)

- Topic-1 : DBMS
- Topic-2 : ORACLE
- Topic-3 : SQL
  - Introduction to SQL
  - Sub - Languages of SQL
  - Datatypes in oracle sql
  - Operators in oracle sql
  - Functions in oracle sql
  - Clauses in oracle sql
  - Jonis
  - Constraints
  - Subqueries
  - Views
  - Sequence
  - Indexes----------Interview level
- Topic-4 : Normalization
  - What is Normalization
  - Where we want to use Normalization
  - Why we need Normalization
  - Types of Normalization
    - > First normal form
    - > Second normal form
    - > Third normal form
    - > BCNF (Boyce-codd normal form)
    - > Fourth normal form
    - > Fifth normal form
- Topic-5 : PL/SQL
  - Introduction to PL/SQL
  - Difference between SQL and PL/SQL
  - Conditional & Looping statements
  - Cursors
  - Exception Handling
  - Stored procedures
  - Stored functions
  - Triggers

About Full stack java developer:

- In IT field a user is interacting the following two types of applications.
    1. Front end application
    2. Back end application

1. Front end application:
==========
- FEA is an application where the end-users are interacting to an application directly.
    Ex: Register form,Login Form,View profile form,Home page,.....etc

Design & Develop:
==========
- UI technologies(Html,Css,Javascript,AngularJS,React JS,Jquery,Json,....etc)

2. Back end application:
==========
- BEA is an application where the end-uses data/information is stored.
    Ex: Database.

Design & Develop:
==========
- DB technologies(Oracle,SQLserver,Mysql,PostgreSQL,DB2,.....etc)

Server Side Technologies:
==========
- these technologies are used to establish a connection between front end application and back end application.
    Ex: Java, .Net / .Net core , Python,.....etc

==========

Topic-1 : DBMS
==========

What is Data?
==========
- it is a rawfact.(i.e characters ,numbers,special characters and symbols)
- data is never give meaningfull statements to users.

Ex:

10001 is data	SMITH is data
10002 is data	ALLEN is data
10003 is data	MILLER is data

What is Information?
==========

- processing data is called as "Information".
- information is always provide meaningfull statements to users.

Ex:

Customer_ID	Customer_NAME
10001	SMITH
10002	ALLEN
10003	MILLER

What is Database ?
==========

- it is a memory which is used to store the collection of inter-related data/information of a particular business organization.

EX:

SBI_BANK_DB
> group of branches----------> group of customers
> group of departments
> group of employees

What is inter-related data/information?
==========

- depending on each other is called as inter-related.

Ex:

No employees = No departments
No departments = No employees

Ex:

No products = No customers
No customers = No products

Types of Databases?
==========

- there are two types of databases in real world.
  1. OLTP(online transaction processing)
  2. OLAP(online analytical processing)

1. OLTP:
==========

- these databases are used for storing "day-to-day" transactional information.
  Ex: oracle,sqlserver,mysql,postgresql,db2,.....etc

2. OLAP:

=======

- these databases are used for storing "historical data".(i.e Bigdata)
    Ex: Datawarehouse

What is DBMS?
============

- it is a software which is used to manage and maintain data/information with in the database.
- by using DBMS s/w we will perform the following operations are,
    > Creating Database
    > Creating Tables
    > Inserting data
    > Updating data
    > Selecting data
    > Deleting data
- Here DBMS s/w will act as an interface between User and Database.
    User <----------> DBMS s/w <---------> Database

Models of DBMS?
===============

- there are three types of DBMS models.
    1. Hierarchical Database management system(HDBMS)
        s/w : IMS(information management system)
    2. Network Database management system(NDBMS)
        s/w : IDBMS(integrated database management system)

NOTE:
======

- HDBMS,NDBMS model are outdated in real time.

3) Realtional Database Management System(RDBMS):
==========

- there are two modules in RDBMS.
    i) Object Relational DBMS(ORDBMS):
    ==========
        - these databases are storing data in the form of "Table" format.
        > a Table = collection of rows & columns.
        > a Row = group of columns
    - a row can be called as "record/tuple".
    - a column can be called as " attribute / field".
    - object relational databases are depends on "SQL".so that these databases are called as "SQL Databases".

Ex: Oracle,SQLserver,Mysql,PostgreSQL,DB2,.....etc

ii) Object Oriented DBMS(OODBMS):
==========
- these databases are storing data in the form of "Object".
- these databases are depends on "OOPS" concept but not SQL.so that these are called as "NoSql Databases".

Ex: MongoDB,Cassandra,.....etc

==========

Topic-2 : ORACLE
==========

Introduction to Oracle:
==========
- Oracle is an RDBMS(ORDBMS) product which was introduced by "Oracle Corporation" in 1979.
- Oracle is used to store data permanently (i.e Hard disk) along with security manner.
- When we want to deploy oracle s/w in the system then we need a platform.

What is Platform:
==========
- it is combination of operating system and micro-processor.
- there are two types of platforms.

i) Platform Dependent:
==========
- it supports only one operating system with the combination of any micro-processor.

Ex: Cobal,Pascal,C,C++.

ii) Platform Independent:
==========
- it supports any operating system with the combination of any micro-processor.

Ex: Oracle,Java,.Net core,Python,.....etc

- Oracle s/w can be installed in any operating system such as windows OS,Linux,Mac,Solaries OS,.....etc.
- Oracle is a platform independent an RDBMS product.

Versions of Oracle:
==========

- the first version of oracle s/w is "oracle1.0".
    - > Oracle1.0
    - > Oracle2.0
    - > Oracle3.0
    - > Oracle4.0
    - > Oracle5.0
    - > Oracle6.0
    - > Oracle7.0
    - > Oracle8.0
    - > Oracle8i (internet)
    - > Oracle9i
    - > Oracle10g (grid technology)
    - > Oracle11g
    - > Oracle12c (cloud technology)
    - > Oracle18c
    - > Oracle19c (latest version)
    - > Oracle21c (very latest version)
    - > Oracle23c(Beta version)

Working with Oracle:
==========

- when we are installing oracle s/w internally there are two components are installed in the system automatically.
    1. Client component
    2. Server component

1. Client component:
==========

- by using client tool we will perform the following the three operations.

    Step1: User can connet to oracle server:
        Enter username : system (default username)
        Enter password : tiger (created at oracle s/w installation)
        connected.

    Step2: User send request to oracle server:
        Request : SQL query / SQL command

    Step3: User will get response from oracle server:
        Response : Result / Output

Ex: SQLplus,SQLDeveloper,Toad are client tools of oracle.

2. Server component:
==========
- Server is having two more sub-components internally.
  i) Instance:
  ==========
    - It is a temporary memory which will allocate from RAM.
    - Here data can be stored temporarly.

  ii) Database:
  ==========
    - It is a permanent memory which will allocate from Harddisk.
    - Here data can be stored permanently.

TYPES OF ORACLE S/W EDITIONS:
==========
- there are two editions in oracle s/w.
  i) Oracle express edition:
  ==========
    - suppoting partial features of oracle database.
  Ex: Recyclebin,flashback,purge,partition table,.....etc
  are not allowed.

  ii) Oracle enterprise edition:
  ==========
    - supporting all features of oracle database.
  Ex: Recyclebin,flashback,purge,partition table,.....etc
  are allowed.

How to download oracle19c enterprise edition s/w:
==========
How to installing oracle19c enterprise edition s/w:
==========

NOTE:
=====
- when we want to work on oracle database then we follow the following two stpes procedure.
  i) Connect:
  ==========
    - when user want connect to oracle server then user required a DB client tool is "SQLPLUS".

ii) Communicate:
==========
- when we want to communicate with database then we need a DB language is "SQL".

SQLPLUS vs SQL:
==========

SQLPLUS	SQL
1. it is a DB client tool which was introduced by "oracle corporation".	1. it is a DB language which was introduced by "IBM".
2. it is used to connect to oracle server.	2. it is used for communicate with database.
3. it will act as an editor for writting & executing SQL queries & PL/SQL programs.	3. it again five sub languages are(DDL,DML,DQL/DRL,TCL,DCL) used to perform some operations over database.

How to connect to oracle server:
==========
> go to all programs
> go to oracle-oradb19home-1 folder
> click on SQLPLUS icon
Enter username : system(default user)
Enter password : tiger (created at installation)
connected.

NOTE:
=====
- Here username is not a case sensitive but password is a case-sensitive.

How to create a new username and password in oracle server:
==========
syntax:
=======
create user <username> identified by <password>;

EX:
> go to open SQLPLUS

Enter username : system/tiger
connected.

SQL> CREATE USER MYDB9AM IDENTIFIED BY 123;
User created.

Enter user-name: MYDB9AM/123
ERROR:
ORA-01045: user MYDB9AM lacks CREATE SESSION privilege; logon denied.

Granting "DBA' permissions to the user(MYDB9AM):
==========
syntax:
======
GRANT <PRIVILEGE NAME> TO <USERNAME>;

EX:
Enter user-name: SYSTEM/TIGER
connected.

SQL> GRANT DBA TO MYDB9AM;
Grant succeeded.

SQL> CONN
Enter user-name: MYDB9AM/123
Connected.

How to change password for user(mydb9am):
==========
syntax:
=======
PASSWORD;

EX:
SQL> CONN
Enter user-name: MYDB9AM/123
Connected.

SQL> PASSWORD;
Changing password for MYDB9AM
Old password:123
New password:ABC
Retype new password:ABC

Password changed

SQL> CONN
Enter user-name: MYDB9AM/ABC
Connected.

How to re-create a new password for user,if we forgot it:
==========
syntax:
=======
ALTER USER <USERNAME> IDENTIFIED BY <NEW PASSWORD>;

Ex:
SQL> CONN
Enter user-name: SYSTEM/TIGER
Connected.

SQL> ALTER USER MYDB9AM IDENTIFIED BY MYDB9AM;
User altered.

SQL> CONN
Enter user-name: MYDB9AM/MYDB9AM
Connected.

How to re-create a new password for SYSTEM admin if we forgot it:
==========
syntax:
=======
ALTER USER <USERNAME> IDENTIFIED BY <NEW PASSWORD>;

EX:
SQL> CONN
Enter user-name: \SYS AS SYSDBA (default username)
Enter password : SYS (default password)
Connected.

SQL> ALTER USER SYSTEM IDENTIFIED BY LION;
User altered.

SQL> CONN
Enter user-name: SYSTEM/LION
Connected.

How to view username in oracle if we forgot it:
==========
syntax:
======
SELECT USERNAME FROM ALL_USERS; (all_users is a pre-defined table)

EX:
SQL> CONN
Enter user-name: SYSTEM/LION
Connected.

SQL> SELECT USERNAME FROM ALL_USERS;

How to drop a user from oracle:
==========
syntax:
======
DROP USER <USERNAME> CASCADE; (cascade is a pre-defined keyword)

Ex:
SQL> CONN
Enter user-name: SYSTEM/LION
Connected.

SQL> DROP USER MYDB9AM CASCADE;
User dropped.

To clear the screen:
==========
syntax:
=======
CL SCR;

To exit form oracle(i.e disconnect):
==========
syntax:
=======
EXIT;
==========

Topic-3 : SQL
==========

Introduction to SQL:
==========

- SQL stands for "structure query language" which was introduced by IBM.
- SQL is used to communicate with DATABASE.
- The initial name is "SEQUEL" and later renamed as "SQL".
- SQL is not a case-sensitive language i.e we will write SQL queries either upper / lowe / combination of lower & upper case characters.

    Ex:

    SQL> select * from emp; -------executed

    SQL> SELECT * FROM EMP;------EXECUTED

    SQL> SeleCT * From Emp;--------Executed

- Every sql query should ends with " ; ".

Sub - Languages of SQL:
==========

- there are five sub-languages of SQL.

1) Data Definition Language(DDL):
==========

- CREATE
- ALTER
    > ALTER - MODIFY
    > ALTER - ADD
    > ALTER - RENAME
    > ALETR - DROP
- RENAME
- TRUNCATE
- DROP

New Features / New commands:
==========

- RECYCLEBIN
- FLASHBACK
- PURGE

2) Data Manipulation Language(DML):
==========

- INSERT
- UPDATE
- DELETE

Note:
=====

- DML commands are called as "Write only operations".

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

==========
- SELECT (read only)

4) Transaction Control Language(TCL):
==========
- COMMIT
- ROLLBACK
- SAVEPOINT

5) Data Control Language(DCL):
==========
- GRANT
- REVOKE

==========

1) Data Definition Language(DDL):
==========
CREATE:
=======
- to create a new database object such as Table,View,Sequence,Procedure, Functions,Triggers,.....etc.

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

Datatypes in oracle:
==========
- it is an attribute which will specify "what type of data" is storing into a column in the table.
- oracle supports the following datatypes are,
    i) Number datatype
    ii) Character / String datatypes
    iii) Long datatype
    iv) Date datatypes
    v) Raw & Long Raw datatypes
    vi) LOB datatypes

i) Number datatype:
==========
- storing integer & float values.

- it contains two arguments those are Precision and Scale.

NUMBER(P,S):
==========
> NUMBER(P) : store integer format values only.
> NUMBER(P,S) : store float values only.

Precision(p):
==========
- counting all digits in the given expression.
- the maximum size is 38 digits.

Ex:
i) 10
precision = 2

ii) 86745
precision = 5

Ex:
SNO number(7)
==========
0
1
2

9999999
10000000--------error

Scale(s):
========
- counting the right side digits of a decimal point from the given expression.
- there is no maximum size of scale because it is a part of precision value.

Ex:
56.23
precision = 4
scale = 2

7584.20
precision = 6
scale = 2

Ex:

PRICE number(8,2)
==========
0.0
56.23
95.23

999999.99
1000000(1000000.00)----------error

Character / String datatypes:
==========

- storing string format data only.
- in database string can be represent with '<string>'.

Ex: Emp_Name char(10)
==========

smith------> error
'smith'----> smith
1021-------> error
'1021'-----> 1021
56.12------> error
'56.12'----> 56.12

string format data
||
characters only string
||
[A - Z / a - z]

alphanumeric string
||
[A-Z / a-z , 0 - 9,@,#,$,%,&,_,...etc]

Ex: 'SMITH','smith',.....etc Ex: 'smith123@gmail.com',PASSWORD,PANCARD,HTNO,.....etc

Types of character / string datatypes:
==========

- there are two types of string datatypes.

1) Non-unicode datatypes
==========

- supporting to store "localized data".(i.e English Language only)

i) char(size)
ii) varchar2(size)

2) Unicode datatypes

==========

- supporting to store "globalized data".(i.e All National Languages)
    i) Nchar(size)
    ii) Nvarchar2(size)
- Here "N" stands for National Language.

i) char(size):
==========

- it is fixed length datatype(i.e static datatype)
- it will store non-unicode characters in the form of 1 char = 1 byte.
- the maximum size is 2000 bytes.

Disadvantage:
==========

- memory wasted.

ii) varchar2(size):
==========

- it is variable length datatype(i.e dynamic datatype)
- it will store non-unicode characters in the form of 1 char = 1 byte.
- the maximum size is 4000 bytes.

Advantage:
==========

- memory saved.

i) Nchar(size):
==========

- it is fixed length datatype(i.e static datatype)
- it will store unicode characters in the form of 1 char = 1 byte.
- the maximum size is 2000 bytes.

Disadvantage:
==========

- memory wasted.

ii) Nvarchar2(size):
==========

- it is variable length datatype(i.e dynamic datatype)
- it will store unicode characters in the form of 1 char = 1 byte.
- the maximum size is 4000 bytes.

Advantage:

==========

- memory saved.

iii) Long datatype:

==========

- it is a variable length datatype(i.e dynamic datatype)
- it will store non-unicode & unicode characters in the form of 1 char = 1 byte.
- a table is having only one long datatype column.
- the maximum size is 2gb.

Date datatypes:

==========

- to store date & time information of a particular day / transaction.
- the range form '01-JAN-4712 BC' to '31-DEC-9999 AD'.
  - i) DATE
  - ii) TIMESTAMP

i) DATE:

=======

- storing date & time information but time is a optional.
- if user not insert time information then oracle server will take '00:00:00am' by default.
- the default date format in oracle database is 'DD-MON-YY/YYYY HH:MI:SS'.

Ex:

```
'DD-MON-YY/YYYY  HH:MI:SS'
' 09-OCT-24/2024     10:07:xx '
  1     1         2           1   1    1 ----------> 7 bytes fixed memory.
```

ii) TIMESTAMP:

==========

- storing date & time information including milliseconds.
- the default timestamp format in oracle database is 'DD-MON-YY/YYYY HH:MI:SS.MS'.

Ex:

```
'DD-MON-YY/YYYY  HH:MI:SS.MS'
' 09-OCT-24/2024     10:07:xx.xxxx '
  1     1         2           1   1    1    4  ----------> 11 bytes fixed memory.
```

Raw & Long Raw datatypes:

==========

- storing image / audio / video file in the form 010010101010 binary format.
  - Raw   : static datatype   : 2000 bytes
  - LongRaw   : dynamic datatype   : 2gb

LOB datatypes:
==========

- LOB stands for large objects.
    - i) CLOB
    - ii) NCLOB
    - iii) BLOB

i) CLOB:
=======

- it stands for "character large object" datatype.
- it is a dynamic datatype.
- it will store non-unicode characters in the form of 1 char = 1 byte.
- the maximum size is 4gb.

ii) NCLOB:
=======

- it stands for "national character large object" datatype.
- it is a dynamic datatype.
- it will store unicode characters in the form of 1 char = 1 byte.
- the maximum size is 4gb.

iii) BLOB:
========

- it stands for "binary large object" datatype.
- it is a dynamic datatype.
- it will store image / audio / video file in the form of 01001010101 binary format.
- the maximum size is 4gb.

Non-unicode characters:
==========

- char(size) - 2000 bytes
- varchar2(size) - 4000 bytes
- long - 2gb
- clob - 4gb

Unicode characters:
==========

- Nchar(size) - 2000 bytes
- Nvarchar2(size) - 4000 bytes
- long - 2gb
- Nclob - 4gb

Binary format data:
==========
- Raw - 2000 bytes
- LongRaw - 2gb
- Blob - 4gb

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

EX:
SQL> CONN
Enter user-name: MYDB9AM/MYDB9AM
Connected.

SQL> CREATE TABLE STUDENT(STID NUMBER(4),SNAME CHAR(10),SFEE NUMBER(6,2));
Table created.

To view the structure of a table:
==========
syntax:
=======
DESC <table name>;

Ex:
SQL> DESC STUDENT;

To view the list of tables in oracle database:
==========
syntax:
=======
SELECT * FROM TAB; (TAB = Pre-defined table)

Ex:
SQL> SELECT * FROM TAB;

ALTER command:
==========
- to change / modify the structure of a table.

- it again four sub-commands those are,

i) ALTER - MODIFY:
==========
- to change datatype and also the size of datatype of a specific column in the table.

syntax:
=======
ALTER TABLE <TABLE NAME> MODIFY <COLUMN NAME> <new datatype>[new size];

EX:
SQL> ALTER TABLE STUDENT MODIFY SNAME VARCHAR2(20);

ii) ALTER - ADD:
==========
- to add a new column to an existing table.

syntax:
======
ALTER TABLE <TABLE NAME> ADD <NEW COLUMN NAME> <DATATYPE>[SIZE];

EX:
SQL> ALTER TABLE STUDENT ADD SADDRESS LONG;

iii) ALTER - RENAME:
==========
- to change a column name in the table.

syntax:
=======
ALTER TABLE <TABLE NAME> RENAME <column> <OLD COLUMN NAME> TO <NEW COLUMN NAME>;

EX:
SQL> ALTER TABLE STUDENT RENAME COLUMN SADDRESS TO SADD;

iv) ALTER - DROP:
==========
- to delete a column from a table permanently.

syntax:
======
ALTER TABLE <TABLE NAME> DROP <column> <COLUMN NAME>;

EX:
SQL> ALTER TABLE STUDENT DROP COLUMN SFEE;

RENAME command:
==========
- 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;

TRUNCATE command:
==========
- deleting all rows but not columns from a table.
- we cannot delete a specific row because truncate command is not allowed "WHERE" clause condition.

syntax:
======
TRUNCATE TABLE <TABLE NAME>;

EX:
SQL> TRUNCATE TABLE STUDENT WHERE STID=1022;----------NOT ALLOWED
SQL> TRUNCATE TABLE STUDENT;----------ALLOWED

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

syntax:
=======
DROP TABLE <TABLE NAME>;

EX:
SQL> DROP TABLE STUDENT;

NOTE:
=====
- Before oracle10g enterprise edition once we drop a table then it was permanently dropped whereas from oracle10 enterprise edition once we drop a table then it was temporarly dropped.

Oracle10g enterprise edition features:
==========
i) RECYCLEBIN:
==========
- it is a pre-defined table in oracle.
- it will store the information about deleted tables from database.
- it is similar to windows recyclebin in computer.

syntax:
=======
SELECT OBJECT_NAME,ORIGINAL_NAME FROM RECYCLEBIN;

OBJECT_NAME ORIGINAL_NAME

BIN$NcRYMukARku03KsVTt+x+Q==$0 STUDENT

ii) FLASHBACK:
==========
- it is a DDL command which is used to restore a deleted table from recyclebin to database.

syntax:
======
FLASHBACK TABLE <TABLE NAME> TO BEFORE DROP;

EX:
SQL> FLASHBACK TABLE STUDENT TO BEFORE DROP;

iii) PURGE:
=========
- this statement is used to delete a table from database permanently.

syntax:
======
DROP TABLE <TABLE NAME> PURGE;

EX:
SQL> DROP TABLE STUDENT PURGE;

NOTE:
=====
- the above features are working under USER(mydb9am) account but not in DBA(system) account.

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

Case-1: Inserting all columns values:
==========
syntax:
=======
INSERT INTO <TABLE NAME> VALUES(value1,value2,.....);

EX:
SQL> INSERT INTO STUDENT VALUES(1021,'ALLEN',2500);

Case-2: Inserting specific columns values:
==========
syntax:
======
INSERT INTO <TABLE NAME>(required column names)VALUES(value1,value2,.....);

EX:
SQL> INSERT INTO STUDENT(STID)VALUES(1021);
SQL> INSERT INTO STUDENT(SNAME,SFEE)VALUES('SMITH',2500);
SQL> INSERT INTO STUDENT(STID,SNAME,SFEE)VALUES(1023,'ALLEN',4500);
SQL> INSERT INTO STUDENT(SNAME,SFEE,STID)VALUES('JONES',3900,1024);

How to insert values into a table dynamically: (multiple rows)
==========
- when we insert values into a table dynamically then we use a special operator is "&".

Case-1: Inserting all columns values:
==========
syntax:
=======
INSERT INTO <TABLE NAME> VALUES(<&column name1>,<&column name2>,.....);

EX:
SQL> INSERT INTO STUDENT VALUES(&STID,'&SNAME',&SFEE);
Enter value for stid: 1025
Enter value for sname: ADAMS

Enter value for sfee: 5400

SQL> / (To re-execute the lastly executed sql query in sqlplus editor)
Enter value for stid: 1026
Enter value for sname: JAMES
Enter value for sfee: 8500

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

Case-2: Inserting specific columns values:
==========
syntax:
======
INSERT INTO <TABLE NAME>(required column names)VALUES(<&column name1>,.....);

EX:
SQL> INSERT INTO STUDENT(STID)VALUES(&STID);
Enter value for stid: 1028

SQL> /
.....

SQL> /
.....

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

syntax:
=======
UPDATE <TABLE NAME> SET <column name1>=<value1>,<column name2>=<value2>,
.....[WHERE <condition>];

EX:
SQL> UPDATE STUDENT SET SFEE=6000 WHERE SNAME='ADAMS';
SQL> UPDATE STUDENT SET SNAME='WARD',SFEE=3000 WHERE STID=1027;

SQL> UPDATE STUDENT SET SNAME=NULL WHERE SNAME='JONES';

SQL> UPDATE STUDENT SET STID=NULL,SNAME=NULL,SFEE=NULL WHERE STID=1026;
SQL> UPDATE STUDENT SET STID=1026,SNAME='JONES',SFEE=9000 WHERE STID IS NULL;

SQL> UPDATE STUDENT SET SFEE=NULL;
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" clause condition.

syntax:
======
DELETE FROM <TABLE NAME> [WHERE <condition>];

EX:
SQL> DELETE FROM STUDENT WHERE STID=1025;
SQL> DELETE FROM STUDENT WHERE SNAME IS NULL;
SQL> DELETE FROM STUDENT;

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

DELETE =======	TRUNCATE ==========
1. it is a DML operation.	1. it is a DDL operation.
2. deleting a specific row.	2. cannot delete a specific row.
3. supporting "WHERE" clause.	3. does not support "WHERE" clause.
4. deleting data temporarly.	4. deleting data permanently.
5. we can restore deleted data by using "ROLLBACK".	5. we cannot restore deleted data by using "ROLLBACK".
6. deleting rows in one-by-one manner.	6. deleting rows as a page wise.
7. execution speed is slow.	7. execution speed is fast.

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

==========
SELECT :
========
- to retrieve all rows from a table at a time.
(or)
- to retrieve a specific rows from a table by using "WHERE" clause condition.

syntax:
======
SELECT * FROM <TABLE NAME> [WHERE <condition>];

Here " * " is representing all columns in a table.

EX:
SQL> SELECT * FROM DEPT;
(or)
SQL> SELECT DEPTNO,DNAME,LOC FROM DEPT;

EX:
SQL> SELECT * FROM EMP WHERE JOB='MANAGER';
SQL> SELECT * FROM EMP WHERE EMPNO=7698;
SQL> SELECT ENAME,DEPTNO,SAL FROM EMP WHERE DEPTNO=20;

ALIAS NAMES:
============
- it is a temporary name / alternate name for columns / table / expression.
- we will create alias names at two levels.

i) column level alias:
==========
- creating alias name for columns.

ii) table level alias:
==========
- creating alias name for table name.

syntax:
=======
SELECT <column name1> [AS] <column alias name1>,<column name2> [AS] <column alias name2>,
..... FROM <table name> <table alias name>;

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

(OR)

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

DISTINCT keyword:
==========

- to eliminate duplicate values from a specific column.

syntax:
=======

distinct <column name>

EX:

```
SQL> SELECT DISTINCT DEPTNO FROM EMP;
SQL> SELECT DISTINCT JOB FROM EMP;
```

CONCATENATION OPERATOR(||):
==========

- to combined two or more than two expressions.

syntax:
======

<expression1> || <expression2> || <expression3> || .....

EX:

```
SQL> SELECT 'THE EMPLOYEE'||' '||ENAME||' '||'IS WORKING AS A'||' '||JOB FROM EMP;
```

NOTE:
======

- to display a table data in proper systematical way in sqlplus editor then we need to set the following two properties are,

i) PAGESIZE n:
==========

- to show no.of rows in a page.
- here "n" is represent no.of rows in a page.
- the maximum size of pagesize property is 50000 rows.

syntax:
======

SET PAGESIZE n;

ii) LINES n:
=========

- to display no.of characters in a single row / line.
- here "n" is represent no.of characters in a line.

- the maximum size of line property is 32767 characters.

syntax:
=======
SET LINES n;

EX:
SQL> SET PAGESIZE 100;
SQL> SET LINES 160;
SQL> SELECT * FROM EMP;
==========
==
Operators in oracle sql:
==========
- to perform some operations on the given operand values.
- oracle supports the following operators are,

- Assignment operator => =
- Arithematic operators => + , - , * , /
- Relational operators => < , > , <= , >= , != (or) < >
- Logical operators => AND,OR,NOT
- Set operators => UNION,UNION ALL,INTERSECT,MINUS

- Special operators => (+ve) (-ve)

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

Assignment operator:
==========
- to assign a value to variable / to attribute.

syntax:
======
<column name> <assignment operator> <value>

EX:
SQL> UPDATE EMP SET SAL=50000 WHERE EMPNO=7788;
SQL> UPDATE EMP SET JOB='HR';

Arithematic operators:

==========

- to perform addition,subtraction,multiple,division.

syntax:

======

<column name> <arithematic operator> <value>

Ex:

waq to display all employees salaries after adding 2000/-?

```
SQL> SELECT ENAME,SAL AS OLD_SALARY,SAL+2000 AS NEW_SALARY FROM EMP;
```

Ex:

waq to display EMPNO,ENAME,BASIC SALARY and ANNUAL SALARY of the employees who are working as a MANAGER?

```
SQL> SELECT EMPNO,ENAME,SAL AS BASIC_SALARY,SAL*12 AS ANNUAL_SALARY
  2  FROM EMP WHERE JOB='MANAGER';
```

Ex:

waq to display all employees salaries after increment of 10%?

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

Ex:

waq to display ENAME,DEPTNO,BASIC_SALARY,INCREMENT of 5% AMOUNT and TOTAL_SALARY of the employees who are working under deptno is 20?

```
SQL> SELECT ENAME,DEPTNO,SAL AS BASIC_SALARY,
  2  SAL*0.05 AS INCREMENT_AMOUNT,SAL+SAL*0.05 AS TOTAL_SALARY
  3  FROM EMP WHERE DEPTNO=20;
```

Ex:

waq to display EMPNO,ENAME,JOB,HIREDATE,BASIC_SALARY,10% of HRA,20% of DA, 5% of PF,GROSS_SALARY and also NET_SALARY of the employees?

```
SQL> SELECT EMPNO,ENAME,JOB,HIREDATE,SAL AS BASIC_SALARY,
  2  SAL*0.1 AS HRA,SAL*0.2 AS DA,SAL*0.05 AS PF,
  3  SAL+SAL*0.1+SAL*0.2+SAL*0.05 AS GROSS_SALARY,
  4  SAL+SAL*0.1+SAL*0.2-SAL*0.05 AS NET_SALARY
  5  FROM EMP;
```

EX:

waq to display all employees salaries after decrement of 10%?

```
SQL> SELECT ENAME,SAL AS BEFORE_DECREMENT,
  2  SAL-SAL*10/100 AS AFTER_DECREMENT FROM EMP;
```

Relational operators:

==========

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

syntax:
======

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

Ex:
waq to display list of employees who are joined after 1981?
SQL> SELECT * FROM EMP WHERE HIREDATE>'31-DEC-1981';

Ex:
waq to display list of employees who are joined before 1981?
SQL> SELECT * FROM EMP WHERE HIREDATE<'01-JAN-1981';

Logical operators:
==========

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

AND operator:
==========

- 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 who are working as "SALESMAN" and whose name is "TURNER"?
SQL> SELECT * FROM EMP WHERE JOB='SALESMAN' AND ENAME='TURNER';

OR operator:
==========

- it return a value if any one condition is true 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 employees whose EMPNO is 7369,7566,7788?
SQL> SELECT * FROM EMP WHERE EMPNO=7369 OR EMPNO=7566 OR EMPNO=7788;

EX:
waq to display employees who are working as "PRESIDENT" or whose salary is more than or is equals to 3000?
SQL> SELECT * FROM EMP WHERE JOB='PRESIDENT' OR SAL>=3000;

NOT operator:
============
- it return all values except the given conditional values in the query.
syntax:
=======
where NOT <condition1> AND NOT <condition2>;

Ex:
waq to display employees who are not working as a "MANAGER" and as a "ANALYST"?
SQL> SELECT * FROM EMP WHERE NOT JOB='MANAGER' AND NOT JOB='ANALYST';

Set operators:
============
- are used to combined the results of two select statements.
syntax:
=======
<select query1> <set operator> <select query2>;

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

I) UNION:
========
- to combined the values of two sets without duplicates.
A U B = {10,20,30,40,50}

II) UNION ALL:
==========

- to combined the values of two sets with duplicates.
  A UL B = {10,20,30,30,40,50}

III) INTERSECT:
==========

- it return the common values from both sets.
  A I B = {30}

IV) MINUS :
==========

- it return uncommon values from the left side set but not the right side set.
  A - B = {10,20}
  B - A = {40,50}

DEMO_TABLES:
==========

```
SQL> SELECT * FROM EMP_HYD;

       EID     ENAME          SAL
----------          ----------          ----------
1021    SMITH     85000
1022    ALLEN     35000
1023    MILLER    68000

SQL> SELECT * FROM EMP_MUMBAI;

       EID     ENAME          SAL
----------          ----------          ----------
1021    SMITH     85000
1024    WARD      38000
```

EX:
waq to fetch employees who are working in HYD but not in MUMBAI branch?

```
SQL> SELECT * FROM EMP_HYD MINUS SELECT * FROM EMP_MUMBAI;
```

EX:
waq to fetch employees who are working in both branches?

```
SQL> SELECT * FROM EMP_HYD INTERSECT SELECT * FROM EMP_MUMBAI;
```

EX:

waq to fetch all employees details who are working in the organization?
SQL> SELECT * FROM EMP_HYD UNION ALL SELECT * FROM EMP_MUMBAI;(including duplicate rows)
SQL> SELECT * FROM EMP_HYD UNION SELECT * FROM EMP_MUMBAI;(excluding duplicate rows)

Special operators:
==========
IN operator:
==========
- comparing the list of values with a single condition.

syntax:
======
where <column name> IN(value1,value2,.....);
where <column name> NOT IN(value1,value2,.....);

EX:
waq to display employees whose empno is 7566,7788,7900?
SQL> SELECT * FROM EMP WHERE EMPNO IN(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 operator:
==========
- comparing a particular range value.

syntax:
======
where <column name> BETWEEN <low value> AND <high value>;
where <column name> NOT BETWEEN <low value> AND <high value>;

EX:
waq to list out employees who are joined in the year of 1981?
SQL> SELECT * FROM EMP WHERE HIREDATE BETWEEN '01-JAN-1981' AND '31-DEC-1981';

EX:
waq to list out employees who are not joined in the year of 1981?
SQL> SELECT * FROM EMP WHERE HIREDATE NOT BETWEEN '01-JAN-1981' AND '31-DEC-1981';

NOTE:
=====
- BETWEEN operator return all values including source value and also destination value.

IS NULL operator:
==========
- comparing NULLS in a table.

syntax:
======
where <column name> IS NULL;
where <column name> IS NOT NULL;

EX:
waq to display employees whose commission is NULL / UNDEFINED / EMPTY / UNKNOWN?
SQL> SELECT * FROM EMP WHERE COMM IS NULL;

EX:
waq to display employees whose commission is NOT NULL / DEFINED / NOT EMPTY / KNOWN?
SQL> SELECT * FROM EMP WHERE COMM IS NOT NULL;

Working with NULL:
==========
- it is an empty / a undefined value / a unknown value in database.
- NULL != 0 , NULL != space.

Ex:
waq to display EMPNO,ENAME,JOB,SAL,COMM and SAL+COMM from emp table whose employee
name is "SMITH"?
SQL> SELECT EMPNO,ENAME,JOB,SAL,COMM,SAL+COMM AS TOTAL_SALARY
  2 FROM EMP WHERE ENAME='SMITH';

OUTPUT:
========
EMPNO ENAME JOB SAL COMM TOTAL_SALARY
---------- ---------- --------- ---------- ---------- ------------
7369 SMITH CLERK 800

- In the above example the employee SMITH salary is 800 and there is no commission so that SAL+COMM is 800 only but it return NULL.
- To overcome the above problem oracle will provide a pre-defined function is NVL().

What is NVL(exp1,exp2):
==========

- to replace a user defined value inplace of NULL in the expression.
- this function is having two arguments are expression1 and expression2.
  - > If EXP1 is NULL then it return EXP2 value(user defined value).
  - > If EXP1 is NOT NULL then it return EXP1 value only.

EX:

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

SQL> SELECT NVL(0,1000) FROM DUAL; ------------> 0
SQL> SELECT NVL(500,200) FROM DUAL; ----------> 500
```

Solution:
=======

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

OUTPUT:
========

```
EMPNO        ENAME    JOB        SAL     COMM          TOTAL_SALARY
----------         ----------         --------- ----------   ----------      ------------
7369   SMITH    CLERK     800              800
```

NVL2():
======

- it is an extension of NVL().
- it contains three arguments are expression1,expression2 and expression3.
  - > If exp1 in NULL then it return EXP3 value (user defined value)
  - > if exp1 is NOT NULL then it return EXP2 value(user defined value)

syntax:
=======

```
nvl2(exp1,exp2,exp3)
```

Ex:

```
SQL> SELECT NVL2(NULL,100,200) FROM DUAL;----------> 200
SQL> SELECT NVL2(500,100,200) FROM DUAL;-------------> 100
```

EX:

waq to update all employees commissions in the table based on the following conditions are,

i) if the employee COMM is NULL then update those employees commissions as 800.

ii) if the employee COMM is NOT NULL then update those employees commissions as COMM+300.

```
SQL> UPDATE EMP SET COMM=NVL2(COMM,COMM+300,800);
```

LIKE operator:
==========

- comparing a specific string character pattern wise.
- when we use "LIKE" operator we must use the following two wildcard operators are:
  - i) % - it represent the remaining group of characters after selected character from the expression.
  - ii) _ - counting a single character from the expression.

syntax:
======

```
where <column name> LIKE '[<wildcard operator>] <character pattern> [<wildcard operator>]'
```

Ex:
waq to fetch employees whose name starts with "S" character?

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

Ex:
to fetch employees whose name ends with "R" character?

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

Ex:
to fetch employees whose name is having "I" character?

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

Ex:
to fetch employees whose name starts with and ends with M and N?

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

Ex:
to fetch employees whose name is having the second position character is O?

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

Ex:
to fetch employees whose name is having 4 characters?

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

Ex:

to fetch employees whose empno starts with 7 and ends with 8?
SQL> SELECT * FROM EMP WHERE EMPNO LIKE '7%8';

Ex:
to fetch employees who are joined in 1981?
SQL> SELECT * FROM EMP WHERE HIREDATE LIKE '%81';

Ex:
to fetch employees who are joined in the month of "DECEMBER"?
SQL> SELECT * FROM EMP WHERE HIREDATE LIKE '%DEC%';

Ex:
to fetch employees who are joined in the month of "DECEMBER" in 1982?
SQL> SELECT * FROM EMP WHERE HIREDATE LIKE '%DEC%82';
(OR)
SQL> SELECT * FROM EMP WHERE HIREDATE LIKE '%DEC_82';
(OR)
SQL> SELECT * FROM EMP WHERE HIREDATE LIKE '%DEC-82';
(OR)
SQL> SELECT * FROM EMP WHERE HIREDATE LIKE '%DEC%' AND HIREDATE LIKE '%82';

Ex:
to fetch employees who are joined in the month of "JUNE","DECEMBER" ?
SQL> SELECT * FROM EMP WHERE HIREDATE LIKE '%JUNE%' OR HIREDATE LIKE
'%DEC%';

LIKE operator on special characters:
==========
DEMO_TABLE:
==========
SQL> SELECT * FROM TEST;

```
       CID CNAME
---------- --------------------
         1 BHUVIN_KUMAR
         2 WAR@NER
         3 #YUVIN
         4 MILL%ER
         5 _ADAMS
         6 JON%ES
```

EX:
waq to fetch customer details whose name is having "@" symbol?
SQL> SELECT * FROM TEST WHERE CNAME LIKE '%@%';

EX:
waq to fetch customer details whose name is having "#" symbol?
SQL> SELECT * FROM TEST WHERE CNAME LIKE '%#%';

EX:
waq to fetch customer details whose name is having "_" symbol?
SQL> SELECT * FROM TEST WHERE CNAME LIKE '%_%';

EX:
waq to fetch customer details whose name is having "%" symbol?
SQL> SELECT * FROM TEST WHERE CNAME LIKE '%%%';

- when we fetch data from a table based on " _ , % " symbols oracle server will treat as wildacard operators but not special characters.
- to overcome the above problem we must use a keyword is " ESCAPE'\' ".

Solution:
========
SQL> SELECT * FROM TEST WHERE CNAME LIKE '%_%'ESCAPE'\';
SQL> SELECT * FROM TEST WHERE CNAME LIKE '%\%%'ESCAPE'\';

Ex:
waq to fetch employees details whose name not starts with "S" character?
SQL> SELECT * FROM EMP WHERE ENAME NOT LIKE 'S%';
==========
=
FUNCTIONS IN ORACLE:
==========
- to perform some task as per the given input values and it must be return a value.
- oracle supports the following two types of functions.those are
    1. Pre-defined functions
        - Use in SQL & PL/SQL
    2. User-defined functions
        - Use in PL/SQL only.

1. Pre-defined functions:
==========
- these functions ara also called as "Built-In-functions" in oracle.
- it again two types:
    i) Single row functions
    ii) Multiple row functions

i) Single row functions:
==========
- these functions are always return a single value.
- there are few types of single row functions:
  - > Numeric functions
  - > Character / String functions
  - > Date functions
  - > Null functions (NVL(),NVL2())
  - > Conversion functions
  - > Analytical functions

How to call a function in oracle:
==========
syntax:
======
SELECT <FNAME>(value/(s)) FROM DUAL;

What is DUAL?
==========
- it is a pre-defined table in oracle.
- it is used to test function functionalities(i.e workflow).
- it contains a single row & a single column.
- it is also called as "Dummy Table" in oracle.

How to view the structure of dual table:
==========
syntax:
=======
DESC <table name>;

EX:
SQL> DESC DUAL;

How to view data of dual table:
==========
syntax:
=======
SELECT * FROM <table name>;

EX:
SQL> SELECT * FROM DUAL;

Numeric functions:
==========
ABS():
=====
- to convert (-ve) sign values into (+ve) sign values.

syntax:
======
abs(n)

EX:
```
SQL> SELECT ABS(-12) AS RESULT FROM DUAL;

    RESULT
----------
        12
```

EX:
waq to display ENAME,SALARY,COMMISSION and COMMISSION-SALARY from emp table?
```
SQL> SELECT ENAME,SAL,COMM,ABS(COMM-SAL) AS RESULT FROM EMP;
```

CEIL():
======
- it return a value which is greater than to (or) is equals to the given expression.

syntax:
=======
ceil(n)

EX:
```
SQL> SELECT CEIL(9.0) AS RESULT FROM DUAL;

    RESULT
----------
         9

SQL> SELECT CEIL(9.1) AS RESULT FROM DUAL;

    RESULT
----------
        10
```

FLOOR():
========
- it return a value which is less than to (or) is equals to the given expression.

syntax:
=======
floor(n)

EX:
SQL> SELECT FLOOR(9.0) AS RESULT FROM DUAL;

RESULT

9

SQL> SELECT FLOOR(9.8) AS RESULT FROM DUAL;

RESULT

9

EX
SQL> SELECT ENAME,CEIL(SAL*0.05) AS RESULT FROM EMP;
SQL> SELECT ENAME,FLOOR(SAL*0.05) AS RESULT FROM EMP;

POWER():
========
- it return the power of the given expression.
syntax:
======
power(m,n)

EX:
SQL> SELECT POWER(2,3) FROM DUAL;----------> 8
SQL> SELECT ENAME,POWER(SAL,2) FROM EMP;

MOD():
======
- it return the remainder value of the expression.
syntax:
=======
mod(m,n)

Ex:
SQL> SELECT MOD(10,2) FROM DUAL;----------> 0
SQL> SELECT * FROM EMP WHERE MOD(EMPNO,2)=0; (for even empno's)
SQL> SELECT * FROM EMP WHERE MOD(EMPNO,2)=1; (for odd empno's)

ROUND():
========

- it return the nearest value of the given expression based on 0.5 value.
    - > if an expression is having less than 0.5 then it return------> 0
    - > if an expression is having greater than or is equals to 0.5 then it return-----> 1

syntax:
=======

round(expression[,decimal places])

EX:
SQL> SELECT ROUND(56.2) AS RESULT FROM DUAL;
Sol:
====

56.2 ===> 0.2 < 0.5 ===> 0
+0
====
56
====

SQL> SELECT ROUND(56.5) AS RESULT FROM DUAL;
Sol:
====

56.5 ===> 0.5 = 0.5 ===> 1
+1
====
57
====

SQL> SELECT ROUND(56.8) AS RESULT FROM DUAL;
Sol:
====

56.8 ===> 0.8 > 0.5 ===> 1
+1
====
57
====

Ex:
SQL> SELECT ROUND(56.870,2) AS RESULT FROM DUAL;

Sol:

===
56.87====> 0.0 < 0.5 ===> 0
+0
=====
56.87
=====
Ex:
SQL> SELECT ROUND(56.875,2) AS RESULT FROM DUAL;

Sol:
===
56.87====> 0.5 = 0.5 ===> 1
+1
=====
56.88
=====
Ex:
SQL> SELECT ROUND(56.877,2) AS RESULT FROM DUAL;

Sol:
===
56.87====> 0.7 > 0.5 ===> 1
+1
=====
56.88
=====

TRUNC():
========
- it return an exact value from the given expression.
- it does not depends on 0.5 value.
syntax:
======
trunc(expression[,decimal places])

EX:
SQL> SELECT TRUNC(56.2) AS RESULT FROM DUAL;------> 56
SQL> SELECT TRUNC(56.5) AS RESULT FROM DUAL;------> 56
SQL> SELECT TRUNC(56.8) AS RESULT FROM DUAL;-------> 56

EX:
SQL> SELECT TRUNC(56.82,1) AS RESULT FROM DUAL;--------> 56.8
SQL> SELECT TRUNC(56.85,1) AS RESULT FROM DUAL;--------> 56.8

SQL> SELECT TRUNC(56.89,1) AS RESULT FROM DUAL;--------> 56.8

character / string functions:
==========
LENGTH():
==========
- it return the length of the given string.

syntax:
======
length(string)

Ex:
SQL> SELECT LENGTH('HELLO') FROM DUAL;----------> 5
SQL> SELECT LENGTH('WEL COME') FROM DUAL;------> 8

EX:
SQL> SELECT ENAME ,LENGTH(ENAME) AS RESULT FROM EMP;
SQL> SELECT * FROM EMP WHERE LENGTH(ENAME)<5;
SQL> SELECT * FROM EMP WHERE LENGTH(ENAME)=5;
SQL> SELECT * FROM EMP WHERE LENGTH(ENAME)>5;

LOWER():
========
- to convert upper case characters into lower case characters.

syntax:
======
lower(string)

EX:
SQL> SELECT LOWER('HELLO') FROM DUAL;---------->hello

EX:
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 characters into upper case characters.

syntax:
======
upper(string)

EX:

SQL> SELECT UPPER('hello') FROM DUAL;---------->HELLO

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

INITCAP():
=========
- to make an initial character is capital from the given string.

syntax:
=======
initcap(string)

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

SQL> SELECT INITCAP('bhuvin kumar') FROM DUAL;
Bhuvin Kumar

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

CONCAT():
=========
- to add two string expressions.

syntax:
======
concat(string1,string2)

EX:
SQL> SELECT CONCAT('HAI','BYE') FROM DUAL;
HAIBYE

EX:
SQL> SELECT ENAME,CONCAT('Mr./Mis.',ENAME) FROM EMP;
SQL> UPDATE EMP SET ENAME=CONCAT('Mr.',ENAME);

LTRIM():
========
- it remove unwanted characters from the given string on left side.

syntax:
=======

ltrim(string,'<trimming character>')

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

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

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

RTRIM():
========
- it remove unwanted characters from the given string on right side.
syntax:
=======
rtrim(string,'<trimming character>')

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

SQL> SELECT RTRIM('HELLOXXXX','X') FROM DUAL;
HELLO

TRIM():
======
- it remove unwanted characters from both sides of the given string.
syntax:
=======
trim('trimming character' from STRING)

EX:
SQL> SELECT TRIM('X' FROM 'XXXXSMITHXXXX') FROM DUAL;
SMITH

EX:
SQL> SELECT TRIM('XY' FROM 'XYSMITHXY') FROM DUAL;
ERROR at line 1:
ORA-30001: trim set should have only one character

REPLACE():

==========

- to replace string to string / string to character / character to string.

syntax:

======

replace(string,'<old characters>','<new characters>')

EX:

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

BLACK AND BLUE

SQL> SELECT REPLACE('HELLO','ELL','D') FROM DUAL;

HDO

SQL> SELECT REPLACE('HELLO','ELLO','XYZ') FROM DUAL;

HXYZ

TRANSLATE():

============

- to translate each character by character.

syntax:

======

translate(string,'<old characters>','<new characters>')

EX:

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

HXYYZ

Here,

E = X , L = Y , O = Z

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

HABB

Here,

E = A , L = B

LPAD():

======

- filling the specified character on the left side of the given string if string length is less than to user defined length.

syntax:

======

lpad(string,<user defined length>,'<specified character>')

EX:
SQL> SELECT LPAD('HELLO',1) FROM DUAL;
H

SQL> SELECT LPAD('HELLO',10) FROM DUAL;
_ _ _ _ _HELLO

SQL> SELECT LPAD('HELLO',10,'A') FROM DUAL;
AAAAAHELLO

RPAD():
======
- filling the specified character on the right side of the given string if string length is less than to user defined length.

syntax:
======
rpad(string,<user defined length>,'<specified character>')

EX:
SQL> SELECT RPAD('HELLO',10,'A') FROM DUAL;
HELLOAAAAA

SUBSTR():
=========
- it return the required substring from the given string expression.

syntax:
=======
substr(string,<starting position of character>,<no.of characters>)

Expression:
========== -7 -6 -5 -4 -3 -2 -1
W E L C O M E
1 2 3 4 5 6 7

EX:
SQL> SELECT SUBSTR('WELCOME',1,1) FROM DUAL;
W

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

SQL> SELECT SUBSTR('WELCOME',6,3) FROM DUAL;

ME

SQL> SELECT SUBSTR('WELCOME',-5,1) FROM DUAL;
L

SQL> SELECT SUBSTR('WELCOME',-7,5) FROM DUAL;
WELCO

SQL> SELECT SUBSTR('WELCOME',-4,-2) FROM DUAL;
Here,
- no.of characters should not be (-ve) sign.

INSTR():
=======
- to find out the occurrence position of the specified character.

syntax:
======
instr(string,'<specified character>',<starting position of character>,<occurrence position of specified character>)

Expression:
========== -13-12-11-10-9-8 -7 -6 -5 -4 -3- 2- 1
' W E L C O M E H E L L O '
1 2 3 4 5 6 7 8 9 10 11 12 13-------fixed positions

NOTE:
======
- we can search our required character from left to right (or) right to left in the string expression but the position of characters are fixed position.

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

INSTR('WELCOMEHELLO','L')

3

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

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

3

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

INSTR('WELCOMEHELLO','L',1,2)
-----------------------------
                           11

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

INSTR('WELCOMEHELLO','L',1,3)
-----------------------------
                           12

SQL> SELECT INSTR('WELCOME HELLO','L',8,3) FROM DUAL;

INSTR('WELCOMEHELLO','L',8,3)
-----------------------------
                            0

SQL> SELECT INSTR('WELCOME HELLO','L',8,1) FROM DUAL;

INSTR('WELCOMEHELLO','L',8,1)
-----------------------------
                           11

SQL> SELECT INSTR('WELCOME HELLO','L',8,2) FROM DUAL;

INSTR('WELCOMEHELLO','L',8,2)
-----------------------------
                           12

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

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

SQL> SELECT INSTR('WELCOME HELLO','L',11,2) FROM DUAL;

INSTR('WELCOMEHELLO','L',11,2)
------------------------------
                            12
```

```
SQL> SELECT INSTR('WELCOME HELLO','E',11,2) FROM DUAL;

INSTR('WELCOMEHELLO','E',11,2)
------------------------------
                             0

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

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

SQL> SELECT INSTR('WELCOME HELLO','E',2,3) FROM DUAL;

INSTR('WELCOMEHELLO','E',2,3)
-----------------------------
                            10

SQL> SELECT INSTR('WELCOME HELLO','E',-8,3) FROM DUAL;

INSTR('WELCOMEHELLO','E',-8,3)
------------------------------
                             0

SQL> SELECT INSTR('WELCOME HELLO','E',-8,1) FROM DUAL;

INSTR('WELCOMEHELLO','E',-8,1)
------------------------------
                             2

SQL> SELECT INSTR('WELCOME HELLO','E',-8,2) FROM DUAL;

INSTR('WELCOMEHELLO','E',-8,2)
------------------------------
                             0

SQL> SELECT INSTR('WELCOME HELLO',' ',-5,2) FROM DUAL;

INSTR('WELCOMEHELLO','',-5,2)
-----------------------------
                             0

SQL> SELECT INSTR('WELCOME HELLO',' ',-5,1) FROM DUAL;
```

INSTR('WELCOMEHELLO','',-5,1)

8

Date functions:
==========
SYSDATE:
========
- it return the current date information of the system.

syntax:
======
sysdate()
EX:
SQL> SELECT SYSDATE FROM DUAL;
01-NOV-24

SQL> SELECT SYSDATE+10 FROM DUAL;
11-NOV-24

SQL> SELECT SYSDATE-10 FROM DUAL;
22-OCT-24

ADD_MONTHS():
==========
- to add / subtract no.of months from / to the given date expression.
syntax:
=======
add_months(date,<no.of months>)

EX:
SQL> SELECT ADD_MONTHS(SYSDATE,6) FROM DUAL;
01-MAY-25

SQL> SELECT ADD_MONTHS(SYSDATE,-6) FROM DUAL;
01-MAY-24

LAST_DAY():
==========
- it return the last day of the month in the given date expression.
syntax:
=======

last_day(date)

EX:

```
SQL> SELECT LAST_DAY(SYSDATE) FROM DUAL;
30-NOV-24

SQL> SELECT LAST_DAY('04-SEP-22') FROM DUAL;
30-SEP-22
```

MONTHS_BETWEEN():

==========

- it return no.of months in between the given two dates.

syntax:

=======

months_between(date1,date2)

NOTE:

=====

- date1 is must be greater than to date2 expression otherwise it return (-ve) sign value.

EX:

```
SQL> SELECT MONTHS_BETWEEN('05-MAY-22','05-MAY-23') FROM DUAL;------> -12
SQL> SELECT MONTHS_BETWEEN('05-MAY-23','05-MAY-22') FROM DUAL;------>  12
```

ANALYTICAL FUNCTIONS:

==========

- to assign rank numbers to each row wise (or) to each group of rows wise.
- oracle supports the following two types of analytical functions are,
  - i) Rank()
  - ii) Dense_Rank()
- these analytical functions are also called as "Ranking Functions".

syntax:

=======

analytical function name()over([partition by <column name>] order by <column name> <asc/desc>)

Here,

- Partition by ------- optional clause
- Order by ---------- mandatory clause

EX:

===

ENAME	SALARY	RANK()	DENSE_RANK()
======	======	======	==========

A 85000 1 1
B 72000 2 2
C 72000 2 2
D 68000 4 3
E 55000 5 4
F 55000 5 4
G 48000 7 5
H 32000 8 6

EX ON WITHOUT PARTITION BY CLAUSE:
==========
SQL> SELECT ENAME,SAL,RANK()OVER(ORDER BY SAL DESC) AS RANKS FROM EMP;
SQL> SELECT ENAME,SAL,DENSE_RANK()OVER(ORDER BY SAL DESC) AS RANKS FROM EMP;

EX ON WITH PARTITION BY CLAUSE:
==========
SQL> SELECT ENAME,DEPTNO,SAL,RANK()OVER(PARTITION BY DEPTNO ORDER BY SAL DESC) AS RANKS FROM EMP;
SQL> SELECT ENAME,DEPTNO,SAL,DENSE_RANK()OVER(PARTITION BY DEPTNO ORDER BY SAL DESC) AS RANKS FROM EMP;

CONVERSION FUNCTIONS:
==========
I) TO_CHAR()
II) TO_DATE()

I) TO_CHAR():
==========
- to convert date type to character(string) type and also display date in different formats.

syntax:
==========
to_char(sysdate,'<intervals>')

Year Formats:

YYYY - Year in four digits format
YY - Last two digits from year
YEAR - Twenty Twenty-Four
CC - Centuary 21
AD / BC - AD Year / BC Year

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

OUTPUT:

2024 24 TWENTY TWENTY-FOUR 21 AD

Month Format:

MM - Month In Number Format
MON - First Three Char's From Month Spelling
MONTH - Full Name Of Month

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

OUTPUT:
---
11 NOV NOVEMBER

Day Formats:

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 DY DAY') FROM DUAL;

OUTPUT:
---
309 04 2 MON MONDAY

Quater Format:

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;

OUTPUT:

4

Week Format:

WW - Week Of The Year
W - Week Of Month

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

OUTPUT:
---
45 1

Time Format:

HH - Hour Part In 12hrs Format
HH24 - Hour Part In 24hrs Fromat
MI - Minute Part
SS - Seconds Part
AM / PM - Am Time (Or) Pm Time

EX:
SQL> SELECT TO_CHAR(SYSDATE,'HH HH24 MI SS PM') FROM DUAL;

OUTPUT:
---
10 10 22 58 AM

EX:
waq to display list of employees who are joined in 1981 by using to_char()?
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'YYYY')='1981';
(OR)
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'YY')='81';
(OR)
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'YEAR')='NINETEEN
EIGHTY-ONE';

EX:
waq to display list of employees who are joined in 1980,1982,1983 by using to_char()?
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'YYYY')IN('1980','1982','1983');

EX:
waq to display list of employees who are joined in the month of DECEMBER?
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'MM')='12';

EX:
waq to display list of employees who are joined in the month of DECEMBER in 1982?
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'MM')='12' AND
TO_CHAR(HIREDATE,'YYYY')='1982';
(OR)
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'MMYYYY')='121982';

EX:
waq to display list of employees who are joined on monday?
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'DY')='MON';

EX:
waq to display list of employees who are joined on weekends?
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'DY') IN('SAT','SUN');
(OR)
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'DY')='SAT' OR
TO_CHAR(HIREDATE,'DY')='SUN';

EX:
waq to display the list of employees who are joned in the 2nd week of JUNE?
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'WMM')='206';

EX:
waq to display the list of employees who are joned 4th quater of 1981?
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'QYYYY')='41981';

II) TO_DATE():
==========
- to convert string type to oracle default date type.

syntax:
=======
to_date(string)

EX:
SQL> SELECT TO_DATE('23/AUGUST/2022') FROM DUAL;
23-AUG-22

SQL> SELECT TO_DATE('23/AUGUST/2022')+5 FROM DUAL;
28-AUG-22

SQL> SELECT TO_DATE('23/AUGUST/2022')-5 FROM DUAL;
18-AUG-22

ii) MULTIPLE ROW FUNCTIONS:
==========
- these functions are also called as "grouping functrions" / "aggregative functions" in database.
- these functions are working on group of values of a column.

syntax:
=======
<aggregative function name>(<column name>)

SUM():
======
- it return total value.

EX:
SQL> SELECT SUM(SAL) FROM EMP;
SQL> SELECT SUM(SAL) FROM EMP WHERE JOB='MANAGER';

AVG():
======
- it return the average value.

EX:
SQL> SELECT AVG(SAL) FROM EMP;
SQL> SELECT AVG(SAL) FROM EMP WHERE DEPTNO=20;

MIN():
======

- it return minimum value.

EX:

SQL> SELECT MIN(SAL) FROM EMP;

SQL> SELECT MIN(HIREDATE) FROM EMP;

SQL> SELECT MIN(SAL) FROM EMP WHERE DEPTNO=30;

MAX():

======

- it return maximum value.

EX:

SQL> SELECT MAX(SAL) FROM EMP;

COUNT():

=========

- it again three formats.

i) count(*):

==========

- counting all rows including duplicates and nulls in a table.

EX:

SQL> SELECT COUNT(*) FROM EMP;----------> 14

ii) count(column name):

======================

- counting all values including duplicate values but not nulls.

EX:

SQL> SELECT COUNT(MGR) FROM EMP; ---------> 13

iii) count(distinct <column name>):

==============================

- counting unique values only.(no duplicates & no nulls)

EX:

SQL> SELECT COUNT(DISTINCT MGR) FROM EMP;-------> 6

==========

CLAUSES:

=========

- it is a statement which is used to add to sql query for providing some additional facilities like "filtering rows,sorting values and grouping similar data" based on columns automatically.

- oracle supports the following clauses are,

    - WHERE

- ORDER BY
- GROUP BY
- HAVING

syntax:
=======
<select query> + <clause statement>;

WHERE:
=======
- filtering rows before grouping the data in a table.
- it can be used in "SELECT,UPDATE,DELETE" commands only.

syntax:
======
where <filtering condition>;

EX:
SQL> SELECT * FROM EMP WHERE DEPTNO=20;
SQL> UPDATE EMP SET SAL=25000 WHERE JOB='MANAGER';
SQL> DELETE FROM EMP WHERE SAL=3000;

ORDER BY:
==========
- to arrange a specific column/(s) values either in ascending or descending order.
- by default order by clause will arrange the values in ascending order.if we want to arrange the values in descending order then we must use "DESC" keyword.
- it can be used in "SELECT" command only.

syntax:
======
select * / <list of columns> from <table name> order by <column name1> <asc/desc>, <column name2> <asc/desc>,.....;

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

SQL> SELECT * FROM EMP ORDER BY ENAME;
SQL> SELECT * FROM EMP ORDER BY ENAME DESC;

SQL> SELECT * FROM EMP ORDER BY HIREDATE;
SQL> SELECT * FROM EMP ORDER BY HIREDATE DESC;

EX:
waq to display employees who are working under deptno is 20 and arrange those employees salaries in descending order?
SQL> SELECT * FROM EMP WHERE DEPTNO=20 ORDER BY SAL DESC;

EX:
waq to arrange 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 we can apply on the position of column in the SELECT query.

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

ORDER BY with null clauses:
==========
- there two types of NULL clauses:
  i) NULLS FIRST
  ii) NULLS LAST

i) NULLS FIRST:
==========
- by default order by clause will arrange the nulls in ascending order is:
  First : Values
  Later : Nulls

Ex:
SQL> SELECT * FROM EMP ORDER BY COMM;
- To overcome the above problem in ascending order then we must use "NULLS FIRST" clause.

Solution:
========
SQL> SELECT * FROM EMP ORDER BY COMM NULLS FIRST;

ii) NULLS LAST:
==========
- by default order by clause will arrange the nulls in descending order is:
  First : Nulls

Later : Values

Ex:

SQL> SELECT * FROM EMP ORDER BY COMM DESC;

- To overcome the above problem in descending order then we must use "NULLS LAST" clause.

Solution:

========

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

GROUP BY:

=========

- it is used to make groups based on column / columns.
- when we use "group by" clause we must use "grouping / aggregative" functions to get the result.
- it can be used in "SELECT" query only.

syntax:

=======

select <column name1>,<column name2>,.....,<aggregative function name1>,.....
from <table name> group by <column name1>,<column name2>,.....;

EX:

waq to find out no.of employees working in the organization?

SQL> SELECT COUNT(*) AS TOTAL_NO_OF_EMPLOYEES FROM EMP;

```
TOTAL_NO_OF_EMPLOYEES
-----------------------------------------
                   14
```

EX:

waq to find out no.of employees are working under the job is "SALESMAN"?

SQL> SELECT COUNT(*) AS TOTAL_NO_OF_EMPLOYEES FROM EMP WHERE JOB='SALESMAN';

```
TOTAL_NO_OF_EMPLOYEES
-----------------------------------------
                    4
```

EX:

waq to find out no.of employees are working under each job?

SQL> SELECT JOB,COUNT(*) AS NO_OF_EMPLOYEES FROM EMP GROUP BY JOB;

```
JOB                       NO_OF_EMPLOYEES
---------           ------------------------------
```

CLERK 4
SALESMAN 4
ANALYST 2
MANAGER 3
PRESIDENT 1

EX:
waq to display no.of employees are working under each job along with their deptno wise?
SQL> SELECT DEPTNO,JOB,COUNT(*) AS NO_OF_EMPLOYEES
2 FROM EMP GROUP BY DEPTNO,JOB ORDER BY DEPTNO;

EX:
waq to display sum of salaries of each deptno wise?
SQL> SELECT DEPTNO,SUM(SAL) AS TOTAL_SALARY FROM EMP
2 GROUP BY DEPTNO ORDER BY DEPTNO;

EX:
waq to display average and total salary of each deptno wise?
SQL> SELECT DEPTNO,AVG(SAL) AS AVG_SALARY,
2 SUM(SAL) AS TOTAL_SALARY FROM EMP
3 GROUP BY DEPTNO ORDER BY DEPTNO;

GROUP BY clause with all aggregative functions:
==========
SQL> SELECT DEPTNO,COUNT(*) AS NO_OF_EMPLOYEES,
2 SUM(SAL) AS TOTAL_SALARY,AVG(SAL) AS AVG_SALARY,
3 MAX(SAL) AS MAX_SALARY,MIN(SAL) AS MIN_SALARY
4 FROM EMP GROUP BY DEPTNO ORDER BY DEPTNO;

HAVING:
=======
- filtering rows after grouping the data in a table.
- having clause should use after "group by" clause only.

syntax:
=======
select <column name1>,<column name2>,.....,<aggregative function name1>,.....
from <table name> group by <column name1>,<column name2>,.....having<filtering condition>;

Ex:
waq to display deptno's from emp table in which deptno the no.of employees are working more than 3?
SQL> SELECT DEPTNO,COUNT(*) FROM EMP
2 GROUP BY DEPTNO HAVING COUNT(*)>3 ORDER BY DEPTNO;

EX:
waq to display jobs from emp table if sum of salary of the job is less than 5000?

```
SQL> SELECT JOB,SUM(SAL) FROM EMP
  2  GROUP BY JOB HAVING SUM(SAL)<5000;
```

WHERE vs HAVING:
==========

WHERE	HAVING
1. filtering rows before grouping data in the table.	1. filtering rows after grouping data in the table.
2. WHERE condition will work on each individual row wise.	2. HAVING condition will work on each group of rows wise.
3. it does not supports "aggregative functions".	3. it supports "aggregative functions".
4. it will use before "group by" clause.	4. it will use after "group by" clause.
5. without "group by" clause WHERE clause can be worked.	5. without "group by" clause HAVING clause will not worked.

Using all clauses in a single "SELECT" statement:
==========

syntax:
======

```
select <column name1>,<column name2>,.....,<aggregative function name1>,.....
from <table name> [ where <filtering condition>
                        group by <column name1>,<column name2>,.....
                                having<filtering condition>
                                        order by <column name1> <asc/desc>
                ];
```

EX:

```
SQL> SELECT DEPTNO,COUNT(*) FROM EMP WHERE SAL>1000 GROUP BY DEPTNO
  2  HAVING COUNT(*)>3 ORDER BY DEPTNO;

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

==========
JOINS:
======

- In RDBMS data can be stored in multiple tables.from those multiple tables if we want to retrieve the required data / information then we should use a technique is known as "JOINS".
- Joins are used to retrieving data/information from multiple tables at a time.
- Oracle supports the following types of joins are:
    1) Inner joins
        - Equi join
        - Non-Equi join
        - Self join
    2) Outer joins
        - Left outer join
        - Right outer join
        - Full outer join
    3) Cross join / Cartisean join
    4) Natural join

How to join two tables:
==========
syntax:
=======
select * / <list of columns> from <table name1> <join key> <table name2> on <joining condition>;

Equi join:
========

- when we retrieve data from multiple tables based on an " = " operator join condition is called as "equi join".
- when we use equi join we should maintain atleast one common column in both tables and those common columns datatypes must match / same.
- whenever we want to join multiple tables there is no need to maintain relationship between tables.Here relationship is optional.
- by using equi join we always retrieve matching rows(data) from multiple tables.

syntax for equi join:
==========
on <table name1>.<common column> = <table name2>.<common column>;

Demo_tables:

==========
SQL> SELECT * FROM COURSE;

CID	CNAME	CFEE
1	ORACLE	2500
2	JAVA	5000
3	PYTHON	8000

SQL> SELECT * FROM STUDENT;

STID	SNAME	CID
1021	SMITH	1
1022	ALLEN	1
1023	JONES	2
1024	ADAMS	

EX:
waq to retrieve STUDENTS and the corresponding COURSE details?
SQL> SELECT * FROM STUDENT JOIN COURSE ON CID=CID;
ERROR at line 1:
ORA-00918: column ambiguously defined.

- To overcome the above problem then we should use a table name as an identity to a column like below,

Solution:
========
SQL> SELECT * FROM STUDENT JOIN COURSE ON STUDENT.CID=COURSE.CID;
(OR)
SQL> SELECT STID,SNAME,CNAME,CFEE FROM STUDENT JOIN COURSE ON STUDENT.CID=COURSE.CID;
(OR)
SQL> SELECT * FROM STUDENT S INNER JOIN COURSE C ON S.CID=C.CID;

Rule for JOINS:
=============
- a row in the first table is comparing with all rows of the second table.

EX:
waq to display students and course details from the tables who are joined in ORACLE course?

```
SQL> SELECT SNAME,CNAME FROM STUDENT S INNER JOIN COURSE C
  2  ON S.CID=C.CID WHERE CNAME='ORACLE';
                    (OR)
SQL> SELECT SNAME,CNAME FROM STUDENT S INNER JOIN COURSE C
  2  ON S.CID=C.CID AND CNAME='ORACLE';
```

Ex:
waq to display employees who are working in the location is "CHICAGO"?

```
SQL> SELECT ENAME,LOC FROM EMP E INNER JOIN DEPT D
  2  ON E.DEPTNO=D.DEPTNO AND LOC='CHICAGO';
```

Ex:
waq to display sum of salaries of each department names wise from emp,dept tables?

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

Ex:
waq to display DEPTNO and also sum of salaries of each departement name wise from emp,dept tables?

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

Ex:
waq to display no.of employees working in each department name from emp,dept tables?

```
SQL> SELECT DNAME,COUNT(*) FROM EMP E JOIN DEPT D
        ON E.DEPTNO=D.DEPTNO GROUP BY DNAME;
```

Ex:
waq to display departement names from emp,dept table in which department name the no.of employees are less than to 3?

```
QL> SELECT DNAME,COUNT(*) FROM EMP E JOIN DEPT D
  2          ON E.DEPTNO=D.DEPTNO GROUP BY DNAME HAVING COUNT(*)<3;
```

2) Outer joins:
==========

- there are again three types.

i)Left outer join:
==========

- retrieving matching rows from both tables and unmatching rows from the left side of the table.

Ex:

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

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

ii) Right outer join:
==========
- retrieving matching rows from both tables and unmatching rows from the right side of the table.

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

iii) Full outer join:
==========
- it is a combination of left outer and right outer join.
- retrieving matching and also unmatching rows from both tables at a time.

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

NOTE:
=====
- Generall "equi join" is retrieving only matching rows from the multiple tables.if we want to retrieve matching and unmatching rows from the multiple tables then we must use "OUTER JOINS" techniques.

NON-EQUI JOIN:
==========
- when we are retrieving data from multiple tables based on any operator except an " = " operator.
- in this join we will use the following operators are < , > , <= , >= , != , AND,OR,BETWEEN,.....etc.

DEMO_TABLES:
==========
SQL> SELECT * FROM TEST11;

```
       SNO NAME
---------- ----------
         1 SMITH
         2 ALLEN
```

SQL> SELECT * FROM TEST12;

```
      SNO     SAL
---------- ----------
         1      23000
         3      34000
```

EX:

```
SQL> SELECT * FROM TEST11 T1 JOIN TEST12 T2 ON T1.SNO>T2.SNO;
SQL> SELECT * FROM TEST11 T1 JOIN TEST12 T2 ON T1.SNO>=T2.SNO;
SQL> SELECT * FROM TEST11 T1 JOIN TEST12 T2 ON T1.SNO<T2.SNO;
SQL> SELECT * FROM TEST11 T1 JOIN TEST12 T2 ON T1.SNO<=T2.SNO;
SQL> SELECT * FROM TEST11 T1 JOIN TEST12 T2 ON T1.SNO!=T2.SNO;
```

EX:

waq to display employees whose salary is between low salary and high salary from EMP,SALGRADE tables?

```
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:

==========

- joining two or more than two tables without any condition.
- in cross join mechanism each row of a table will join with each row of another table. for example a table is having (m) no.of rows and another table is having(n) no.of rows then the result is (mXn) rows.

EX:

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

DEMO_TABLES:

==========

```
SQL> SELECT * FROM ITEMS1;

      SNO   INAME          PRICE
---------- ----------          ----------
         1     PIZZA           180
         2     BURGER           85

SQL> SELECT * FROM ITEMS2;

      SNO   INAME          PRICE
---------- ----------          ----------
         1     PEPSI            20
```

2 COCACOLA 25

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

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

- it is a similar to equi join.
- retrieving matching rows only.
- natural join condition is preparing by system by default based on an " = " operator.
- the advantage of natural join is avoiding duplicate columns from the result set.

EX:

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

SELF JOIN:
==========

- joining a table by itself is known as "self join".

(or)

- comparing a table data by itself is known as "self join".
- when we use self join we must create alias names otherwise self join cannot be implemented.
- whenever we are creating alias names on a table internally oracle server will prepare virtual tables on each alias name.
- we can create any no.of alias names on a single table but each alias name should be different.
- self join can be used at two levels:
  - Level-1: comparing a single column values by itself with in the table.
  - Level-2: comparing two different columns values to each other with in the table.

Level-1: comparing a single column values by itself with in the table:
==========

Ex:

waq to display employees who are working in the same location where the employee SMITH is also working?

DEMO_TABLE:
==========

```
SQL> SELECT * FROM TEST;

ENAME      LOC
----------      ----------
```

```
SMITH      HYD
ALLEN      PUNE
WARD       HYD
MILLER     DELHI

SQL> SELECT T1.ENAME,T1.LOC FROM TEST T1 JOIN TEST T2
        ON T1.LOC=T2.LOC AND T2.ENAME='SMITH';

OUTPUT:
=========
ENAME      LOC
----------      ----------
SMITH      HYD
WARD       HYD
```

EX:
waq to display employees whose salary is same as the employee "SCOTT" salary?

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

Level-2: comparing two different columns values to each other with in the table.:
==========

Ex:
waq to display managers and their employees from emp table?

```
SQL> SELECT M.ENAME AS MANAGERS,E.ENAME AS EMPLOYEES
  2  FROM EMP E JOIN EMP M ON M.EMPNO=E.MGR;
```

Ex:
waq to display employees who are working under the manager "BLAKE"?

```
SQL> SELECT M.ENAME AS MANAGERS,E.ENAME AS EMPLOYEES
  2  FROM EMP E JOIN EMP M ON M.EMPNO=E.MGR WHERE M.ENAME='Mr.BLAKE';
```

Ex:
waq to display manager of the employee "BLAKE"?

```
SQL> SELECT M.ENAME AS MANAGERS,E.ENAME AS EMPLOYEES
  2  FROM EMP E JOIN EMP M ON M.EMPNO=E.MGR WHERE E.ENAME='Mr.BLAKE';
```

Ex:
waq to display employees who are joined before their manager?

```
SQL> SELECT E.ENAME AS EMPLOYEE,E.HIREDATE AS E_DOJ,
  2  M.ENAME AS MANAGER,M.HIREDATE AS M_DOJ FROM
  3  EMP E JOIN EMP M ON M.EMPNO=E.MGR AND E.HIREDATE<M.HIREDATE;
```

Ex:
waq to display employees whose salary is more than to their manager salary?

```
SQL> SELECT E.ENAME AS EMPLOYEE,E.SAL AS E_SAL,
  2  M.ENAME AS MANAGER,M.SAL AS M_SAL FROM
  3  EMP E JOIN EMP M ON M.EMPNO=E.MGR AND E.SAL>M.SAL;
```

How to join more than two tables:
==========
syntax:
======
SELECT * / <list of columns> 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>;

EX:
DEMO_TABLE:
==========
SQL> SELECT * FROM REGISTER;

REGNO	REGDATE	CID
1001	15-NOV-24	1
1002	16-NOV-24	2

Joining Three Tables:
==========
SQL> SELECT * FROM STUDENT;
SQL> SELECT * FROM COURSE;
SQL> SELECT * FROM REGISTER;

Example on Equi join:
==========
```
SQL> SELECT STID,SNAME,CNAME,REGDATE,CFEE FROM
  2  STUDENT S JOIN COURSE C ON S.CID=C.CID
  3  JOIN REGISTER R ON C.CID=R.CID;
```

OUTPUT:
========

STID	SNAME	CNAME	REGDATE	CFEE

1021 SMITH ORACLE 15-NOV-24 2500
xxxx xxxxx xxxxxxx xxxxxxxx xxxx

==========

CONSTRAINTS:

==========

- constraints are used to enforce unwanted / invalid data from a table.
- oracle supports the following six types of constraints those are,
  - UNIQUE
  - NOT NULL
  - CHECK
  - PRIMARY KEY
  - FOREIGN KEY
  - DEFAULT
- constraints are applied on a table in two ways.
  - i) column level
  - ii) table level

i) column level:

==========

- constraint can be defined on each individual column wise.

syntax:

======

create table <table name>(<column name1> <datatype>[size] <constraint type>,
<column name2> <datatype>[size] <constraint type>,.....);

ii) table level:

==========

- constraint can be defined after all columns definitions i.e the end of the table.

syntax:

======

create table <table name>(<column name1> <datatype>[size],<column name2>
<datatype>[size],
.....,<constraint type>(<column name1>,<column name2>,.....);

UNIQUE:

=======

- to restricted duplicate values but allowed nulls into a column.

EX:

column level:

==========

SQL> CREATE TABLE TEST1(SNO NUMBER(2)UNIQUE,NAME VARCHAR2(10)UNIQUE);

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

table level:
==========
SQL> CREATE TABLE TEST2(SNO NUMBER(2),NAME
VARCHAR2(10),UNIQUE(SNO,NAME));

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

NOT NULL:
=========
- to restricted nulls but allowed duplicate values into a column.
- it can be defined at column level only.

EX:
column level:
============
SQL> CREATE TABLE TEST3(SNO NUMBER(3) NOT NULL,NAME 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

CHECK:
=======
- to check the values with user defined condition on a column before accepting
the values.

EX:
column level:

==========
```
SQL> CREATE TABLE TEST4
  2  (
  3  REGNO NUMBER(5)UNIQUE NOT NULL,
  4  CNAME VARCHAR2(10) NOT NULL,
  5  ENTRY_FEE NUMBER(6,2)NOT NULL CHECK(ENTRY_FEE=500),
  6  AGE NUMBER(2)NOT NULL CHECK(AGE BETWEEN 18 AND 30),
  7  LOC VARCHAR2(10)NOT NULL CHECK(LOC IN('HYD','MUMBAI','DELHI'))
  8  );
```

TESTING:
=========
```
SQL> INSERT INTO TEST4 VALUES(10001,'SMITH',450,17,'HYDERABAD');----NOT
ALLOWED
SQL> INSERT INTO TEST4 VALUES(10001,'SMITH',500,30,'HYD');--------ALLOWED
```

table level:
=========
```
SQL> CREATE TABLE TEST5(ENAME VARCHAR2(10),SAL
NUMBER(8,2),CHECK(ENAME=LOWER(ENAME) AND SAL>15000));
```

TESTING:
========
```
SQL> INSERT INTO TEST5 VALUES('ALLEN',23000);------NOT ALLOWED
SQL> INSERT INTO TEST5 VALUES('allen',23000);-------ALLOWED
```

PRIMARY KEY:
=============

- it is a combination of unique and not null constraint.
- by using primary key we can restricted duplicates and nulls from a column.
- a table is having only one primary key.

EX:
column level:
============
```
SQL> CREATE TABLE TEST6(STID NUMBER(4) PRIMARY KEY,
       SNAME VARCHAR2(10) NOT NULL,REGNO NUMBER(5)PRIMARY KEY);
ERROR at line 2:
ORA-02260: table can have only one primary key
```

Solution:
========
```
SQL> CREATE TABLE TEST6(STID NUMBER(4) PRIMARY KEY,
```

2 SNAME VARCHAR2(10) NOT NULL,REGNO NUMBER(5)UNIQUE NOT NULL);

TESTING:
=========
SQL> INSERT INTO TEST6 VALUES(1021,'SMITH',10001);------> ALLOWED
SQL> INSERT INTO TEST6 VALUES(1021,'JONES',10001);------> NOT ALLOWED
SQL> INSERT INTO TEST6 VALUES(NULL,'JONES',NULL);------> NOT ALLOWED
SQL> INSERT INTO TEST6 VALUES(1022,'JONES',10002);------> ALLOWED

COMPOSITE PRIMARY KEY: (table level)
======================
- when we apply a primary key constraint on combination of multiple columns are called as "composite primary key".
- in composite primary key individual columns are accepting duplicate values but combination columns are not accepting duplicate values.

EX:
SQL> CREATE TABLE TEST7(BCODE NUMBER(4),BNAME VARCHAR2(10),LOC VARCHAR2(10),PRIMARY KEY(BCODE,BNAME));

TESTING:
========
SQL> INSERT INTO TEST7 VALUES(1021,'SBI','MADHAPUR');-----ALLOWED
SQL> INSERT INTO TEST7 VALUES(1021,'SBI','SRNAGAR');-----NOT ALLOWED
SQL> INSERT INTO TEST7 VALUES(NULL,NULL,'SRNAGAR');-----NOT ALLOWED
SQL> INSERT INTO TEST7 VALUES(1022,'SBI','SRNAGAR');-----ALLOWED

FOREIGN KEY:
============
- to make relationship between tables for taking an identity(i.e reference) from one table to another table.

Basic Rules:
===========
1. to maintain atleast one commonness column in both tables.
2. commonness column datatype must be same / match.
3. in a relationship one table should contain primary key and another table should contain foreign key and these two constraints should apply on commonness column only.
4. a primary key table is called as "parent table" and a foreign key table is called as "child table".
5. foreign key column is allowed the values which was found in primary key column.
6. by default a foreign key column is allowed duplicates and nulls.

syntax:

=======
<common column of child table> <datatype>[size] references <parent table>(common column of parent table)

EX:
SQL> CREATE TABLE DEPT1(DNO NUMBER(2) PRIMARY KEY,DNAME VARCHAR2(10));---PARENT

SQL> INSERT INTO DEPT1 VALUES(1,'ORACLE');
SQL> INSERT INTO DEPT1 VALUES(2,'JAVA');
SQL> COMMIT;

SQL> CREATE TABLE EMP1(EID NUMBER(4) PRIMARY KEY,
  2 ENAME VARCHAR2(10),DNO NUMBER(2) REFERENCES DEPT1(DNO));------CHILD

SQL> INSERT INTO EMP1 VALUES(1021,'SMITH',1);
SQL> INSERT INTO EMP1 VALUES(1022,'JONES',1);
SQL> INSERT INTO EMP1 VALUES(1023,'ADAMS',2);
SQL> INSERT INTO EMP1 VALUES(1024,'MILLER',NULL);
SQL> COMMIT;

NOTE:
======
- Once we established relationship between tables there are two rules are come into picture.

Rule-1: Insertion:
===============
- we cannot insert values into a child table those values are not existing in primary key column of parent table.
==========
i.e no parent = no child
==========
EX:
SQL> INSERT INTO EMP1 VALUES(1025,'SCOTT',3);
ERROR at line 1:
ORA-02291: integrity constraint (MYDB9AM.SYS_C009399) violated - parent key not found

Rule-2: Deletion:
==============
- we cannot delete a row from parent table that row is having the corresponding child rows in child table without addressing to child.

EX:
SQL> DELETE FROM DEPT1 WHERE DNO=1;
ERROR at line 1:
ORA-02292: integrity constraint (MYDB9AM.SYS_C009399) violated - child record found

How to address to child table:
==========
- to address to child table there are two cascade rules in relationship.
i) ON DELETE CASCADE
ii) ON DELETE SET NULL

i) ON DELETE CASCADE:
==========
- when we delete a row from a parent table then the corresponding child rows are deleted from child table automatically.

EX:
SQL> CREATE TABLE DEPT2(DNO NUMBER(2) PRIMARY KEY,DNAME VARCHAR2(10));---PARENT

SQL> INSERT INTO DEPT2 VALUES(1,'ORACLE');
SQL> INSERT INTO DEPT2 VALUES(2,'JAVA');
SQL> COMMIT;

SQL> CREATE TABLE EMP2(EID NUMBER(4) PRIMARY KEY,
ENAME VARCHAR2(10),DNO NUMBER(2) REFERENCES DEPT2(DNO) ON DELETE CASCADE);------CHILD

SQL> INSERT INTO EMP2 VALUES(1021,'SMITH',1);
SQL> INSERT INTO EMP2 VALUES(1022,'JONES',2);
SQL> COMMIT;

TESTING:
==========
SQL> DELETE FROM DEPT2 WHERE DNO=1;----------> DELETED
SQL> SELECT * FROM DEPT2;
SQL> SELECT * FROM EMP2;

ii) ON DELETE SET NULL:
==========
- when we delete a row from parent table then the corresponding child rows of a foreign key column values are converting into NULL in child table automatically.

EX:
SQL> CREATE TABLE DEPT3(DNO NUMBER(2) PRIMARY KEY,DNAME VARCHAR2(10));---PARENT

SQL> INSERT INTO DEPT3 VALUES(1,'ORACLE');
SQL> INSERT INTO DEPT3 VALUES(2,'JAVA');
SQL> COMMIT;

SQL> CREATE TABLE EMP3(EID NUMBER(4) PRIMARY KEY,
ENAME VARCHAR2(10),DNO NUMBER(2) REFERENCES DEPT3(DNO) ON DELETE SET NULL);------CHILD

SQL> INSERT INTO EMP3 VALUES(1021,'SMITH',1);
SQL> INSERT INTO EMP3 VALUES(1022,'JONES',2);
SQL> COMMIT;

TESTING:
=========
SQL> DELETE FROM DEPT3 WHERE DNO=1;----------> DELETED
SQL> SELECT * FROM DEPT3;
SQL> SELECT * FROM EMP3;
==========

DATADICTIONARY:
==========

- when we are installing oracle s/w internally system is creating some pre-defined tables for storing the information about database objects such as Tables,Constraints,Views, Sequences,Indexes,Procedures,Functions,Triggers,.....etc.
- datadictionaries are not allowed DML operations but allowed "SELECT" operation only.so that datadictionaries are also called as "READ ONLY TABLES" in oracle.
- if we want to view all datadictionaries in oracle database then we follow the following syntax is:

SQL> SELECT * FROM DICT; (Dictionary is a Main table)

SYSTEM DEFINED CONSTRAINT NAMES:
==========

- when we apply constraints on columns internally system is creating a unique identification for each constraint for identifying a constraint.

EX:
SQL> CREATE TABLE TEST8(SNO NUMBER(2) PRIMARY KEY,NAME VARCHAR2(10) UNIQUE);

NOTE:
======
- If we want to view constraint name along with column name of a specific table in oracle database then we should use a datadictionary is "USER_CONS_COLUMNS".

Ex:
```
SQL> DESC USER_CONS_COLUMNS;
SQL> SELECT COLUMN_NAME,CONSTRAINT_NAME FROM USER_CONS_COLUMNS
     WHERE TABLE_NAME='TEST8';

COLUMN_NAME                    CONSTRAINT_NAME
-------------------------------- ------------------------------
SNO                            SYS_C009415
NAME                           SYS_C009416
```

USER DEFINED CONSTRAINT NAME:
==============================
- inplace of system defined constraint name we can also create user defined constraint name(ID) for identifying a constraint.

syntax:
=======
<column name> <datatype>[size] constraint <user defined constraint name> <constraint type>

EX:
```
SQL> CREATE TABLE TEST9(SNO NUMBER(2) CONSTRAINT SNO_PK PRIMARY KEY,
     NAME VARCHAR2(10)CONSTRAINT NAME_UQ UNIQUE);

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

COLUMN_NAME                    CONSTRAINT_NAME
-------------------------------- ------------------------------
SNO                            SNO_PK
NAME                           NAME_UQ
```

How to add constraints to an existing table:
====================================
syntax:
=======
ALTER TABLE <TN> ADD CONSTRAINT <CONSTRAINT NAME> <CONSTRAINT TYPE>(<COLUMN NAME>);

EX:
SQL> CREATE TABLE PARENT(STID NUMBER(4),SNAME VARCHAR2(10),SFEE NUMBER(6,2));

i) Adding a primary key:
==========
SQL> ALTER TABLE PARENT ADD CONSTRAINT STID_PK PRIMARY KEY(STID);

ii) Adding Unique,Check constraint:
==========
SQL> ALTER TABLE PARENT ADD CONSTRAINT SNAME_UQ UNIQUE(SNAME);
SQL> ALTER TABLE PARENT ADD CONSTRAINT SFEE_CHK CHECK(SFEE>=5000);

NOTE:
=====
- if we want to view check constraint conditional value of a specific column in the specific table in oracle then we use a datadictionary is "USER_CONSTRAINTS".

EX:
SQL> DESC USER_CONSTRAINTS;
SQL> SELECT CONSTRAINT_NAME,SEARCH_CONDITION FROM
  2 USER_CONSTRAINTS WHERE TABLE_NAME='PARENT';

CONSTRAINT_NAME SEARCH_CONDITION
------------------------------ ----------
SFEE_CHK SFEE>=5000

iii) Adding a Foregin key references:
==========
syntax:
=======
ALTER TABLE <TN> ADD CONSTRAINT <CONSTRAINT 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(CNAME VARCHAR2(10),STID NUMBER(4));

SQL> ALTER TABLE CHILD ADD CONSTRAINT STID_FK
  2 FOREIGN KEY(STID)REFERENCES PARENT(STID)
  3 ON DELETE CASCADE;

How to apply NOT NULL constraint:
==========
syntax:
======
ALTER TABLE <TN> MODIFY <COLUMN NAME> CONSTRAINT <CONSTRAINT NAME>
NOT NULL;

EX:
SQL> ALTER TABLE PARENT MODIFY SNAME CONSTRAINT SNAME_NN NOT NULL;

How to drop constraint from an existing table:
==========
syntax:
=======
ALTER TABLE <TN> DROP CONSTRAINT <CONSTRAINT NAME>;

i) Dropping a Primary key:
==========
method-1: with realtionship:
==========
SQL> ALTER TABLE PARENT DROP CONSTRAINT STID_PK CASCADE;

method-2: without relationship:
==========
SQL> ALTER TABLE PARENT DROP CONSTRAINT STID_PK;

ii) Dropping Unique,Check,Not null constraint:
==========
SQL> ALTER TABLE PARENT DROP CONSTRAINT SNAME_UQ;
SQL> ALTER TABLE PARENT DROP CONSTRAINT SNAME_NN;
SQL> ALTER TABLE PARENT DROP CONSTRAINT SFEE_CHK;

How to change a constraint name:
==========
syntax:
=======
ALTER TABLE <TN> RENAME CONSTRAINT <OLD CONSTRAINT NAME> TO <NEW
CONSTRAINT NAME>;

EX:
SQL> CREATE TABLE TEST10(REGNO NUMBER(5) PRIMARY KEY);

```
SQL> SELECT COLUMN_NAME,CONSTRAINT_NAME
       FROM USER_CONS_COLUMNS WHERE TABLE_NAME='TEST10';

COLUMN_NAME                    CONSTRAINT_NAME
------------------------------ ------------------------------------------------------
REGNO                          SYS_C009432

SQL> ALTER TABLE TEST10 RENAME CONSTRAINT SYS_C009432 TO REGNO_PK;

COLUMN_NAME                    CONSTRAINT_NAME
------------------------------ ------------------------------------------------------
REGNO                          REGNO_PK
```

DEFAULT constraint:
==========

- to assign a user defined default value to a column.

EX:

```
SQL> CREATE TABLE TEST13(ENAME VARCHAR2(10),SAL NUMBER(8,2) DEFAULT
15000);
```

TESTING:

```
SQL> INSERT INTO TEST13(ENAME,SAL)VALUES('A',25000);
SQL> INSERT INTO TEST13(ENAME)VALUES('B');
```

How to add a default value to an existing table column:
==========

syntax:
=======

```
ALTER TABLE <TN> MODIFY <COLUMN NAME> DEFAULT <value>;
```

EX:

```
SQL> CREATE TABLE TEST14(ENAME VARCHAR2(10),LOC VARCHAR2(10));
SQL> ALTER TABLE TEST14 MODIFY LOC DEFAULT 'HYD';
```

TESTING:

```
SQL> INSERT INTO TEST14(ENAME)VALUES('SMITH');
SQL> SELECT * FROM TEST14;
```

NOTE:
======

- If we want to view the default constraint value of a column in the table in oracle database then use a datadictionary is "USER_TAB_COLUMNS".

EX:
SQL> DESC USER_TAB_COLUMNS;
SQL> SELECT COLUMN_NAME,DATA_DEFAULT FROM
  2 USER_TAB_COLUMNS WHERE TABLE_NAME='TEST14';

COLUMN_NAME DATA_DEFAULT
----------- ----------
LOC 'HYD'

How to remove a default constraint values from a column:
==========
syntax:
=======
ALTER TABLE <TN> MODIFY <COLUMN NAME> DEFAULT null;

EX:
SQL> ALTER TABLE TEST14 MODIFY LOC DEFAULT NULL;

COLUMN_NAME DATA_DEFAULT
----------- ----------
LOC NULL
==========
========
TRANSACTION CONTROL LANGUAGE(TCL):
==========
What is Transaction:
==========
- to perform some operation over database.
- to control these transactions on database then we must use TCL commands.
i) commit
ii) rollback
iii) savepoint

i) commit:
==========
- to make a transaction is permanent.
- there are two types of commit transactions those are,
i) Implicit commit:
==========
- these transactions are committed by system by default.
EX: DDL commands

i) Explicit commit:
==========
- these transactions are committed by user.
EX: DML commands

syntax:
=======
COMMIT;

EX:
```
SQL> CREATE TABLE BRANCH(BCODE NUMBER(4),BNAME VARCHAR2(10),LOC
VARCHAR2(10));

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

SQL> UPDATE BRANCH SET LOC='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 LOC='PUNE' WHERE BCODE=1021;
SQL> DELETE FROM BRANCH WHERE BCODE=1021;
SQL> COMMIT;
```

ii) rollback:
==========
- to cancel a transaction.
- once a transaction is committed then we cannot rollback.

syntax:
=======
rollback;

EX:
```
SQL> INSERT INTO BRANCH VALUES(1021,'SBI','HYD');
SQL> ROLLBACK;

SQL> UPDATE BRANCH SET LOC='PUNE' WHERE BCODE=1021;
SQL> ROLLBACK;
```

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

iii) savepoint:
==========
- when we created a savepointer internally system is allocating a special memory for storing the required row / rows which we want to cancel(i.e rollback) in the future.

syntax to create a savepoint:
==========
savepoint <pointer name>;

syntax to rollback a savepoint:
==========
rollback to <pointer name>;

EX:
SQL> DELETE FROM BRANCH WHERE BCODE=1021;
SQL> DELETE FROM BRANCH WHERE BCODE=1025;

SQL> SAVEPOINT P1;
SQL> DELETE FROM BRANCH WHERE BCODE=1023;

TESTING:
========
CASE-1:
========
SQL> ROLLBACK TO P1; (rollback 1023 row only)

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

EX:
SQL> DELETE FROM BRANCH WHERE BCODE=1021;

SQL> SAVEPOINT P1;

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

TESTING:
========
CASE-1:
========
SQL> ROLLBACK TO P1; (rollback 1023,1025 rows only)

CASE-2:
=======
SQL> COMMIT / ROLLBACK;
==========
========
SUBQUERY:
==========
- a query inside another query is called as subquery or nested query.

syntax:
======
select * from <tn> where <condition>(select * from .....(select * from .....));

Types of subqueries:
==========
1. Non-corelated subquery
2. Co-related subquery

1. Non-corelated subquery:
==========
i) single row subquery
ii) multiple row subquery
iii) multiple column subquery

i) single row subquery:
==========
- when a subquery return a single value.
- in this we will use the following operators are " = , < , > , <= , >= , != ".

Ex:
waq to display employees details who are getting the 1st highest salary?
==========
subquery statement = outer query + inner query
==========

step1: inner query:
==========
SQL> SELECT MAX(SAL) FROM EMP;-------> 5000

step2: outer query:
==========
SQL> SELECT * FROM EMP WHERE < inner query return value column name> = (inner query);

step3: subquery statement = (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 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 details who are earning the 2nd highest salary?
SQL> SELECT * FROM EMP WHERE SAL=
(SELECT MAX(SAL) FROM EMP WHERE SAL<
(SELECT MAX(SAL) FROM EMP);

EX:
waq to display employees details 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 3rd highest salary from emp table?
SQL> SELECT MAX(SAL) FROM EMP WHERE SAL<
(SELECT MAX(SAL) FROM EMP WHERE SAL<
(SELECT MAX(SAL) FROM EMP));

Ex:
waq to display employees details who are earning 3rd highest salary?
SQL> SELECT * FROM EMP WHERE SAL=
(SELECT MAX(SAL) FROM EMP WHERE SAL<

(SELECT MAX(SAL) FROM EMP WHERE SAL<
(SELECT MAX(SAL) FROM EMP)));

Nth	N+1
1ST	2Q
2ND	3Q
3RD	4Q
30TH	31Q
150TH	151Q

How to overcome the above problem?
==========
ii) multiple row subquery:
==========
- when a subquery return more than one value.
- we will use the following operators are " IN,ANY,ALL".

Ex:
waq to display the list of employees whose job is same as the employees SMITH,MARTIN jobs?

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

EX:
waq to display the list of employees who are getting the 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 details from each deptno wise?

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

ANY operator:
==========

- it returns TRUE if any one value is satisfied with the given expression value.

Ex:

i) IF X(40)>ANY(10,20,30)

X=09 ===> FALSE

X=25 ===> TRUE

X=40 ===> TRUE

ALL operator:

==========

- it returns TRUE if all values are satisfied with the given expression value.

Ex:

i) IF X(40)>ALL(10,20,30)

X=09 ===> FALSE

X=25 ===> FALSE

X=40 ===> TRUE

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');
```

EX:

waq to display employees whose salary is more than any salesman salary?

```
SQL> SELECT * FROM EMP WHERE SAL>ANY(SELECT SAL FROM EMP WHERE
JOB='SALESMAN');
```

ANY operator	ALL operator
==========	==========
X > ANY(list of values)	X>ALL(list of values)
X >= ANY(list of values)	X>=ALL(list of values)
X < ANY(list of values)	X<ALL(list of values)
X <= ANY(list of values)	X<=ALL(list of values)
X = ANY(list of values)	X=ALL(list of values)
X != ANY(list of values)	X!=ALL(list of values)

iii) multiple column subquery:

==========

- comparing multiple columns values of inner query with multiple columns values of outer query is known as "MCSQ".

syntax:

=======
select * from <table name> where(<column name1>,<column name2>,.....) IN(select * from .....);

EX:
waq to display the list of employees who are getting the maximum salary from each job wise?
SQL> SELECT * FROM EMP WHERE(JOB,SAL)IN(SELECT JOB,MAX(SAL) FROM EMP GROUP BY JOB);

EX:
waq to display employees whose employee job,mgr are same as the job,mgr of the employee ALLEN?
SQL> SELECT * FROM EMP WHERE(JOB,MGR)IN(SELECT JOB,MGR FROM EMP WHERE ENAME='Mr.ALLEN');

2. Co-related subquery:
====================
- In this mechanism first outer query is executed and later inner query will execute to give the final result.

How to find out "Nth" high / low salary:
================================
SELECT * FROM <TN> <TABLE ALIAS NAME1> WHERE N-1=(SELECT COUNT(DISTINCT <COLUMN NAME>)
FROM <TN> <TABLE ALIAS NAME2> WHERE <TABLE ALIAS NAME2>.<COLUMN NAME> (< / >) <TABLE ALIAS NAME1>.<COLUMN NAME>);

Here,

> - high salary

< - low salary

DEMO_TABLE:
============
SQL> SELECT * FROM TEST;

ENAME	SAL
ALLEN	85000
BLAKE	23000
WARD	85000
JONES	45000
ADAMS	12000

EX:
waq to find out the first highest salary employees details?

Solution:
========
If N=1 ===> N-1 ===> 1-1 ===> 0

SQL> SELECT * FROM TEST T1 WHERE 0=(SELECT COUNT(DISTINCT SAL) FROM TEST T2 WHERE T2.SAL>T1.SAL);

EX:
waq to find out the 4th highest salary employees details?
Solution:
========
If N=1 ===> N-1 ===> 4-1 ===> 3

SQL> SELECT * FROM TEST T1 WHERE 3=(SELECT COUNT(DISTINCT SAL) FROM TEST T2 WHERE T2.SAL>T1.SAL);

EX:
waq to find out the first lowest salary employees details?

Solution:
========
If N=1 ===> N-1 ===> 1-1 ===> 0

SQL> SELECT * FROM TEST T1 WHERE 0=(SELECT COUNT(DISTINCT SAL) FROM TEST T2 WHERE T2.SAL<T1.SAL);

How to display "TOP n" high / low salaries:
==========
SELECT * FROM <TN> <TABLE ALIAS NAME1> WHERE N>(SELECT COUNT(DISTINCT <COLUMN NAME>)
FROM <TN> <TABLE ALIAS NAME2> WHERE <TABLE ALIAS NAME2>.<COLUMN NAME> (< / >) <TABLE ALIAS NAME1>.<COLUMN NAME>);
Here,
> - high salary
< - low salary

EX:
waq to display top 3 highest salaries employees details?

Solution:
========
If N=3 ===> N> ===> 3>

```
SQL> SELECT * FROM TEST T1 WHERE 3>(SELECT COUNT(DISTINCT SAL) FROM TEST
T2
      WHERE T2.SAL>T1.SAL);
```

EX:
waq to display top 3 lowest salaries employees details?

Solution:
========
If N=3 ===> N> ===> 3>

```
SQL> SELECT * FROM TEST T1 WHERE 3>(SELECT COUNT(DISTINCT SAL) FROM TEST
T2
      WHERE T2.SAL<T1.SAL);
```

NOTE:
=======
1. To find out "Nth" high / low salary ----------> N-1
2. To display "Top n" high / low salaries ---------> N>

EXISTS operator:
===============
- it is a special operator which is used in co-related subquery only.
- to check the required row is existing in a table or not.
    > if a row is exists in a table then it return TRUE.
    > if a row is not exists in a table then it return FALSE.

syntax:
======
where exists(<inner 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 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);

==========

==========

VIEWS:

======

- it is subset of a base table.
- it will create with help of "select" query.
- view does not store any data/information.
- when we perform DML operations on view internally those operations are executed on base table and refelected in view table.
- without base table we cannot access view table in database.

Types of views:

==========

1. simple view
2. complex view

1. simple view :

==========

- when we created a view to access the required data from a single base table is known as "simple view".

syntax:

=======

create view <view name> as <select query>;

Ex:

create a view to access the data from DEPT table?

SQL> CREATE VIEW V1 AS SELECT * FROM DEPT;

TESTING:

SQL> INSERT INTO V1 VALUES(50,'DBA','HYD');

SQL> UPDATE V1 SET LOC='PUNE' WHERE DEPTNO=50;

SQL> DELETE FROM V1 WHERE DEPTNO=50;

SQL> SELECT * FROM V1;

NOTE:

=====

- simple views are allowed DML operations to perform on a base table in database.

Ex:

create a view to access EMPNO,ENAME,SALARY from emp table?

SQL> CREATE VIEW V2 AS SELECT EMPNO,ENAME,SAL FROM EMP;

TESTING:
SQL> INSERT INTO V2 VALUES(1122,'YUVIN',5500);
SQL> SELECT * FROM V2;

Ex:
create a view to access the employees details who are working under deptno is 20?
SQL> CREATE VIEW V3 AS SELECT * FROM EMP WHERE DEPTNO=20;

TESTING:
SQL> SELECT * FROM V3;

VIEW OPTIONS:
==========
I) WITH CHECK OPTION
II) WITH READ ONLY

I) WITH CHECK OPTION:
==========
- to restricted data on base table through a view.

EX:
create a view to display and accept the employees details whose salary is 3000?
SQL> CREATE VIEW V4 AS SELECT * FROM EMP WHERE SAL=3000;

TESTING:
SQL> SELECT * FROM V4;
SQL> INSERT INTO V4
VALUES(1122,'BHUVIN','HR',2345,'12-JUN-2017',2500,NULL,10);---ALLOWED

- In the above example we are accepting employees details into a base table whose salary is not equals to 3000.
- To avoid this problem and accept employees details whose salary is 3000 only.

Solution:
========
SQL> CREATE VIEW V5 AS SELECT * FROM EMP WHERE SAL=3000 WITH CHECK OPTION;

TESTING:
SQL> SELECT * FROM V5;
SQL> INSERT INTO V5
VALUES(1123,'YUVIN','HR',2345,'12-JUN-2017',2500,NULL,10);---NOT ALLOWED
SQL> INSERT INTO V5

VALUES(1123,'YUVIN','HR',2345,'12-JUN-2017',3000,NULL,10);----ALLOWED

II) WITH READ ONLY:
==========
- to restricted DML operations on a base table through a view.

EX:
create a view to restrict DML operations on base table?
SQL> CREATE VIEW V6 AS SELECT * FROM DEPT WITH READ ONLY;

TESTING:
SQL> SELECT * FROM V6;-------ALLOWED
SQL> INSERT INTO V6 VALUES(50,'DBA','HYD');----------NOT ALLOWED
SQL> UPDATE V6 SET LOC='PUNE' WHERE DEPTNO=10;---------NOT ALLOWED
SQL> DELETE FROM V6 WHERE DEPTNO=20;----------NOT ALLOWED

2) COMPLEX VIEW:
==========
- when a view is created based on :
  - multiple tables
  - by using group by
  - by using aggregative functions
  - by using having
  - by using set operators
  - by using distinct keyword
  - by using subquery
  - by using joins.
- by default complex views are not allowed DML operations.it is a read only view.

syntax:
======
create view <view name> as <select query>;

Ex:
create a view to access the data from multiple tables by using set operators?
SQL> CREATE VIEW V7 AS
  2 SELECT * FROM EMP_HYD
  3 UNION
  4 SELECT * FROM EMP_MUMBAI;

TESTING:
SQL> INSERT INTO V7 VALUES(1025,'ADAMS',55000);--------NOT ALLOWED
SQL> UPDATE V7 SET SAL=33000 WHERE EID=1021;----------NOT ALLOWED

SQL> DELETE FROM V7 WHERE EID=1023;----------NOT ALLOWED
SQL> SELECT * FROM V7;

Ex:
create a view to access the sum of salaries of each deptno wise?
SQL> CREATE VIEW V8 AS
  2 SELECT DEPTNO,SUM(SAL)AS SUM_OF_SALARY FROM EMP
  3 GROUP BY DEPTNO;
        - DML operations are not allowed.

How to drop a view:
==========
syntax:
=======
DROP VIEW <view name>;

EX:
SQL> DROP VIEW V1;

To see all views in oracle database:
==========
SQL> DESC USER_VIEWS;
SQL> SELECT VIEW_NAME FROM USER_VIEWS;
==========
======
SEQUENCE:
==========
        - it is a db object which is used to generate the sequence numbers on a specific column in the table automatically.
        - it will provide "auto incremental values" 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:
==========
        - to specify starting value of sequence.here "n" is a number.

minvalue n:
==========

- to show minimum value in the sequence.here "n" is a number.

increment by n:
==============

- fo specify incremental value in between seqeucne numbers.here "n" is a number.

maxvalue n:
===========

- to show maximum value from sequence.here "n" is a number.

no cycle:
========

- it a default attribute of sequence object.
- when we created a sequecne object with "NO CYCLE" then the set of sequence numbers are not repeat again and again.

cycle:
========

- when we created a sequecne object with "CYCLE" then the set of sequence numbers are repeat again and again.

no cache:
==========

- it is a default attribute of sequence object.
- cache is a temporary memory.
- when we created a sequence object with "NO CACHE" then the set of sequence numbers are saved in database memory directly.so that every user request will go to database and retrieving the required data from database and send to client application.
by this reason the burdon on database will increse and degrade the performance of database.

cache:
=====

- when we created a sequence object with "CACHE" then the set of sequence numbers are saved in database memory and also the copy of data is saved in cache memory.
so that now every user request will go to cache instead of database.and retrieving the required data
from cache memory and send to client application.so that we reduce the burdon on database and improve the performance of database.

NOTE:

=====

- to generate the sequence numbers on a column then we must use a pseudo column of sequence object is "NEXTVAL".

- by NEXTVAL pseudo column it will generate sequence numbers next by next.

syntax:
=======

<sequence name>.<nextval>

EX:

```
SQL> CREATE SEQUENCE SQ1
  2  START WITH 1
  3  MINVALUE 1
  4  INCREMENT BY 1
  5  MAXVALUE 3;
Sequence created.
```

TESTING:
=========

```
SQL> CREATE TABLE TEST55(SNO NUMBER(3),NAME VARCHAR2(10));

SQL> INSERT INTO TEST55 VALUES(SQ1.NEXTVAL,'&NAME');
Enter value for name: A

SQL> /
Enter value for name: B

SQL> /
Enter value for name: C

SQL> /
Enter value for name: D
ERROR at line 1:
ORA-08004: sequence SQ1.NEXTVAL exceeds MAXVALUE and cannot be instantiated.
```

ALTERING A SEQUECNE:
====================

syntax:
=======

alter sequence <sequence name> <attribute name> n;

EX:

```
SQL> ALTER SEQUENCE SQ1 MAXVALUE 5;
```

```
SQL> INSERT INTO TEST55 VALUES(SQ1.NEXTVAL,'&NAME');
Enter value for name: D

SQL> /
Enter value for name: E

SQL> SELECT * FROM TEST55;
```

EX:

```
SQL> CREATE SEQUENCE SQ2
  2  START WITH 3
  3  MINVALUE 1
  4  INCREMENT BY 1
  5  MAXVALUE 5
  6  CYCLE
  7  CACHE 2;
Sequence created.
```

TESTING:

```
=========
SQL> CREATE TABLE TEST56(SNO NUMBER(3),NAME VARCHAR2(10));

SQL> INSERT INTO TEST56 VALUES(SQ2.NEXTVAL,'&NAME');
Enter value for name: A

SQL>/
.....

SQL> /
.....

SQL> SELECT * FROM TEST56;
```

To view all sequence objects in oracle:

```
================================
SQL> USER_SEQUENCES;
SQL> SELECT SEQUENCE_NAME FROM USER_SEQUENCES;
```

How to drop a sequence object:

```
===========================
syntax:
========
```

drop sequence <sequence name>;

Ex:
SQL> DROP SEQUENCE SQ1;
==========
INDEXES:
========
- it is a database object which is used to retrieve the required row/rows from a table fastly.
- database index is similar to book index page in a text book.by using book index page how we are retrieving the required topic from a text book fastly same as by using database index
we are retrieving the required row/rows from a table fastly.
- database index can be created on a particular columns in the table and this column is called as "INDEXED KEY COLUMN".
- whenever we want to retrive the required row/rows from a table then we must use INDEXED KEY COLUMN under WHERE clause condition then only indexes are working.
- all databases are supporting the following two types of searching mechanisms.
    1. Table scan (default scan)
    2. Index scan

1. Table scan:
============
- In this scan,oracle server is searching the entire table for required data.so that it takes much time to give the required data to users.

EX:
SQL> SELECT * FROM EMP WHERE SAL=3000;

```
                                 SAL
                                ----------
                                800
                                1600
                                1250
                                2975
                                1250
WHERE SAL=3000;                 2850
                                2450
                                3000
                                5000
                                1500
                                1100
                                950
```

3000
1300

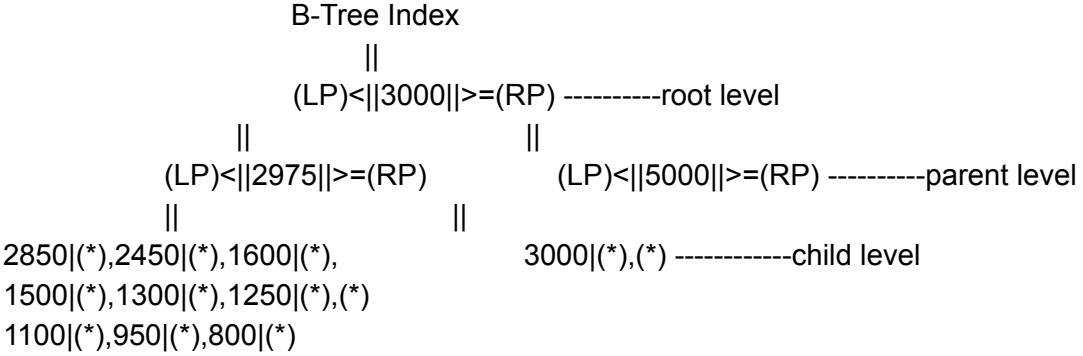
2. Index scan:
==========
- In this scan oracle server is searching the required row based on an indexed column wise.

i) B-tree index
ii) Bitmap index

i) B-tree index:
==========
- in this mechanism data can be organized in the form "Tree" structure by the system.

syntax:
=======
create index <index name> on <table name>(column name);

EX:
SQL> CREATE INDEX I1 ON EMP(SAL);
SQL> SELECT * FROM EMP WHERE SAL=3000;



Here,
LP - left pointer
RP - right pointer
* - rowid / rowaddress

ii) Bitmap index:
==========
- in this mechanism data can be organized in the form "Table" format by the system based on

bit numbers are 0,1.

Here,

0 is represent when condition is false

1 is represent when condition is true.

syntax:

=======

create bitmap index <index name> on <table name>(column name);

EX:

SQL> CREATE BITMAP INDEX BIT1 ON EMP(JOB);

SQL> SELECT * FROM EMP WHERE JOB='CLERK';

Bitmap Indexed Table Format

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	0	1	1	0	1
	(*)										(*)	(*)	(*)	

To view all indexes in oracle:

=========================

SQL> DESC USER_IND_COLUMNS;

SQL> SELECT COLUMN_NAME,INDEX_NAME FROM USER_IND_COLUMNS WHERE TABLE_NAME='EMP';

To view type of index in oracle:

==========================

SQL> DESC USER_INDEXES;

SQL> SELECT INDEX_NAME,INDEX_TYPE FROM USER_INDEXES WHERE TABLE_NAME='EMP';

INDEX_NAME	INDEX_TYPE
I1	NORMAL(B-TREE INDEX)
BIT1	BITMAP

How to drop index:
==========
syntax:
======
DROP INDEX <INDEX NAME>;

EX:
SQL> DROP INDEX I1;
SQL> DROP INDEX BIT1;
==========

PL/SQL
======

CURSOR:
========
- it is a temporary memory / sql private area.
- oracle supports the following two types of cursors.
    i) Explicit cursors
    ii) Implicit cursors

i) Explicit cursors:
==========
- these cursor arr created by users for retrieving multiple rows from a table in pl/sql.
- cursor can hold multiple rows but we can access a single row at a time.
- to create explicit cursor then we follow the following four steps:

step1: Declare cursor variable:
==========
syntax:
=======
declare cursor <cursor name> is <select query>;

step2: Open cursor connection:
==========
syntax:
======
open <cursor name>;

step3: Fetching rows from cursor table:
==========
syntax:
=======

fetch <cursor name> into <variables>;

step4: Close cursor connection:
==========
syntax:
=======
close <cursor name>;

Attributes of an explicit cursor:
==========
- to check the status of cursor.

syntax:
======
<cursor name>%<attribute name>;

i) %isopen:
==========
- it is a default attribute of cursor.
- it returns TRUE when cursor connection is successfully open otherwise FALSE.

ii) %notfound:
==========
- it returns TRUE when cursor is not having data otherwise FALSE.

iii) %found:
==========
- it return TRUE when cursor is having data otherwise FALSE.

iv) %rowcount:
==========
- it returns how many no.of rows are fetched from a cursor table.

EX:
write a cursor program to fetch a single row from a table?

```
SQL> DECLARE CURSOR C1 IS SELECT ENAME,SAL FROM EMP;
  2  v_ENAME VARCHAR2(10);
  3  v_SAL NUMBER(8,2);
  4  BEGIN
  5  OPEN C1;
  6  FETCH C1 INTO v_ENAME,v_SAL;
  7  DBMS_OUTPUT.PUT_LINE(v_ENAME||','||v_SAL);
  8  CLOSE C1;
  9  END;
 10  /
```

OUTPUT:
=========
SMITH,800

EX:
write a cursor program to fetch multiple rows from a table?

```
SQL> DECLARE CURSOR C1 IS SELECT ENAME,SAL FROM EMP;
  2  v_ENAME VARCHAR2(10);
  3  v_SAL NUMBER(8,2);
  4  BEGIN
  5  OPEN C1;
  6  FETCH C1 INTO v_ENAME,v_SAL;
  7  DBMS_OUTPUT.PUT_LINE(v_ENAME||','||v_SAL);
  8  FETCH C1 INTO v_ENAME,v_SAL;
  9  DBMS_OUTPUT.PUT_LINE(v_ENAME||','||v_SAL);
 10 CLOSE C1;
 11 END;
 12  /
```

OUTPUT:
=========
SMITH,800
ALLEN,1600

- In the above example we used multiple fetch statements for fetching multiple rows from a table.To avoid to write multiple fetch statements we use "Looping statements".

I) By using "Simple Loop":
======================

```
SQL> DECLARE CURSOR C1 IS SELECT ENAME,SAL FROM EMP;
  2  v_ENAME VARCHAR2(10);
  3  v_SAL NUMBER(8,2);
  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  /
```

OUTPUT:
=========
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":
======================
```
SQL> DECLARE CURSOR C1 IS SELECT ENAME,SAL FROM EMP;
  2  v_ENAME VARCHAR2(10);
  3  v_SAL NUMBER(8,2);
  4  BEGIN
  5  OPEN C1;
  6  FETCH C1 INTO v_ENAME,v_SAL; ----------------------------> fetching 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 will be continue upto
last row.
 11  END LOOP;
 12  CLOSE C1;
 13  END;
 14  /
```

OUTPUT:
=========
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  /
```

OUTPUT:
==========
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) Implicit cursors:
==========

- these cursors are created by the system when user perform DML operations on a table in database.
- these cursor are used to store the information / the status of a DML command is executed successfully or not.

EX:
SQL> INSERT INTO DEPT VALUES(50,'SAP','HYD');
1 row created.

SQL> UPDATE DEPT SET LOC='PUNE' WHERE DEPTNO=50;
1 row updated.

SQL> DELETE FROM DEPT WHERE DEPTNO=50;
1 row deleted.
==========
==========
EXCEPTION HANDLING:
==========

What is an Exception:
==========

- it is a runtime error / execution error.

What is Exception Handling:
==========

- to avoid abnormal terimination of a program execution process.

> PL/SQL suports the following two types of exceptions.those are,

1) System defined exceptions
2) User defined exceptions

1) System defined exceptions:
==========

- these exceptions are called as pre-defined exceptions in pl/sql.

ex: no_data_found,too_many_rows,zero_divide,.....etc

no_data_found:
==========

- If our required row is not found in a table then oracle server return an exception is "no data found" exception.

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

SQL> /
Enter value for empno: 1122
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 4
```

- To handle the above exception oracle provide a pre-defined exception name is "NO_DATA_FOUND".

Handling an exception:

```
==================
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('SORRY,RECORD IS NOT FOUND.PLZ TRY AGAIN!!!');
  9  END;
 10  /
Enter value for empno: 7788
SCOTT

SQL> /
Enter value for empno: 1122
SORRY,RECORD IS NOT FOUND.PLZ TRY AGAIN!!!
```

too_many_rows:

```
=============
```

- when we try to retrieving multiple rows by using "select.....into" statement then oracle return an exception is "exact fetch returns more than requested number of rows".

EX:

DEMO_TABLE:

```
=============
SQL> SELECT * FROM TEST;

ENAME                SAL
----------           ----------
```

```
SMITH          23000
JONES          45000

SQL> DECLARE
  2  v_SAL NUMBER(8,2);
  3  BEGIN
  4  SELECT SAL INTO v_SAL FROM TEST;
  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
```

- To handle the above exception oracle provide a pre-defined exception name is "too_many_rows" exception.

Handling an exception:
==========

```
SQL> DECLARE
  2  v_SAL NUMBER(8,2);
  3  BEGIN
  4  SELECT SAL INTO v_SAL FROM TEST;
  5  DBMS_OUTPUT.PUT_LINE(v_SAL);
  6  EXCEPTION
  7  WHEN TOO_MANY_ROWS THEN
  8  DBMS_OUTPUT.PUT_LINE('TABLE IS HAVING MORE THAN ONE ROW.PLZ CHECK
IT!!!');
  9  END;
 10  /
TABLE IS HAVING MORE THAN ONE ROW.PLZ CHECK IT!!!
```

zero_divide:
==========

- when we perform division with zero then oracle returne an exception is "divisor is equal to zero".

EX:

```
SQL> DECLARE
  2  X NUMBER(7);
  3  Y NUMBER(8);
  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: 5
2

SQL> /
Enter value for x: 10
Enter value for y: 0
ERROR at line 1:
ORA-01476: divisor is equal to zero
```

- To handle the above exception oracle provide a pre-defined exception name is "zero_divide".

Handling an exception:
==========

```
SQL> DECLARE
  2  X NUMBER(7);
  3  Y NUMBER(8);
  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: 5
2
SQL> /
Enter value for x: 10
Enter value for y: 0
SECOND NUMBER SHOULD NOT BE ZERO
```

2) User defined exceptions:

==========

- there are three steps to design user defined exception name.

step1: declare user defined exception name:
==========
syntax:
=======

<user defined exception name> exception;

step2: raise a user defined exception name:
==========
syntax-1:
=========

raise <user defined exception name>;

syntax-2:
========

raise_application_error(number,message);

Note:
=====

- "raise" statement will raise and handle an exception in the program.
- "raise_application_error" statement will raise an exception but not handle exception.

Here,

number : it return exception number -20001 to -20999.
message : it return user defined message.

step3: handling exception by using user defined exception name:
==========
syntax:
=======

exception
when <user defined exception name> then
<statement>;
end;

- when we create our own exception name and raise an explicitly whenever we required.this type of exceptions are called as "user defined exceptions".
- to create a user defined exception name then we follow the following three steps are,

Step1: Declare user defined exception name:
==========

syntax:
=======

<UD exception name> Exception;

Step2: Raise user defined exception name:
==========

Method-1:
========

Raise <UD exception name>;

Method-2:
========

Raise_application_error(number,message)

Here,

Number : it should be from -20000 to -20999.

Message : it display UD message.

Note:
=====

- Here Raise statement will raise an exception and aslo handle an exception. whereas Raise_application_error() will raise ane exception but not handle an exception.

Step3: Handling an exception with user defined exception name:
==========

syntax:
======

```
Exception
when <UD exception name> then
< statement>;
End;
/
```

i) By using "Raise" statement:
==========

EX:

```
SQL> DECLARE
  2  X NUMBER(5);
  3  Y NUMBER(5);
  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  DBMS_OUTPUT.PUT_LINE('SECOND NUMBER SHOULD NOT BE ZERO');
 18  END;
 19  /
Enter value for x: 10
Enter value for y: 2
5
SQL> /
Enter value for x: 10
Enter value for y: 0
SECOND NUMBER SHOULD NOT BE ZERO
```

ii) By using "Raise_application_error()" statement:

==========

EX:

```
SQL> DECLARE
  2  X NUMBER(5);
  3  Y NUMBER(5);
  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 BE ZERO');
 18  END;
 19  /
Enter value for x: 10
Enter value for y: 5
```

```
2
SQL> /
Enter value for x: 10
Enter value for y: 0
ERROR at line 1:
ORA-20478: SECOND NUMBER NOT BE ZERO
ORA-06512: at line 17
================================================================================
==================
```

SUB BLOCKS:

```
============
```

- it is a named block which will save the code in database automatically.
- pl/sql supports the following three types of sub blocks objects are,
    1. stored procedures
    2. stored functions
    3. triggers

1. stored procedures:

```
==================
```

- it is a named block which contains "pre-compiled code".
- it is a block of code to perform some operations on the given input values

but it may be (or) may not be return a value.

- when we use "OUT" parameters then only procedures are return a value otherwise never return any value.

syntax:

```
======
CREATE [OR REPLACE] PROCEDURE <PNAME>(<parameter name1> [mode type]
<datatype>,.....)
IS
<DECLARE VARIABLES>;
BEGIN
<PROCEDURE BODY / STATEMENTS>;
END;
/
```

How to call a stored procedure:

```
===========================
```

syntax:

```
======
EXECUTE <PNAME>(values);
```

Types of parameters modes:

==========

- there two types of parameters modes.
    - i) IN mode:
      ==========
        - default parameter of a stored procedure.
        - to store input values which was given by user at runtime.

    - ii) OUT mode:
      ===========
        - when a procedure want to return a value then we should use "OUT" parameters.
        - it return output value.

Examples on "IN" parameters:
==========

EX:
create a SP to display sum of two numbers by using IN parameters?

```
SQL> CREATE OR REPLACE PROCEDURE SP1(X IN NUMBER,Y IN NUMBER)
  2  IS
  3  BEGIN
  4  DBMS_OUTPUT.PUT_LINE(X+Y);
  5  END;
  6  /
```

OUTPUT:
========

```
SQL> EXECUTE SP1(10,20);
30
```

NOTE:
=====

- if we want to view all subblock objects(procedure/function/trigger) in oralce then we use a datadictionary is "user_objects".

EX:

```
SQL> DESC USER_OBJECTS;
SQL> SELECT OBJECT_NAME FROM USER_OBJECTS WHERE
OBJECT_TYPE='PROCEDURE';
```

NOTE:
=====

- if we want to view the source code of sub block object(procedure/function/trigger) in oracle then use a datadictionary is "user_source".

EX:
SQL> DESC USER_SOURCE;
SQL> SELECT TEXT FROM USER_SOURCE WHERE NAME='SP1';

EX:
create a SP to input EMPNO and display that employee NAME,SALARY details from emp table?

```
SQL> CREATE OR REPLACE PROCEDURE SP2(p_EMPNO IN NUMBER)
  2  IS
  3  v_ENAME VARCHAR2(10);
  4  v_SAL NUMBER(8,2);
  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  /
```

OUTPUT:
========
SQL> EXECUTE SP2(7900);
JAMES,950

Examples on "OUT" parameters:
==========
Ex:
create a SP to return the cube of the given value by using OUT parameter?

```
SQL> CREATE OR REPLACE PROCEDURE SP3(X IN NUMBER,Y OUT NUMBER)
  2  IS
  3  BEGIN
  4  Y:=X*X*X;
  5  END;
  6  /
```

OUTPUT:

```
SQL> EXECUTE SP3(5);
ERROR at line 1:
ORA-06550: line 1, column 7:
PLS-00306: wrong number or types of arguments in call to 'SP3'
```

- To overcome the above problem we need to follow the following 3 steps are,

step1: Declare a bind / referenced variables for "OUT" parameters of SP:
==========
syntax:
======
var[iable] <bind/referenced variable name> <datatype>[size];

step2: Adding this bind / referenced variables to a SP:
==========
syntax:
======
execute <pname>(value1,value2,.....,:<bind variables name1>,.....);

step3: Print bind / referenced variables :
==========
syntax:
======
print < bind / referenced variable name>;

OUTPUT:
=======
```
SQL> VAR A NUMBER;
SQL> EXECUTE SP3(5,:A);
SQL> PRINT A;

         A
----------
       125
```

EX:
create a SP to input EMPNO and return that employee provident fund and professional tax at 5%,10% on their basic salary by using "OUT" parameters?
```
SQL> CREATE OR REPLACE PROCEDURE SP4(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  /
```

OUTPUT:
========
SQL> VAR rPF NUMBER;
SQL> VAR rPT NUMBER;
SQL> EXECUTE SP4(7788,:rPF,:rPT);

SQL> PRINT rPF rPT;

RPF

150

RPT

300

How to drop a stored procedure:
==========
syntax:
=======
DROP PROCEDURE <PNAME>;

EX:
SQL> DROP PROCEDURE SP1;
==========
STORED FUNCTIONS:
==========
- Function is a block of code to perform some task and it must return a value.
- these functions are created by user as per client requirements so that these functions are also called as "user defined functions" in oracle.

syntax:
======
create [or replace] function <fname>(<parameters name1> <datatype>,.....)
return <return variable DATATYPE>
as
<declare variables>;
begin
<function body / statements>;
return <return variable NAME>;
end;

/

How to call a stored function:
==========
syntax:
=======
SELECT <FNAME>(VALUES) FROM DUAL;

EX:
create a SF to input EMPNO and return that ENAME 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  /
```

OUTPUT:
=======

```
SQL> SELECT SF1(7900) FROM DUAL;

SF1(7900)
--------------------
JAMES
```

EX:
create a SF to return sum of salary of the 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  /
```

OUTPUT:
=========

```
SQL> SELECT SF2('SALES') FROM DUAL;

SF2('SALES')
------------------------
       9400
```

EX:
create a SF to return the no.of employees are joined in between the given two dates 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  /
```

OUTPUT:
=========

```
SQL> SELECT SF3('01-JAN-1981','31-DEC-1981') FROM DUAL;

SF3('01-JAN-1981','31-DEC-1981')
------------------------------------------------------------
                          10
```

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?

```
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  /
```

OUTPUT:
==========

```
SQL> SELECT SF4(7788) FROM DUAL;

 SF4(7788)
----------
      4200
```

How to drop a stored function:
==========

syntax:
=======

```
DROP FUNCTION <FNAME>;
```

EX:

```
DROP FUNCTION SF1;
```

==========

TRIGGERS:
==========

- it is a named block which will execute by the system automatically when we perform DDL,DML operations over database.
- there are two types of triggers in oracle.
    1. DML triggers
    2. DDL triggers

Purpose of triggers:
==========

> to raise security alerts in an application.
> to controll DML,DDL operations based on business logical conditions.
> for validating data
> for auditing

1. DML triggers:
==========

- when we created a trigger object based on DML(insert,update,delete) commands are called as "DML triggers".
- these triggers are executed by system automatically when we perform DML commands on a specific table.

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:
==========

i) Before event:
==========

- when we created a trigger object with "BEFORE" event.
  - First : Trigger body executed.
  - Later : DML command will execute.

i) After event:
==========

- when we created a trigger object with "AFTER" event.
  - First : DML command is executed.
  - Later : Trigger body will execute.

NOTE:
=====

- But both are providing same result.

Levels of triggers:
==========

- trigger can be created at two levels.
  1. statement level triggers
  2. row level triggers

1. statement level triggers:
==========

- in this level a trigger body is executing only one time for a DML operation.

DEMO_TABLE:
==========
SQL> SELECT * FROM TEST;

```
       EID   ENAME    SAL
    ---------- ----------      ----------
      1021 SMITH     15000
      1022 ALLEN     23000
      1023 JONES     15000
      1024 MILLER    43000
```

EX:
```
SQL> CREATE OR REPLACE TRIGGER TR1
  2  AFTER UPDATE ON TEST
  3  BEGIN
  4  DBMS_OUTPUT.PUT_LINE('HELLO');
  5  END;
  6  /
```

TESTING:
```
SQL> UPDATE TEST SET SAL=18000 WHERE SAL=15000;
HELLO
2 rows updated.
```

2. row level triggers:
==========
- in this level a trigger body is executing for each row wise for DML operation.so that we must use "for each row" clause.

EX:
```
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  /
```

TESTING:
==========
```
SQL> UPDATE TEST SET SAL=12000 WHERE SAL=18000;
```

HELLO
HELLO
2 rows updated.

BIND VARIABLES:
==========
- these are working just like normal variables in a program.
i) :NEW :
========
- to store the values when we are inserting data into a table.
syntax:
=======
:NEW.<column name>;

ii) :OLD :
=======
- to store the values when we are deleting data from a table.
syntax:
=======
:OLD.<column name>;

NOTE:
=====
- whenever we want to perform UPDATE operation then we use the combination of :NEW and :OLD variables.
- these bind variables are used in row level triggers only.

Examples on raising a security alert in an application:
==========
Ex:
create a trigger to raise a alert for INSERT operation on a table?

```
SQL> CREATE OR REPLACE TRIGGER TRINSERT
  2  AFTER INSERT ON TEST
  3  BEGIN
  4  RAISE_APPLICATION_ERROR(-20478,'Alert!!!SOMEONE IS INSERTING A NEW ROW
INTO YOUR TABLE.Plz..CHECK IT!!!');
  5  END;
  6  /
```

TESTING:
========

```
SQL> INSERT INTO TEST VALUES(1026,'SCOTT',48000);
ERROR at line 1:
ORA-20478: Alert!!!SOMEONE IS INSERTING A NEW ROW INTO YOUR TABLE.Plz..CHECK
```

IT!!!

FOR UPDATE:
==========
```
SQL> CREATE OR REPLACE TRIGGER TRUPDATE
  2  AFTER UPDATE ON TEST
  3  BEGIN
  4  RAISE_APPLICATION_ERROR(-20471,'Alert!!! SOMEONE IS UPDATING A ROW IN
YOUR TABLE.Plz..CHECK IT!!!');
  5  END;
  6  /
```

FOR DELETE:
==========
```
SQL> CREATE OR REPLACE TRIGGER TRDELETE
  2  AFTER DELETE ON TEST
  3  BEGIN
  4  RAISE_APPLICATION_ERROR(-20471,'Alert!!! SOMEONE IS DELETING A ROW FROM
YOUR TABLE.Plz..CHECK IT!!!');
  5  END;
  6  /
```

Ex:
create a trigger to raise a alert for DML operations on a table?
```
SQL> CREATE OR REPLACE TRIGGER TRDML
    AFTER INSERT OR UPDATE OR DELETE ON TEST
    BEGIN
    RAISE_APPLICATION_ERROR(-20471,'Alert!!! SOMEONE IS PERFORMING DML
OPERATION ON YOUR TABLE.Plz..CHECK IT!!!');
    END;
    /
```
- Here, all DML operations are restricted.

Examples on controlling DML operations based on business logical conditions:
==========
EX:
create a trigger to control all DML operations on every weekends on a table?
```
SQL> CREATE OR REPLACE TRIGGER TRWEEKENDS
  2  AFTER INSERT OR UPDATE OR DELETE ON BRANCH
  3  BEGIN
  4  IF TO_CHAR(SYSDATE,'DY')IN('SAT','SUN') THEN
  5  RAISE_APPLICATION_ERROR(-20456,'WE CANNOT PERFORM DML OPERATIONS ON
WEEKENDS');
```

```
  6  END IF;
  7  END;
  8  /
```
TESTING:
```
SQL> CREATE TABLE BRANCH(BCODE NUMBER(4),BNAME VARCHAR2(10),BLOC
VARCHAR2(10));
SQL> INSERT INTO BRANCH VALUES(1021,'SBI','HYD');
```

EX:
create a trigger to control all DML operations on a table between 9am to 5pm?

Logic:
=====

24hrs FORMAT
==========
9am(9) : 9:00:00 to 9:59:59 ----------> comes under 9 o clock.
5pm(17) : 5:00:00 to 5:59:59 ----------> upto 6 o clock
4pm(16) : 4:00:00 to 4:59:59 ----------> upto 5 o clock

```
SQL> CREATE OR REPLACE TRIGGER TRTIME
  2  AFTER INSERT OR UPDATE OR DELETE ON BRANCH
  3  BEGIN
  4  IF TO_CHAR(SYSDATE,'HH24')BETWEEN 9 AND 16 THEN
  5  RAISE_APPLICATION_ERROR(-20478,'SORRY,INVALID TIME');
  6  END IF;
  7  END;
  8  /
```

TESTING:
=========
```
SQL> INSERT INTO BRANCH VALUES(1022,'HDFC','PUNE');
```

Examples on validating data:
==========
Ex:
create a trigger to validate insert operation on a table if new salary is less than to 10000?
```
SQL> CREATE OR REPLACE TRIGGER TRIN
  2  BEFORE INSERT ON TEST
  3  FOR EACH ROW
  4  BEGIN
  5  IF :NEW.SAL<10000 THEN
  6  RAISE_APPLICATION_ERROR(-20478,'NEW SALARY SHOULD NOT BE LESS THAN TO
```

```
10000');
  7  END IF;
  8  END;
  9  /
```

TESTING:

```
SQL> INSERT INTO TEST VALUES(1026,'JAMES',9500);---------NOT ALLOWED
SQL> INSERT INTO TEST VALUES(1026,'JAMES',1200);---------ALLOWED
```

EX:

create a trigger to validate delete operation on a table if we try to delete the employee SMITH details?

```
SQL> CREATE OR REPLACE TRIGGER TRDEL
  2  BEFORE DELETE ON TEST
  3  FOR EACH ROW
  4  BEGIN
  5  IF :OLD.ENAME='SMITH' THEN
  6  RAISE_APPLICATION_ERROR(-20569,'WE CANNOT DELETE SMITH DETAILS');
  7  END IF;
  8  END;
  9  /
```

TESTING:

```
==========
SQL> DELETE FROM TEST WHERE ENAME='JAMES';-----ALLOWED
SQL> DELETE FROM TEST WHERE ENAME='SMITH';-----NOT ALLOWED
```

Ex:

create a trigger to validate update operation on a table if new salary is less than to old salary?

```
SQL> CREATE OR REPLACE TRIGGER TRUP
  2  BEFORE UPDATE ON TEST
  3  FOR EACH ROW
  4  BEGIN
  5  IF :NEW.SAL<:OLD.SAL THEN
  6  RAISE_APPLICATION_ERROR(-20587,'INVALID SALARY');
  7  END IF;
  8  END;
  9  /
```

TESTING:

```
=========
SQL> UPDATE TEST SET SAL=10000 WHERE SAL=12000;-----NOT ALLOWED
```

SQL> UPDATE TEST SET SAL=14000 WHERE SAL=12000;-----ALLOWED

AUDITING:
=========
- when we perform some operations on a table those operational data will save in another table is called as "audit table".

EX:
SQL> CREATE TABLE EMP44(EID NUMBER(4),ENAME VARCHAR2(10));-------------------------MAIN TABLE
SQL> CREATE TABLE EMP44_AUDIT(EID NUMBER(4),AUDIT_INFR VARCHAR2(100));------AUDIT TABLE

```
SQL> CREATE OR REPLACE TRIGGER TRAUDIT1
  BEFORE INSERT ON EMP44
  FOR EACH ROW
  BEGIN
  INSERT INTO EMP44_AUDIT VALUES(:NEW.EID,'SOMEONE INSERTED A NEW ROW
INTO A TABLE ON:'||
  TO_CHAR(SYSDATE,'DD-MON-YYYY HH:MI:SS PM'));
  END;
  /
```

TESTING:
=========
SQL> INSERT INTO EMP44 VALUES(1021,'ALLEN');
SQL> SELECT * FROM EMP44;
SQL> SELECT * FROM EMP44_AUDIT;

For UPDATE:
===========
```
CREATE OR REPLACE TRIGGER TRAUDIT2
  BEFORE UPDATE ON EMP44
  FOR EACH ROW
  BEGIN
  INSERT INTO EMP44_AUDIT VALUES(:OLD.EID,'SOMEONE UPDATED A ROW IN A
TABLE ON:'||
  TO_CHAR(SYSDATE,'DD-MON-YYYY HH:MI:SS PM'));
  END;
  /
```
TESTING:
SQL> UPDATE EMP44 SET ENAME='JONES' WHERE EID=1021;

For DELETE:
==========
```
CREATE OR REPLACE TRIGGER TRAUDIT3
  BEFORE DELETE ON EMP44
  FOR EACH ROW
  BEGIN
  INSERT INTO EMP44_AUDIT VALUES(:OLD.EID,'SOMEONE DELETED A ROW FROM A
TABLE ON:'||
  TO_CHAR(SYSDATE,'DD-MON-YYYY HH:MI:SS PM'));
  END;
  /
```
TESTING:
```
SQL> DELETE FROM EMP44 WHERE EID=1022;
```

2. DDL triggers:
==========
- when we create a trigger based on DDL commands(create/alter/rename/drop) are called as "DDL triggers".
- these triggers are executed by the system automatically when we perform DDL operations on a specific database.so that DDL triggers are also called as "DB triggers".

syntax:
======
```
create [or replace] trigger <trigger name>
before / after create or alter or rename or drop on <username/db name>.schema
[for each row]
begin
<trigger body / statements>;
end;
/
```

EX:
create a trigger to raise security alert on CREATE command?
```
SQL> CREATE OR REPLACE TRIGGER TRDDL
  2  AFTER CREATE ON MYDB9AM.SCHEMA
  3  BEGIN
  4  RAISE_APPLICATION_ERROR(-20478,'WE CANNOT PERFORM CREATE OPERATION
ON MYDB9AM DATABASE');
  5  END;
  6  /
```

TESTING:
========

SQL> CREATE TABLE T1(SNO NUMBER(2));-----NOT ALLOWED

==========

NORMALIZATION:

==========

What is Normalization ?

==========

- it is a technique which is used to decompose(i.e divide) a table data into multiple tables.

Where we use Normalization?

==========

- DB designing level.

Why Normalization ?

==========

EX: Branch_Student_Details

STID	SNAME	BRANCH	HOD	OFFICE_NUMBER
1021	smith	cse	Mr.x	040-22334455
1022	allen	cse	Mr.y	040-22334455
1023	ward	cse	Mr.x	040-22334455
1024	miller	cse	Mr.y	040-22334455
1025	jones	cse	Mr.x	040-22334455

Disadvantages:

==========

- Dataredundancy problem(i.e duplicate data).
- It occupied more memory.
- Data inconsistency problem(i.e irregular data).
- Insertion problem.
- Updation problem.
- Deletion problem.

- To overcome the above problems we need to use a technique is known as "Normalization".

Solution:

==========

(pk) Branch_Details Student_Details (fk)

Bcode	Bname	HOD	Office_number	Stid	Sname	Bcode
1	cse	Mr.x	040-22334455	1021	smith	1
				1022	allen	1
				1023	ward	1
				1024	miller	1
				1025	jones	1

Advantages:

- To avoid dataredundancy problem(i.e no duplicate data).
- It occupied less memory.
- To avoid data inconsistency problem(i.e regular data).
- To avoid Insertion problem.
- To avoid Updation problem.
- To avoid Deletion problem.

Types of Normaliztion forms:

- First normal form(1NF)
- Second normal form(2NF)
- Third normal form(3NF)
- Boyce-codd normal form(BCNF)
- Fourth normal form(4NF)
- Fifth normal form(5NF)

First normal form(1NF):

- For a table to be in the First Normal Form, it should follow the following 4 rules:
  1. Each column should contain atomic value (atomic = single value).
  2. A column should contain values that are same datatype.
  3. All the columns in a table should have unique names.
  4. The order in which data is stored, does not matter.

EX: Student_details

Stid	Sname	Bcode

1021	smith	2
1022	allen	1
1023	ward	3

Second normal form(2NF):
==========

- For a table to be in the Second Normal Form, it must satisfy two conditions:
    1. The table should be in the First Normal Form.
    2. There should be no Partial Dependency.

WHAT IS DEPENDENCY:
==========

- IN A TABLE IF NON-KEY COLUMNS (NON PRIMARY KEY) ARE DEPENDS ON KEY COLUMN (PRIMARY KEY) THEN IT IS CALLED AS FULLY DEPENDENCY / FUNCTIONAL
DEPENDENCY.

(PK)
EX: STID SNAME BRANCH ADDRESS

- Here, “STID “IS A KEY COLUMN and “SNAME”,”BRANCH”,”ADDRESS” ARE NON-KEY COLUMNS.
- These non-key columns are linked with key column is STID.so that in this table there is no partial dependency columns.

WHAT IS PARTIAL DEPENDENCY:
==========

- IN A TABLE IF NON-KEY COLUMN DEPENDS ON PART OF THE KEY COLUMN, THEN IT IS CALLED AS PARTIAL DEPENDENCY.

<PRIMARY KEY (stu_id,sub_id) / COMPOSITE PRIMARY KEY>
EX: STU_ID SUB_ID STU_MARKS TEACHER

- Here, “STU_ID and SUB_ID “ IS A KEY COLUMNS - ” STU_MARKS”,” TEACHER” ARE NON-KEY
COLUMNS. THEN “TEACHER” DEPENDS ON “SUB_ID” BUT NOT “STU_ID” COLUMN.
- Here we found a partial dependency column is "TEACHER" so that we need to do decompose a table like below,

Subject_Table Student_table
========== ==========
(pk) (pk) (fk)

SUB_ID SUB_NAME TEACHER STU_ID STU_MARKS SUB_ID

Third normal form(3NF):
==========

- For a table to be in the third normal form there is two conditions.
    1. It should be in the Second Normal form.
    2. And it should not have Transitive Dependency.

TRANSITIVE DEPENDENCY:
==========

- IN TABLE IF NON-KEY COLUMN DEPENDS ON ANOTHER NON-KEY COLUMN, THEN IT IS CALLED AS TRANSITIVE DEPENDENCY.

EX:

|--------CPK----------|

STUDENT_ID	SUBJECT_ID	STU_MARKS	EXAM_NAME	TOTAL_MARKS

- Here, “STU_ID and SUB_ID “ ARE KEY COLUMNS . ” EXAM_NAME”,” TOTAL_MARKS”
ARE NON-KEY COLUMNS. THEN “TOTAL_MARKS” DEPENDS ON “EXAM_NAME” BUT NOT “STU_ID and SUB_ID” COLUMNS.
- Here we found transitive dependency columns are "EXAM_NAME" and "TOTAL_MARKS"
so that we need to do decompose the above table into multiple tables.

(pk) Exam_Table (pk) Score_Table (fk)

SUB_ID	EXAM_NAME	TOTAL_MARKS	STUDENT_ID	STU_MARKS	SUB_ID

Boyce-codd normal form(BCNF):
==========

- For a table to satisfy the Boyce - Codd Normal Form, it should satisfy the following two conditions:
    1. It should be in the Third Normal Form.
    2. And, for any dependency A → B, A should be a super key.

SUPER KEY:
==========

- A COLUMN (OR) COMBNATION OF COLUMNS WHICH ARE UNIQUELY IDENTIFYING
A ROW IN A TABLE IS CALLED AS SUPER KEY.

CANDIDATE KEY:
==============

- A MINIMAL SUPER KEY WHICH IS UNIQUELY IDENTIFYING A ROW IN A TABLE IS CALLED AS CANDIDATE KEY.

(OR)

- A SUPER KEY WHICH IS SUBSET OF ANOTHER SUPER KEY, BUT THE COMBINATION
OF SUPER KEYS ARE NOT A CANDIDATE KEY.

EX:

STUDENT TABLE

STUDENT_ID	NAME	BRANCH	MAILID	REG_NUMBER

Super key columns:
==========

student_id		student_id + mailid	
mailid		mailid + reg_number	student_id + mailid + reg_number
reg_number		reg_number + student_id	

Candidate key columns:
==========

student_id
mailid
reg_number

Ex:

Professor Table

|----------cpk ----------|

PROFESSOR_ID	SUBJECT(B)	PROFESSOR(A)
1	java	p.java
2	java	p.java

- Here, PROFESSOR column depends on SUBJECT so that PROFESSOR should be

super key but not a super key.

- Now to make a PROFESSOR column is a super key and SUBJECT is non-super key column in the table like below,

Professor Table

|----------cpk----------|

professor_id	professor	Subject
1	p.java	java
2	p.java	java

5. Fourth normal form(4NF):

==========

- For a table to satisfy the Fourth Normal Form, it should satisfy the following two conditions:
  1. It should be in the Boyce-Codd Normal Form.
  2. A table does not contain more than one independent multi valued attribute / Multi Valued Dependency.

Multi valued Dependency:

==========

- In a table one column same value mapping with multiple values of another column is called as multi valued dependency.

EX:

COLLEGE ENROLLMENT TABLE (5NF)

STUDENT_ID	COURSE	HOBBY
1	ORACLE	Cricket
1	JAVA	Reading
1	C#	Hockey

Mapping with multiple values of columns: (Decomposing table)

==========

Course_details (4NF)

==========

STUDENT_ID	COURSE
1	oracle
1	java

Hobbies_details(4NF)

==========

STUDENT_ID	HOBBY
1	cricket
1	reading

1 c# 1 hockey

Fifth Normal Form (5NF):
==========

- If a table is having multi valued attributes and also that table cannot decomposed into multiple tables are called as fifth normal form.

EX:

COLLEGE ENROLLMENT TABLE (5NF)

STUDENT_ID	COURSE	HOBBY
1	ORACLE	Cricket
1	JAVA	Reading
1	C#	Hockey

==========
PSUEDO COLUMNS:
==========

- are working just like normal columns.
  - i) ROWID
  - ii) ROWNUM

ROWID :
=======

- it is a psuedo column which is used to generate row address / row identity for each row wise on a table automatically.
- by using ROWID psuedo column we will delete duplicate rows from a table.
- when we insert a new row into a table internally oracle server is creating a unique row identity for each row wise and these rowid's are permanently saved in database.

EX:

```
SQL> SELECT ROWID,ENAME,DEPTNO FROM EMP;
SQL> SELECT ROWID,EMP.* FROM EMP WHERE DEPTNO=10;
```

EX:

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

MIN(ROWID)
------------------
AAAW3qAAHAAAAGMAAA
```

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

MAX(ROWID)
------------------
AAAW3qAAHAAAAGMAAN
```

How to delete multiple duplicate rows except one duplicate row from a table:

==========

DEMO_TABLE:

==========

```
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
```

ROWNUM:

=========

- To generate row numbers to each row wise (or) to each group of rows wise on a table automatically.
- These row numbers are not saved in database so that these are temporary numbers.
- To find out "Nth" position row and also "TOP n" rows from a table.

EX:

```
SQL> SELECT ROWNUM,ENAME,DEPTNO FROM EMP;
SQL> SELECT ROWNUM,ENAME,DEPTNO FROM EMP WHERE DEPTNO=10;
```

Ex:

waq to fetch the 1st row from emp table by using ROWNUM?

```
SQL> SELECT * FROM EMP WHERE ROWNUM=1;
```

Ex:

waq to fetch the 2nd row 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 the above problem we must use the following operators are " < , <= " along with MINUS operator.

Solution:

========

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

Ex:

waq to fetch 10th row from emp table by using ROWNUM?

```
SQL> SELECT * FROM EMP WHERE ROWNUM<=10
  2  MINUS
  3  SELECT * FROM EMP WHERE ROWNUM<=9;
```

Ex:

waq to fetch from 3rd row to 9th row from emp table by using ROWNUM?

```
SQL> SELECT * FROM EMP WHERE ROWNUM<=9
  2  MINUS
  3  SELECT * FROM EMP WHERE ROWNUM<3;
```

Ex:
waq to fetch the last two rows from a table by using ROWNUM?

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

EX:
waq to fetch top 5 rows from a table by using ROWNUM?

```
SQL> SELECT * FROM EMP WHERE ROWNUM<=5;
```

==========THE END-==========

ID : sudhakarsqldvp@gmail.com