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 filed user is interacting with 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 directly.

    Ex: Resgister form,Login form,View profile form,Home page,.....etc

Design & Develop:

==========

- By using UI technologies.

Ex: Html,Css,Javascript,AngularJS,React,Jquery,Json,.....etc

2. Back End Application:

==========

- BEA is an application where we store the end users data / information.

    Ex: Databases application

Design & Develop:

==========

- By using DB technologies.

Ex: Oracle,MS-Sqlserver,Mysql,PostgreSQL,DB2,Sybase,MaxDB,.....etc

Server Side Technologies:

==========

- These technologies are used to establish connection in between front end application to back end application.

Ex: Java,.Net,Python,Php,.....etc

==========

Topic-1 : DBMS

==========

What is Data?

==========

- it is a rawfact.(i.e characters,numbers,special characters,symbols)

- data never give meaningfull statements.

Ex:

10021	**ADAMS**
10022	**SMITH**
10023	**MILLER**

What is Information ?

==========

- processing data is called as information.

- it always provide meaningfull statements.

Ex:

Customer_ID	Customer_Name
==========	==========

10021 ADAMS

10022 SMITH

10023 MILLER

What is Database?

==========

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

What is inter-related data/information:

==========

- it means depending on each other.

Ex:

No departments = No employees

No employees = No departments

Ex:

No customers = No products

No products = No customers

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,Sybase,MaxDB,Informix,Ingrees..etc

2. OLAP:

=======

- these databases are used for storing "historical " data / information.

Ex: Datawarehouse

What is DBMS?

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

- it is a software which is used to manage and maintain data in database.

- by using DBMS we can perform the following operations are,

> Creating databases

> Designing Tables

> Storing data

> Manipulating data

> Retrieve / Reading data

> Deleting data

- it will work as an interface between User and Database.

USER <--------> DBMS s/w <---------> DATABASE

Models of DBMS:

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

- DBMS s/w again classified into 3 types those are,

i) Hierarchical database management system(HDBMS)

ex: IMS s/w (information management system)

ii) Network database management system(NDBMS)

ex: IDBMS s/w (integrated database management system)

NOTE: HDBMS,NDBMS are out dated models in real time.

iii) Relational Database Management System(RDBMS):

==========

- RDBMS model again divided into two modules.
    1) Object relational database management system(ORDBMS)
    2) Object Oriented database management system(OODBMS)

1) ORDBMS:

==========

- Here data can be stored in the form of "Table".
    a Table = collection of rows and columns.
    a row = group of columns.
- a row can also called as "records / tuples".
- a column can also called as "fields / attributes".
- these databases are completly depends on "SQL" so that these databases are called as "SQL Databases" in real world.

Ex: oracle,sqlserver,mysql,postgresql,maxdb,teradata,db2,sybase,informix,ingrees.

2) OODBMS:

==========

- Here data can be organized in the form of "Objects".
- these databases are not depends on "SQL" but depends on "OOPS" concept. so that these databases are called as "No SQL Databases" in real world.

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

==========

Topic-2 : ORACLE

==========

Introduction to Oracle:

==========

- it is a DB software / RDBMS product / ORDBMS module / Backend tool.
- it was introduced by "oracle corporation" in 1979.
- the first version of oracle s/w is "oracle1.0".
- it is used to store data / information permanently along with security.

Versions of oracle:

==========

- the following versions are,

- Oracle 1.0

- Oracle 2.0

- Oracle 3.0

- Oracle 4.0

- Oracle 5.0

- Oracle 6.0

- Oracle 7.0

- Oracle 8.0

- Oracle 8i (internet)

- Oracle 9i

- Oracle 10g (grid technologies)

- Oracle 11g

- Oracle 12c (cloud technologies)

- Oracle 18c

- Oracle 19c(latest version)

- Oracle 21c(very very latest version)

- Oracle 23c (Beta version)

Types of oracle s/w editions:

==========

- oracle s/w is available in two editions in real time.

1. Oracle express edition:

==========

- supporting partial features of oracle.

Ex: Recyclebin,Flashback,Purge,Partition tables,... are not allowed.

2. Oracle enterprise edition:

==========

- supporting all features if oracle.

Ex: Recyclebin,Flashback,Purge,Partition tables,... are allowed.

How to download oracle19c enterprise edition:

==========

https://www.oracle.com/in/database/technologies/oracle19c-windows-downloads.html

How to installing oracle19c enterprise edition s/w:

==========

- installing oracle s/w

NOTE:

======

- Once we installed oracle s/w internally there are two components are installed in the system.

1. Client component

2. Server component

1. Client component:

==========

- by using client component we will perform the following three operations,

Step1: User can send request to oracle server.

Request : SQL query / SQL command

Step2: Processing Client request.

Step3: User will get response from oracle server.

Response : Result / Output

Ex: SQLPlus,SQL developer,Toad.

2. Server component:

==========

- server component is having two more sub-components those are,

i) Instance

ii) Database

i) Instance:

==========

- it is a temporary memory which will allocate from RAM.

- data can be stored temporarly.

ii) Database:

==========

- it is a permanent memory which will allocate from Harddisk.

- data can be stored permanently.

NOTE:

=====

- when we want to work on oracle database server then we need to follow the following two steps are,

step1: Connect

step2: Communicate

How to connect to oracle server:

==========

> go to all programs

> go to oracle-oraDB19home1 folder

> click on SQLPLUS icon

Enter username : SYSTEM (default username)

Enter password : TIGER (created at installation of oracle s/w)

connected.

How to create a new username & password in oracle:

==========

syntax:

=======

CREATE USER <username> IDENTIFIED BY <password>;

EX:

- Generally username & password is creating by DBA in real time.

- Here our admin name is "SYSTEM".

> go to open SQLplus

Enter username : SYSTEM

Enter password : TIGER

connected.

SQL> CREATE USER MYDB9AM IDENTIFIED BY 123;

user created.

SQL> CONN

Enter user-name: MYDB9AM

Enter password: 123

ERROR:

ORA-01045: user MYDB9AM lacks CREATE SESSION privilege; logon denied

Granting all permissions to user:

==========

syntax:

=======

GRANT <privilege name> TO <username>;

EX:

SQL> CONN

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:

==========

syntax:

=======

PASSWORD;

EX:

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:

> go to open SQLPLUS

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 SYSTEM IDENTIFIED BY <new password>;

EX:

> go to open SQLPLUS

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;

EX:

SQL> CONN

Enter user-name: SYSTEM/LION

Connected.

SQL> SELECT USERNAME FROM ALL_USERS;

How delete / drop a user from oracle server:

==========

syntax:

======

DROP USER <username> CASCADE;

EX:

SQL> CONN

Enter username : SYSTEM/LION

connected.

SQL> DROP USER MYDB9AM CASCADE;

User dropped.

How to clear the screen:

==========

syntax:

======

SQL> CL SCR;

How to disconnect from oracle server:

==========

syntax:

======

SQL> 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" later renamed as "SQL" by IBM.**
- **SQL queries are not a case-sensitive i.e we will write sql queries in either upper / lower / combination of upper and lower case characters.**

  Ex:

  SQL> SELECT * FROME EMP; --------> executed

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

  SQL> SelecT * From Emp; ----------> executed

- **In oracle database data is a case-sensitive.**
- **Every sql query should ends with ";".**

Sub-Lanaguages of SQL:

==========

- **there are five sub-languages of sql.**

1) Data Definition Language(DDL):

==========

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

New features:

==========

- **RECYCLEBIN**
- **FLASHBACK**
- **PURGE**

2) Data Manipulation Language(DML):

==========

- **INSERT**
- **UPDATE**
- **DELETE**

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

==========

- **SELECT**

4) Transaction Control Language(TCL):

==========

- **COMMIT**

- ROLLBACK

- SAVEPOINT

5) Data Control Language(DCL):

==========

- GRANT

- REVOKE

==========

How to create a new table in oracle database server:

==========

CREATE:

=======

- to create a new table in database.

syntax:

=======

CREATE TABLE <table name>(<column name1> <datatype>[size],.....);

DATATYPES IN ORACLE:

==========

- it is an attribute to 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 (large objects)

i) Number datatype:

==========

NUMBER(P,S):

==========

- to store integer values & float expressions.

i) Number(p) : it will store integer values only.

ii) Number(p,s) : it will store float expressions.

Precision(p):

==========

- counting all digits including left and right sides of the given expression.

- the maximum size is 38 digits.

Ex:

i) 452

precision - 3

ii) 452.67

precision - 5

Ex:

SNO number(6)

==========

0

1

2

999999

1000000---------error

Scale(s):

=========

- counting the right side digits from the given float expression.

- there is no maximum size of scale because scale is a part of precision.

Ex:

i) 45.26

precision - 4

scale - 2

ii) 78634.685

precision - 8

scale - 3

Ex:

PRICE number(8,2)

==========

0.0

34.23

.

.

.

999999.99

1000000(1000000.00)

ii) Character / String datatypes:

==========

- storing string format data only.

- in database string can be represent with '<string>'.

EX: ENAME char(10)

==========

smith -------> error

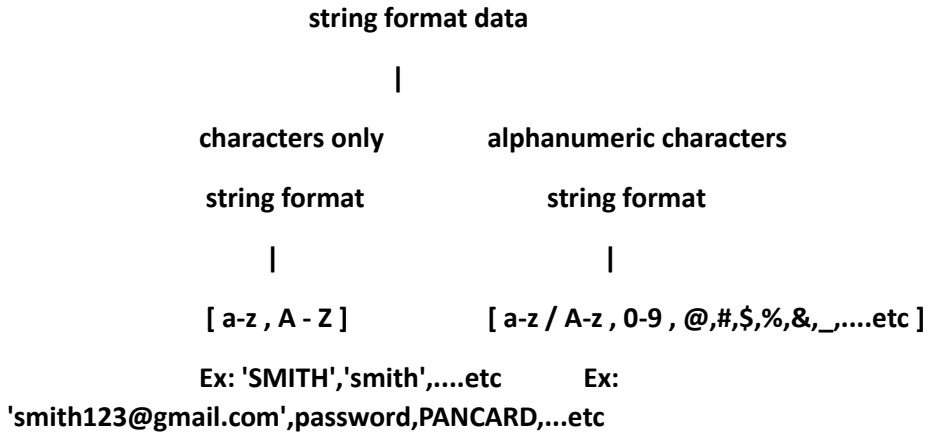
'smith'------> smith

1234 --------> error

'1234'-------> 1234

34.12 -------> error

'34.12'------> 34.12



- string datatypes are again classified into two types.

1. Non-unicode datatypes

==========

- these are storing "localized data" (i.e English language only)

i) char(size)

ii) varchar2(size)

2. Unicode datatypes:

==========

- these are storing "globalized data" (i.e all national languages)

i) Nchar(size)

ii) Nvarchar2(size)

i) char(size):

==========

- it is a fixed length datatype(static).

- it will store non-unicode characters in the form of 1 byte = 1 char.

- the maximum size is 2000 bytes.

Disadvantage:

==========

- memory wasted.

ii) varchar2(size):

==========

- it is a variable length datatype(dynamic).

- it will store non-unicode characters in the form of 1 byte = 1 char.

- the maximum size is 4000 bytes.

Advantage:

==========

- memory saved.

i) Nchar(size):

==========

- it is a fixed length datatype(static).

- it will store unicode characters in the form of 1 byte = 1 char.

- the maximum size is 2000 bytes.

Disadvantage:

==========

- memory wasted.

ii) Nvarchar2(size):

==========

- it is a variable length datatype(dynamic).

- it will store unicode characters in the form of 1 byte = 1 char.

- the maximum size is 4000 bytes.

Advantage:

==========

- memory saved.

Long datatype:

==========

- it is a variable length datatype(dynamic).

- it will store non-unicode & unicode characters in the form of 1 byte = 1 char.

- the maximum size is 2 gb.

Date datatypes:

==========

- storing date & time information of a particular day.

- the range of date datatypes are from'01-jan-4712BC' to '31-dec-9999 AD'.

- date and time should enclosed with ' <date time> '.

- default date format of oracle database is 'DD-MON-YY/YYYY HH:MI:SS'.

i) DATE

ii) TIMESTAMP

i) DATE:

=======

- storing date & time information of a particular day whereas time is optional.

- if user not enter time then oracle server will take '12:00:00 / 00:00:00 am'.

- it will occupie 7 bytes of memory and it is a fixed memory.

' DD-MON-YY HH:MI:SS '

1 1 2 1 1 1 --------> 7 bytes

ii) TIMESTAMP:

==========

- storing date & time information including milliseconds.

- it will occupie 11 bytes of memory and it is a fixed memory.

' DD-MON-YY HH:MI:SS.MS '

1 1 2 1 1 1 4 --------> 11 bytes

Raw & Long Raw datatypes:

==========

- storing image file / audio file / video file in the form of 01010010101001 binary format.

Raw :

=====

- it is a static datatype.

- maximum size is 2000 bytes.

Long Raw:

=========

- it is a dynamic datatype.

- maximum size is 2gb.

LOB datatypes:

==========

i) CLOB

ii) NCLOB

iii) BLOB

i) CLOB:

=======

- it stands for "character large object" datatype.

- it will store non-unicode characters in the form of 1 char = 1 byte.

- it is a dynamic datatype.

- the maximum size is 4gb.

ii) NCLOB

=========

- **it stands for "national character large object" datatype.**
- **it will store unicode characters in the form of 1 char = 1 byte.**
- **it is a dynamic datatype.**
- **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 0101010101 binary format.**
- **the maximum size 4 gb.**

Non-unicode characters:

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

- **Char(size) - static - 2000 bytes**
- **Varchar2(size) - dynamic - 4000 bytes**
- **Long - dynamic - 2gb**
- **Clob - dynamic - 4gb**

Unicode characters:

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

- **NChar(size) - static - 2000 bytes**
- **NVarchar2(size) - dynamic - 4000 bytes**
- **Long - dynamic - 2gb**
- **NClob - dynamic - 4gb**

Binary data:

==========

- Raw - static - 2000 bytes

- Long Raw - dynamic - 2gb

- Blob - dynamic - 4gb

==========

How to create a new table in oracle database:

==========

CREATE:

========

- to create a new table in database.

syntax:

======

CREATE TABLE <TABLE NAME>(<column name1> <datatype>[size],.....);

Ex:

> go to open SQLPLUS

Enter user-name: SYSTEM/TIGER

connected.

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

SQL> GRANT DBA TO MYDB9AM;
Grant succeeded.

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

SQL> CREATE TABLE STUDENT(STID NUMBER(4),SNAME CHAR(5),SFEE NUMBER(6,2));
Table created.
```

To view the structure of a table in oracle:

==========

syntax:

=======

DESC <table name> ; [Describe command]

EX:

SQL> DESC STUDENT;

ALTER:

=======

- **to change the structure of a table.**
- **the sub-commands of alter command.**
  - **i) ALTER - MODIFY**
  - **ii) ALTER - ADD**
  - **iii) ALTER - RENAME**
  - **iv) ALTER - DROP**

i) ALTER - MODIFY:

==========

- **to change datatype of specific column and also to change the size of datatype of a specific column in the table.**

syntax:

=======

ALTER TABLE <TN> MODIFY <COLUMN NAME> <NEW DATATYPE>[NEW SIZE];

EX:

SQL> ALTER TABLE STUDENT MODIFY SNAME VARCHAR2(10);

ii) ALTER - ADD:

==========

- to add a new column to an existing table.

syntax:

=======

ALTER TABLE <TN> ADD <NEW COLUMN NAME> <DATATYPE>[SIZE];

EX:

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

iii) ALTER - RENAME:

==========

- to change a column name in the table.

syntax:

======

ALTER TABLE <TN> RENAME <COLUMN> <OLD COLUMN NAME> TO <NEW COLUMN NAME>;

EX:

SQL> ALTER TABLE STUDENT RENAME COLUMN SNAME TO STUDENTNAME;

iv) ALTER - DROP:

==========

- to delete / drop a specific column from a table.

syntax:

======

ALTER TABLE <TN> DROP <COLUMN> <COLUMN NAME>;

EX:

SQL> ALTER TABLE STUDENT DROP COLUMN SFEE;

RENAME:

========

- to change a table name.

syntax:

=======

RENAME <OLD TABLE NAME> TO <NEW TABLE NAME>;

EX:

SQL> RENAME STUDENT TO STUDENT_DETAILS;

SQL> RENAME STUDENT_DETAILS TO STUDENT;

TRUNCATE:

==========

- deleting rows but not columns in a table.

- by using truncate command we cannot delete a specific row from a table because truncate command is not support "WHERE" clause condition.

- deleting rows from a table permanently.

syntax:

======

TRUNCATE TABLE <TABLE NAME>;

EX:

SQL> TRUNCATE TABLE STUDENT;

DROP:

=====

- to delete a table from database.(i.e collection rows & columns)

syntax:

=======

DROP TABLE <TABLE NAME>;

EX:

SQL> DROP TABLE STUDENT;

NOTE:

======

- Before oracle10g enterprise edition once we drop a table from database then it was dropped permanently whereas from oracle10g enterprise edition once we drop a table from database then it was dropped temporarly.

New Features in Oracle10g enterprise edition:

==========

i) RECYCLEBIN:

==========

- it is a similar to windows recyclebin in the computer.

- it is used to store the information about deleted tables from database.

How to view deleted tables in recyclebin:

==========

syntax:

=======

SELECT OBJECT_NAME,ORIGINAL_NAME FROM RECYCLEBIN;

OBJECT_NAME	ORIGINAL_NAME
BIN$Law0yN2RR5GT/JDFEGSvSw==$0	STUDENT

ii) FLASHBACK:

==========

- it is used to restore a deleted table from recyclebin to database memory.

syntax:

======

FLASHBACK TABLE <TABLE NAME> TO BEFORE DROP;

EX:

SQL> FLASHBACK TABLE STUDENT TO BEFORE DROP;

iii) PURGE:

=========

- it is used to delete a table permanently.

Case-1: Deleting a table from recyclebin permanently:

==

syntax:

======

PURGE TABLE <TABLE NAME>;

EX:

PURGE TABLE TEST1;

Case-2 : Deleting all tables from recyclebin permanently:

==

syntax:

=======

PURGE RECYCLEBIN;

EX:

SQL> PURGE RECYCLEBIN;

Case-3: Deleting 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) but not in "ADMIN" (system).

Data Manipulate Language(DML):

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

INSERT :

========

- to insert a new row data into a table.

syntax-1:

=======

INSERT INTO <TABLE NAME> VALUES(value1,value2,.....);

Ex:

SQL> CREATE TABLE PRODUCT(PCODE NUMBER(4),PNAME VARCHAR2(10),PRICE NUMBER(6,2));

SQL> INSERT INTO PRODUCT VALUES(1021,'P1',3400.45);

1 row created.

syntax-2:

========

INSERT INTO <TABLE NAME>(llst of columns)VALUES(value1,value2,.....);

EX:

SQL> INSERT INTO PRODUCT(PCODE,PNAME,PRICE)VALUES(1022,'P2',5600.34);

SQL> INSERT INTO PRODUCT(PCODE,PNAME)VALUES(1023,'P3');

SQL> INSERT INTO PRODUCT(PCODE)VALUES(1024);

How to insert multiple rows into a table dynamically(i.e at Runtime):

==========

syntax-1:

========

INSERT INTO <TN> VALUES(&<column name1>,&<column name2>,.....);

Ex:

SQL> INSERT INTO PRODUCT VALUES(&PCODE,'&PNAME',&PRICE);

Enter value for pcode: 1025

Enter value for pname: P5

Enter value for price: 6700.24

1 row created.

SQL> / (re-execute the last executed sql query in sqlplus editor)

Enter value for pcode: 1026

Enter value for pname: P6

Enter value for price: 6734.12

1 row created.

SQL> /

.....

.....

.....

syntax-2:

========

INSERT INTO <TN>(list of columns)VALUES(&<column name1>,.....);

EX:

SQL> INSERT INTO PRODUCT(PCODE)VALUES(&PCODE);

Enter value for pcode: 1028

SQL> /

Enter value for pcode: 1029

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 <TN> SET <column name1>=<value1>,<column name2>=<value2>,.....[WHERE <CONDITION>];

EX:

SQL> UPDATE PRODUCT SET PNAME='P4',PRICE=35.12 WHERE PCODE=1024;

SQL> UPDATE PRODUCT SET PCODE=NULL,PNAME=NULL,PRICE=NULL WHERE PCODE=1026;

SQL> UPDATE PRODUCT SET PCODE=1026,PNAME='P6',PRICE=567.45 WHERE PCODE IS NULL;

SQL> UPDATE PRODUCT SET PRICE=NULL;

SQL> UPDATE PRODUCT SET PRICE=5000.25;

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 PRODUCT WHERE PNAME='P4';

SQL> DELETE FROM PRODUCT WHERE PCODE=1029;

SQL> DELETE FROM PRODUCT WHERE PNAME IS NULL;

SQL> DELETE FROM PRODUCT;

DELETE vs TRUNCATE

==========

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

Data Query / Retrieval Language(DQL/DRL):

==========

SELECT:

========

- to retrieve a specific row from a table by using "WHERE" clause condition.

(or)

- to retrieve all rows from a table at a time.

syntax:

======

SELECT * / <list of columns> FROM <TABLE NAME>;

EX:

SQL> SELECT * FROM EMP;

SQL> SELECT * FROM EMP WHERE EMPNO=7788;

SQL> SELECT * FROM EMP WHERE COMM IS NULL;

ALIAS NAMES:

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

- it is a temporary name for columns and also table.

- we can create alias names at two levels.

i) column level alias names:

==========

- creating alias name on column.

ii) table level alias names:

==========

- creating alias name on table.

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;

CONCATENATION OPERATOR(||):

==========

- to add two or more than two expressions.

syntax:

=======

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

EX:

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

DISTINCT keyword:

==========

- to eliminate duplicate values from a specific column.

syntax:

======

distinct <column name>

EX:

SQL> SELECT DISTINCT JOB FROM EMP;

SQL> SELECT DISTINCT DEPTNO FROM EMP ORDER BY DEPTNO;

NOTE:

=====

- when we want to display data in proper systematically in sqlplus editor then we need to set the following two properties are,

i) pagesize n

ii) lines n

i) pagesize n:

==========

- in sqlplus editor a single page will display 14 rows by default.

- to display more than 14 rows in a single page then we set "pagesize n" property. here "n" is no.of rows in a page.

- the maximum size of pagesize property is 50000 rows.

syntax:

======

```
SET PAGESIZE n;
```

EX:

```
SQL> SET PAGESIZE 100;
```

ii) lines n:

========

- by default each line contains 80 bytes(1 char = 1 byte).

- to increse / decrese the line size then we set "lines n" property. here "n" is no.of bytes.

- maximum size of lines property is 32767 bytes.

syntax:

=======

```
SET LINES n;
```

Ex:

SQL> SET LINES 160;

==========

Operators in oracle sql:

==========

- to perform some operation as per the given operand values.

- oracle supports the following operators are,

i) Assignment operator	**=>**	**=**	
ii) Arithmetic operators	**=>**	**+ , - , * , /**	
iii) Relational operators		**=>**	**< , > , <= , >= , != (or) <>**
iv) Logical operators	**=>**	**AND,OR,NOT**	
v) Set operators		**=>**	**UNION,UNION ALL,INTERSECT,MINUS.**
vi) Special operators	**=>**	**(+ve)**	**(-ve)**
		=====	**====**
		IN	**NOT IN**
		BETWEEN	**NOT BETWEEN**
		IS NULL	**IS NOT NULL**
		LIKE	**NOT LIKE**

i) Assignment operator:

==========

- to assign a value to variable (or) attribute.

syntax:

======

<column name> <assignment operator> <value>

EX:

SQL> UPDATE EMP SET SAL=25000;

SQL> UPDATE EMP SET JOB='MANAGER' WHERE EMPNO=7788;

ii) Arithmetic operators:

==========

- to perform addition,subtraction,multiple and division.

syntax:

======

<column name> <arithmetic operator> <value>

Ex:

waq to display employees details after adding 1000 /- to a salary?

```
SQL> SELECT ENAME,SAL AS OLD_SALARY,SAL+1000 AS NEW_SALARY FROM EMP;
```

Ex:

waq to display EMPNO,ENAME,BASIC SALARY and ANNUAL SALARY of the employees who are working under deptno is 20?

```
SQL> SELECT EMPNO,ENAME,SAL AS BASIC_SALARY,SAL*12 AS ANNUAL_SALARY
     FROM EMP WHERE DEPTNO=20;
```

Ex:

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

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

Ex:

waq to display EMPNO,ENAME,BASIC SALARY,INCREMENT OF 5% AMOUNT,TOTAL SALARY FROME EMP TABLE WHO ARE WORKING AS A "MANAGER"?

```
SQL> SELECT EMPNO,ENAME,SAL AS BASIC_SALARY,
  2  SAL*0.05 AS INCREMENT_AMOUNT,
  3  SAL+SAL*0.05 AS TOTAL_SALARY
  4  FROM EMP WHERE JOB='MANAGER';
```

EX:

waq to display EMPNO,ENAME,JOB,BASIC SALARY,HRA with 10%,DA with 20%,PF with 5% and find GROSS SALARY of the employees who are working as a "SALESMAN" ?

```
SQL> SELECT EMPNO,ENAME,JOB,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  FROM EMP WHERE JOB='SALESMAN';
```

EX:

waq to display all employees salaries after decrement of 5%?

```
SQL> SELECT ENAME,SAL AS BEFORE_DECREMENT,SAL-SAL*0.05 AS AFTER_DECREMENT
  2  FROM EMP;
```

iii) 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 employees who are joined before 1981?

```
SQL> SELECT * FROM EMP WHERE HIREDATE<'01-JAN-1981';
```

EX:

waq to display employees who are joined after 1981?

```
SQL> SELECT * FROM EMP WHERE HIREDATE>'31-DEC-1981';
```

iv) Logical operators:

==========

- to check more than one condition in the query.

- AND,OR,NOT operator.

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 whose name is "SMITH" and working as a "CLERK" ?

```
SQL> SELECT * FROM EMP WHERE ENAME='SMITH' AND JOB='CLERK';
```

OR operator:

==========

- it return a value if any one condition is true from the given group of conditions in the query.

cond1	cond2

T T ===> T

T F ===> T

F T ===> T

F F ===> F

syntax:

======

where <condition1> OR <condition2>

Ex:

waq to display employees whose empno is 7369,7566,7788?

SQL> SELECT * FROM EMP WHERE EMPNO=7369 OR EMPNO=7566 OR EMPNO=7788;

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 under deptno is 10,20?

SQL> SELECT * FROM EMP WHERE NOT DEPTNO=10 AND NOT DEPTNO=20;

v) 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}

UNION:

======

- to combined two sets of values without duplicate values.

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

UNION ALL:

==========

- to combined two sets of values with duplicate values.

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

INTERSECT:

===========

- it return common values from both sets.

A I B = { 30 }

MINUS:

======

- it return uncommon values from the left set only.

A-B = { 10,20 }

B-A = { 40,50 }

DEMO_TABLES:

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

SQL> SELECT * FROM EMP_HYD;

EID	ENAME	SAL
----------	----------	----------

1021 SMITH 85000

1022 ALLEN 45000

SQL> SELECT * FROM EMP_MUMBAI;

EID	ENAME	SAL
1021	SMITH	85000
1023	JONES	56000
1024	WARD	30000

EX:

waq to display all employees details who are working in the organization?

```
SQL> SELECT * FROM EMP_HYD
  2  UNION ALL
  3  SELECT * FROM EMP_MUMBAI; ---------> (including duplicate rows)
```

(OR)

```
SQL> SELECT * FROM EMP_HYD
  2  UNION
  3  SELECT * FROM EMP_MUMBAI; ---------> (excluding duplicate rows)
```

EX:

waq to display employees who are working in HYD but not in MUMBAI branch?

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

EX:

waq to display employees who are working in both branches?

```
SQL> SELECT * FROM EMP_HYD
  2  INTERSECT
```

3 SELECT * FROM EMP_MUMBAI;

NOTE:

======

1. no.of columns in both select queries should be same.

2. order of the columns and datatypes of columns must be match in both select queries.

vi) Special operators:

==========

IN operator:

==========

- comparing group of values with a single condition.

syntax:

======

where <column name> IN(value1,value2,.....);

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

Ex:

waq to display the list of employees whose empno is 7566,7788,7900?

SQL> SELECT * FROM EMP WHERE EMPNO IN(7566,7788,7900);

Ex:

waq to display the list of 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 display employees who are joined in 1981?

SQL> SELECT * FROM EMP WHERE HIREDATE BETWEEN '01-JAN-1981' AND '31-DEC-1981';

Ex:

waq to display employees who are not joined in 1981?

SQL> SELECT * FROM EMP WHERE HIREDATE NOT BETWEEN '01-JAN-1981' AND '31-DEC-1981';

IS NULL:

=======

- 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 empty / undefined ?

SQL> SELECT * FROM EMP WHERE COMM IS NULL;

Ex:

waq to display employees whose commission is empty / undefined ?

SQL> SELECT * FROM EMP WHERE COMM IS NOT NULL;

LIKE operator:

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

- comparing a specific character pattern wise.

- when we use like operator we must use the following

wildcard operators are,

i) % - it represent the remaining group of char's after selected

- character.

ii) _ - counting a single character from the given string 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:

waq to fetch employees whose name ends with "R" character?

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

Ex:

waq to fetch employees whose name starts with "M" and ends with "N" character?

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

Ex:

waq to fetch employees whose name is having "I" character?

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

Ex:

waq to fetch employees whose name is having 4 char's?

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

Ex:

waq to fetch employees whose name contains the 2nd character is "O" ?

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

Ex:

waq to fetch employees who are joined in 1981?

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

Ex:

waq to fetch employees who are joined in the month of "DECEMBER"?

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

Ex:

waq to fetch employees who are joined in the month of "DECEMBER" in 1981?

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

Ex:

waq to fetch employees who are joined in the month of "JUNE","DECEMBER"?

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

LIKE operator with Special characters:

==========

DEMO_TABLE:

==========

SQL> SELECT * FROM TEST;

ENAME	SAL
_SMITH	12000
MILL#ER	23000
WAR_NER	34000
ADAMS%	42000
SCO%TT	51000
JONE@S	64000

EX:

waq fetch employees whose name is having "@" symbol?

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

EX:

waq fetch employees whose name is having "#" symbol?

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

EX:

waq fetch employees whose name is having "_" symbol?

SQL> SELECT * FROM TEST WHERE ENAME LIKE '%_%'; -------> wrong result

EX:

waq fetch employees whose name is having "%" symbol?

SQL> SELECT * FROM TEST WHERE ENAME LIKE '%%%'; ------> wrong result

- by default "_ , % " are treat as wildcard operators by oracle.if we need to treat as special characters then we should use a pre-defined keyword is " ESCAPE '\' " in oracle.

solution:

=======

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

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

EX:

waq to fetch employees 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 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 are also called as "Built-In-Functions" in oracle database.
- these are again two types.
    i) Single row functions (Scalar Functions)
    ii) Multiple row functions (Grouping functions / Aggregative functions)

i) Single row functions:

==========

- these functions are return a single value.
    > Numeric functions
    > Character / String functions
    > Date functions
    > Null functions
    > Analytical functions
    > Conversion functions

syntax to call a Function:

==========

```
SELECT <FNAME>(value / values) FROM DUAL;
```

What is DUAL?

==========

- **it is a system defined table in oracle.**
- **it contains a single row and a single column.**
- **it is used to test function functionalities(i.e workflow)**

How to view the structure of DUAL table:

==========

syntax:

=======

DESC DUAL;

Name	Null?	Type
DUMMY		VARCHAR2(1)

How to view data in DUAL table:

==========

syntax:

======

SELECT * FROM DUAL;

D ----------> single column

-

X ----------> single row

CHARACTER / STRING FUNCTIONS:

==========

LENGTH():

=========

- it return the length of the given string expression.

syntax:

======

length(string)

Ex:

SQL> SELECT LENGTH('HELLO') FROM DUAL;----------> 5

SQL> SELECT LENGTH('WEL COME') FROM DUAL;------> 8

SQL> SELECT ENAME,LENGTH(ENAME) FROM EMP;

SQL> SELECT * FROM EMP WHERE LENGTH(ENAME)=6;

LOWER():

=======

- to convert upper case characters into lower case characters.

syntax:

======

lower(string)

EX:

SQL> SELECT LOWER('HELLO') FROM DUAL;

hello

SQL> SELECT ENAME,LOWER(ENAME) FROM EMP;

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

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

LTRIM():

=======

- it remove unwanted characters from the left side of the given expression.

syntax:

======

ltrim(string,<trimming characters>)

Ex:

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

SMITH

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

SMITH

RTRIM():

=======

- it remove unwanted characters from the right side of the given expression.

syntax:

======

rtrim(string,<trimming characters>)

Ex:

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

SMITH

TRIM():

======

- it remove unwanted chracters from both sides of the given string expression.

syntax:

======

trim('trimming character' from STRING)

EX:

SQL> SELECT TRIM('XY' FROM 'XYSMITHXY') FROM DUAL;

ERROR at line 1:

ORA-30001: trim set should have only one character

SQL> SELECT TRIM('X' FROM 'XXXSMITHXXXXX') FROM DUAL;

SMITH

CONCAT():

========

- to add two string expressions.

syntax:

======

concat(string1,string2)

EX:

SQL> SELECT CONCAT('HAI','HELLO') FROM DUAL;

HAIHELLO

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

INITCAP():

========

- to convert the initial character is capital.

syntax:

======

initcap(string)

EX:

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

Smith

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

Smith

SQL> SELECT ENAME,INITCAP(ENAME) FROM EMP;

REPLACE():

==========

- to replace string to string / string to characters / chracter to string.

syntax:

=======

replace(string,<old char's>,<new char's>)

Ex:

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

HABCDO

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

HEXYZXYZO

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

HX

TRANSLATE():

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

- to translate a single character by a single character.

syntax:

=======

translate(string,'old character','new character')

EX:

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

HXYYZ

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

HXYY

SUBSTR():

=========

- it return the required sub string from the given string expression.

syntax:

=======

substr(string,<starting position of character>,<length of the char's>)

EX: -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',1,2) FROM DUAL;

WE

SQL> SELECT SUBSTR('WELCOME',1,4) FROM DUAL;

WELC

SQL> SELECT SUBSTR('WELCOME',5,4) FROM DUAL;

OME

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

CO

SQL> SELECT SUBSTR('WELCOME',-6,1) FROM DUAL;

E

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

- Here, length should not be (-ve) sign.

Date functions:

==========

SYSDATE:

========

- it returns current date information of the system.

syntax:

======

sysdate

Ex:

SQL> SELECT SYSDATE FROM DUAL;

SQL> SELECT SYSDATE+10 FROM DUAL;

SQL> SELECT SYSDATE-10 FROM DUAL;

ADD_MONTHS():

==========

- to add / subtract no.of months from the given date / to the given date.

syntax:

======

add_months(date,<no.of months>)

EX:

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

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

LAST_DAY():

==========

- it return the last day from the given month in the date expression.

syntax:

=======

last_day(date)

Ex:

SQL> SELECT LAST_DAY(SYSDATE) FROM DUAL;

SQL> SELECT LAST_DAY('23-APR-2024') FROM DUAL;

MONTHS_BETWEEN:

==========

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

syntax:

=======

months_between(date1,date2)

NOTE:

=====

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

Ex:

SQL> SELECT MONTHS_BETWEEN('05-MAR-2023','05-MAR-2024') FROM DUAL;----> -12

SQL> SELECT MONTHS_BETWEEN('05-MAR-2024','05-MAR-2023') FROM DUAL;-----> 12

Null functions:

==========

i) NVL(exp1,exp2)

ii) NVL2(exp1,exp2,exp3)

What is NULL?

==========

- NULL is an empty / unknown value / undefined value in database.
- NULL != 0 and also NULL != space.
- If any arithmetic operator is performing some operation with NULL then it again return NULL only.

Ex:

> IF X=1000;

i) x+null ===> 1000+null ===> null

ii) x-null ===> 1000-null ====> null

iii) x*null ==> 1000*null ====> null

iv) x / null ==> 1000/null ====> null

Ex:

waq to display EMPNO,ENAME,SALARY,COMMISSION and SAL+COMM from emp table whose employee name is "SMITH"?

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

OUTPUT:

========

```
    EMPNO   ENAME   SAL      COMM  TOTAL_SALARY
     ----------    ----------     ----------    ----------  ------------
     7369     SMITH      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 provide some pre-defined functions are called as "Null functions",

i) NVL(exp1,exp2)

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

- it stands for NULL VALUE.

- NVL() is used to replace a user defined value inplace of NULL in the expression.

- this function is having two arguments those 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(0,100) FROM DUAL; ----------> 0
SQL> SELECT NVL(500,100) FROM DUAL;----------> 500
```

Solution:

========

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

OUTPUT:

========

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

ii) NVL2(exp1,exp2,exp3):

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

- **it is an extension of NVL().**
- **this function is having 3 arguments are expression1,expression2 and expression3.**
  - **> if exp1 is NULL then it return exp3 value(user deifned value).**
  - **> if exp1 is NOT NULL then it return exp2 value(user defined value).**

Ex:

```
SQL> SELECT NVL2(NULL,100,200) FROM DUAL;---------> 200
SQL> SELECT NVL2(0,100,200) FROM DUAL;----------------> 100
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 employees commission is NULL then update those employees commissions as 900.

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

```
SQL> UPDATE EMP SET COMM=NVL2(COMM,COMM+300,900);
```

Analytical functions:

==========

- **to assign rank numbers to each row wise / to each group of rows wise.**
  - **i) RANK()**
  - **ii) DENSE_RANK()**
- **these analytical functions are also called as " Ranking Functions" in database.**

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	42000	7	5
H	33000	8	6

syntax:

======

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

Here,

- **partition by clause ----- optional**
- **order by clause ---------- mandatory**

Without partition by clause:

==========

Ex:

SQL> SELECT 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;

With partition by clause:

==========

Ex:

SQL> SELECT ENAME,JOB,SAL,RANK()OVER(PARTITION BY JOB ORDER BY SAL DESC) AS RANKS FROM EMP;

SQL> SELECT ENAME,JOB,SAL,DENSE_RANK()OVER(PARTITION BY JOB 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 type and also display date in different format.

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,'YY YYYY YEAR CC BC') FROM DUAL;

OUTPUT:

----

24 2024 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:

----

08 AUG AUGUST

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:

--

235 22 5 THU THURSDAY

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:

3

Week Format:

WW - Week Of The Year

W - Week Of Month

EX:

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

OUTPUT

----

34 4

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,'HH24 HH MI SS PM') FROM DUAL;

OUTPUT:

----

10 10 33 39 AM

EX:

waq to display 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 employees who are joined in 1980,1982,1983?

SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'YY')IN('80','82','83');

EX:

waq to display employees who are joined in the month of DECEMBER?

SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'MON')='DEC';

EX:

waq to display employees who are joined in the month of DECEMBER in 1982?

SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'MON')='DEC' AND TO_CHAR(HIREDATE,'YYYY')='1982';

(OR)

SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'MMYYYY')='121982';

EX:

waq to display employees joined day?

SQL> SELECT ENAME,HIREDATE,TO_CHAR(HIREDATE,'DAY') AS JOINED_DAY FROM EMP;

EX:

waq to display employees who are joined on FRIDAY?

SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'DY')='FRI';

EX:

waq to display employees who are joined on WEEKEND?

SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'DY')IN('SAT','SUN');

EX:

waq to display employees who are joined in 2nd week of any year?

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

EX:

waq to display employees who are joined in 2nd week of DECEMBER month?

```
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'W')='2' AND
TO_CHAR(HIREDATE,'MON')='DEC';
```

(OR)

```
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'WMM')='212';
```

EX:

waq to display employees who are joined in 1st week of DECEMBER month in 1981 ?

```
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'WMMYYYY')='1121981';
```

EX:

waq to display employees who are joned 3rd quater of 1981?

```
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'Q')='3' AND
TO_CHAR(HIREDATE,'YYYY')='1981';
```

(OR)

```
SQL> SELECT * FROM EMP WHERE TO_CHAR(HIREDATE,'QYYYY')='31981';
```

ii) TO_DATE():

==========

- to convert character type to oracle default date type.

syntax:

=======

to_date(string)

Ex:

```
SQL> SELECT TO_DATE('23/AUGUST/2024') FROM DUAL;
```

23-AUG-24

```
SQL> SELECT TO_DATE('23/AUGUST/2024')+5 FROM DUAL;
```

28-AUG-24

SQL> SELECT TO_DATE('23/AUGUST/2024')-5 FROM DUAL;

18-AUG-24

ii) Multiple row functions:

==========

- these functions are also called as "grouping / aggregative" functions.

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 JOB='MANAGER';

MIN():

=====

- it return minimum value.

EX:

SQL> SELECT MIN(SAL) FROM EMP;

SQL> SELECT MIN(HIREDATE FROM EMP;

SQL> SELECT MAI(HIREDATE) FROM EMP WHERE DEPTNO=30;

MAX():

=====

- it return maximum value.

EX:

SQL> SELECT MAX(SAL) FROM EMP;

COUNT():

========

- it again classified into three types.

i) count(*):

=========

- counting all rows (i.e including duplicates and nulls) in a table .

EX:

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

ii) count(column name):

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

- counting all values including duplicates but not nulls from a column.

EX:

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

ii) count(DISTINCT column name):

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

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

EX:

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

==========

CLAUSES IN ORACLE:

==========

- It is a statement which is used to add to SQL QUERY for providing some additional facilities are FILTERING ROWS,SORTING VALUES and GROUPING similar data based on columns from a table automatically.

- Oracle supports the following clauses are,

  > WHERE

  > ORDER BY

  > GROUP BY

  > HAVING

syntax:

=======

<SQL QUERY> + <clause statement>;

WHERE:

=======

- filtering rows before grouping data in a table.

syntax:

=======

where <filtering condition>;

EX:

SQL> SELECT * FROM EMP WHERE EMPNO=7788;

SQL> UPDATE EMP SET SAL=56000 WHERE JOB='MANAGER';

SQL> DELETE FROM EMP WHERE DEPTNO=10;

ORDER BY:

==========

- **to arrange the values either in ascending or in descending order.**
- **by default order by clause will arrange values in ascending order.**
- **if we want to arrange the values in descending order then use "DESC" keyword.**

syntax:

=======

select * from <table name> order by <column name1> <asc/desc>,<column name2> <asc/desc>,.....;

EX:

SQL> SELECT * FROM EMP ORDER BY ENAME;

SQL> SELECT * FROM EMP ORDER BY ENAME DESC;

SQL> SELECT * FROM EMP ORDER BY SAL;

SQL> SELECT * FROM EMP ORDER BY SAL 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 deptno's in ascending order and those employees salaries in descending order from each deptno wise?

SQL> SELECT * FROM EMP ORDER BY DEPTNO,SAL DESC;

GROUP BY:

=========

- it is used to make groups based on a particular column wise.

- when we use "group by" clause we must use "grouping functions".

syntax:

======

select <column name1>,.....,<grouping function name1>,.....

from <table name> group by <column name1>,.....;

Ex:

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

QL> SELECT JOB,COUNT(*) AS NO_0F_EMPLOYEES FROM EMP GROUP BY JOB;

Ex:

waq to display no.of employees are working under each job wise along with their deptno?

SQL> SELECT JOB,DEPTNO,COUNT(*) AS NO_0F_EMPLOYEES FROM EMP

GROUP BY JOB,DEPTNO ORDER BY JOB,DEPTNO;

Using all aggregative functions along with Group by clause:

==========

EX:

```
SQL> SELECT DEPTNO,COUNT(*) AS NO_OF_EMPLOYEES,
  2  SUM(SAL) AS SUM_OF_SALARY,
  3  AVG(SAL) AS AVG_SALARY,
  4  MIN(SAL) AS MIN_SALARY,
  5  MAX(SAL) AS MAX_SALARY
  6  FROM EMP GROUP BY DEPTNO ORDER BY DEPTNO;
```

HAVING:

=======

- filtering rows after grouping data in the table.

- it should use after goup by claues in select statement.

syntax:

======

select <column name1>,.....,<grouping function name1>,.....

from <table name> group by <column name1>,.....having<filtering condition>;

Ex:

waq to display no.of employees working under a job from emp table in which job no.of employees are more than 3?

```
SQL> SELECT JOB,COUNT(*) AS NO_0F_EMPLOYEES FROM EMP
        GROUP BY JOB HAVING COUNT(*)>3;
```

Ex:

waq to display sum of salary of deptno from emp table if sum of salary of deptno is less than to 10000?

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

==========

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 use a technique is called as "JOINS".

- Joins are used retrieving data / information from multiple tables at a time.

- oracle supports the following types of joins,

1. Inner join

> Equi join

> Non-equi join

> Self join

2. Outer join

> Left outer join

> Right outer join

> Full outer join

3. Cross join

4. Natural join

syntax:

========

SELECT * FROM <TN1> <join key> <TN2> ON <JOINING CONDITION>;.....

1. Inner join :

===========

Equi join :

=========

- when we are retrieving data from multiple tables based on an " = " operator condition is known as "Equi join".

- when we use equi join we should maintain at least one commoness column in both tables and also their datatypes must be match.

- it always retrieving matching rows from multiple tables.

syntax for joining condition:

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

ON <table name1 / table alias name1>.<common column name> = <table name2/table alias name2>.<common column name>;

Ex:

waq to retrieve student and the corresponding course details from the tables?

SQL> SELECT * FROM STUDENT JOIN COURSE ON STUDENT.CID=COURSE.CID;

(OR)

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

RULE for JOINS:

==========

==========

|| a row in a table is comparing with all rows of another table ||

==========

EX:

waq to retrieve students and their corresponding course details from the tables who are selected "ORACLE" course?

```
SQL> SELECT * FROM STUDENT S JOIN COURSE C ON S.CID=C.CID AND CNAME='ORACLE';
```

(OR)

```
SQL> SELECT * FROM STUDENT S JOIN COURSE C ON S.CID=C.CID WHERE CNAME='ORACLE';
```

Ex:

waq to retrieve employees and the corresponding working location from tables.who are working in "CHICAGO" location?

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

EX:

waq to display sum of salaries of each department name wise from emp,dept tables?

EX:

waq to display DEPTNO,sum of salaries of each department name from emp,dept tables?

Ex:

waq to display sum of salaries of department names from emp,dept tables in which department sum of salary is less than 10000?

EX:

waq to display no.of employees are working under department name from emp,dept tables

in which department no.of employees are more than 3?

ii) NON-EQUI JOIN:

==========

- when we retrieve data from multiple tables based on any condition except an " = " operator.

DEMO_TABLES:

==========

```
SQL> SELECT * FROM TEST1;

       SNO NAME
---------- ----------
        10 SMITH
        20 ALLEN

SQL> SELECT * FROM TEST2;

       SNO        SAL
----------        ----------
        10      23000
        30      45000
```

EX:

```
SQL> SELECT * FROM TEST1 T1 JOIN TEST2 T2 ON T1.SNO<T2.SNO;
SQL> SELECT * FROM TEST1 T1 JOIN TEST2 T2 ON T1.SNO<=T2.SNO;
SQL> SELECT * FROM TEST1 T1 JOIN TEST2 T2 ON T1.SNO>T2.SNO;
SQL> SELECT * FROM TEST1 T1 JOIN TEST2 T2 ON T1.SNO>=T2.SNO;
SQL> SELECT * FROM TEST1 T1 JOIN TEST2 T2 ON T1.SNO!=T2.SNO;
```

EX:

waq to display employees whose salary is between low salary and high salary?

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

OUTER JOINS:

==========

- Outer joins are again three types.

i) Left outer join:

==========

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

EX:

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

ii) Right outer join:

==========

- retrieving matching rows from both tables and unmatching rows from the right side table only.

EX:

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

iii) Full outer join:

==========

- it is a combination of left outer join and right outer join.

- by using full outer join we will retrieve 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;

CROSS JOIN:

==========

- Joining two or more than two tables without any condition.

- In cross join mechanism each row of a table will joins 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 S CROSS JOIN COURSE C;

EX:

DEMO_TABLES:

==========

SQL> SELECT * FROM ITEMS1;

SNO	INAME	PRICE
1	PIZZA	180
2	BURGER	85

SQL> SELECT * FROM ITEMS2;

SNO	INAME	PRICE
11	PEPSI	20
12	COCACOLA	25

EX:

```
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:

==========

- Retrieving matching rows from the tables just like equi join.

- Here natural join condition is preparing by system implicitly based on common column in the tables with " = " operator.

- Natural join is eliminate duplicate columns from the result set.

EX:

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

SELF JOIN:

=========

- Joining a table by itself is called as "self join".

(or)

- Comparing a table data by itself is also called as "self join".

- Self join can be implemented on a single table only.

- When we use self join we must required alias names otherwise it cannot be implemented.

- Once we created alias name on table internally system will prepare virtual copy of a table on each alias name automatically.

- we can create any no.of alias names on a single table but each alias name should be different.

- Self join can use at levels those are,

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.

Examples on Comparing a single column values by itself with in the table:

==========

DEMO_TABLE:

==========

```
SQL> SELECT * FROM TEST;

ENAME      LOC
----------          ----------
SMITH      HYD
ALLEN      MUMBAI
MILLER     HYD
JONES      CHENNAI
ADAMS      PUNE
```

EX:

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

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

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='SCOTT';
```

Examples on 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 MANAGER,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 "BLAKE" manager?

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

Ex:

waq to display manager of BLAKE employee?

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

Ex:

waq to display employees who are joined before their manager?

```
SQL> SELECT E.ENAME AS EMPLOYEES,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 their manager salary?

```
SQL> SELECT E.ENAME AS EMPLOYEES,E.SAL AS EMP_SAL,
  2  M.ENAME AS MANAGER,M.SAL AS MGR_SAL FROM
  3  EMP E JOIN EMP M ON M.EMPNO=E.MGR
  4  AND E.SAL>M.SAL;
```

How to join more than two tables:

==========

syntax:

======

SELECT * FROM <TN1> <JOIN KEY> <TN2> ON <JOIN CONDITION1>

<JOIN KEY> <TN3> ON <JOIN CONDITION2>

<JOIN KEY> <TN4> ON <JOIN CONDITION3>

.....

.....

<JOIN KEY> <TN n> ON <JOIN CONDITION n-1>;

DEMO_TABLE:

==========

SQL> SELECT * FROM REGISTER;

REGNO	REGDATE	CID
1001	29-AUG-24	1
1002	30-AUG-24	3

SQL> SELECT * FROM STUDENT;

SQL> SELECT * FROM COURSE;

SQL> SELECT * FROM REGISTER;

EX:

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

==========

Constraints:

==========

- **Constraints are used to enfore unwanted data(i.e invalid data) from columns in the table.**
- **by using constraints we can apply validations on a table.**
- **oracle supports the following types of constraints those are,**
  - **i) UNIQUE**
  - **ii) NOT NULL**
  - **iii) CHECK**

iv) PRIMARY KEY

v) FOREIGN KEY / REFERENCES

vi) DEFAULT

syntax:

======

create table <table name>(<column name1> <datatype>[size] <constraint type>,

<column name2> <datatype>[size] <constraint type>,.....);

i) UNIQUE:

=========

- to restricted duplicate values but allowed nulls.

Ex:

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

TESTING:

SQL> INSERT INTO TEST VALUES(1,'A');----------> ALLOWED

SQL> INSERT INTO TEST VALUES(1,'A');----------> NOT ALLOWED

SQL> INSERT INTO TEST VALUES(NULL,NULL);-------> ALLOWED

ii) NOT NULL:

===========

- to restricted nulls but allowed duplicate values.

Ex:

SQL> CREATE TABLE TEST2(SNO NUMBER(2) NOT NULL,NAME VARCHAR2(10) NOT NULL);

TESTING:

SQL> INSERT INTO TEST2 VALUES(1,'A');----------ALLOWED

SQL> INSERT INTO TEST2 VALUES(1,'A');----------ALLOWED

SQL> INSERT INTO TEST2 VALUES(NULL,NULL);---------NOT ALLOWED

iii) CHECK:

=========

- to check a value with user defined condition before accepting into a column.

Ex:

```
SQL> CREATE TABLE TEST3(REG_NO NUMBER(4) UNIQUE NOT NULL,
  2  NAME VARCHAR2(10) NOT NULL,
  3  ENTRY_FEE NUMBER(6,2) NOT NULL CHECK(ENTRY_FEE=500),
  4  AGE NUMBER(3) NOT NULL CHECK(AGE BETWEEN 18 AND 30),
  5  LOC VARCHAR2(10)NOT NULL CHECK(LOC IN('HYD','MUMBAI','DELHI')));
```

TESTING:

```
SQL> INSERT INTO TEST3 VALUES(1001,'SMITH',1000,17,'HYDERABAD');------NOT ALLOWED
SQL> INSERT INTO TEST3 VALUES(1001,'SMITH',500,18,'HYD');------ALLOWED
```

iv) PRIMARY KEY:

==========

- it is a combination of UNIQUE and NOT NULL.

- by using primary key we will restrict duplicate and nulls at a time.

- a table is having only one primary key.

Ex:

```
SQL> CREATE TABLE TEST4(PCODE NUMBER(4)PRIMARY KEY,PNAME VARCHAR2(10)UNIQUE NOT
NULL);
```

TESTING:

```
SQL> INSERT INTO TEST4 VALUES(1021,'P1');---------ALLOWED
SQL> INSERT INTO TEST4 VALUES(1021,'P1');---------NOT ALLOWED
SQL> INSERT INTO TEST4 VALUES(NULL,NULL);-----NOT ALLOWED
SQL> INSERT INTO TEST4 VALUES(1022,'P2');--------ALLOWED
```

COMPOSITE PRIMARY KEY:

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

- when we apply a primary key constraint on combination of mulitple columns are called as "composite primary key".

- in composite primary key individual colums are accepting duplicate values but combination of columns are not accepting duplicate values.

syntax:

======

create table <table name>(<column name1> <datatype>[size],

<column name2> <datatype>[size],.....,primarykey(<col1>,<col2>,.....);

EX:

SQL> CREATE TABLE TEST5(SNO NUMBER(4),NAME VARCHAR2(10),

PRIMARY KEY(SNO,NAME));

TESTING:

=========

SQL> INSERT INTO TEST5 VALUES(1,'A');-----ALLOWED

SQL> INSERT INTO TEST5 VALUES(1,'A');-----NOT ALLOWED

SQL> INSERT INTO TEST5 VALUES(1,'B');----ALLOWED

SQL> INSERT INTO TEST5 VALUES(2,'A');----ALLOWED

DEFAULT:

========

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

syntax:

=======

<column name> <datatype>[size] default <value/expression>

Ex:

```
SQL> CREATE TABLE TEST6(STID NUMBER(4),SFEE NUMBER(6,2) DEFAULT 5000);
```

TESTING:

```
SQL> INSERT INTO TEST6(STID,SFEE)VALUES(1,7500);
SQL> INSERT INTO TEST6(STID)VALUES(2);
SQL> SELECT * FROM TEST6;
```

FOREIGN KEY:

==========

- to make relationship between tables.
- by using relationship we will take a referencial identity from one table to another table.

syntax:

======

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

Ex:

```
SQL> CREATE TABLE DEPT1(DNO NUMBER(2) PRIMARY KEY,DNAME VARCHAR2(10));----parent table

SQL> INSERT INTO DEPT1 VALUES(10,'ORACLE');
SQL> INSERT INTO DEPT1 VALUES(20,'JAVA');
SQL> COMMIT

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

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

SQL> INSERT INTO EMP1 VALUES(1023,'WARD',20);

SQL> INSERT INTO EMP1 VALUES(1024,'JONES',NULL);

Updating NULL in child table with referencial values of parent table only:

==========

Ex:

SQL> UPDATE EMP1 SET DNO=10 WHERE DNO IS NULL;-------Allowed

SQL> UPDATE EMP1 SET DNO=20 WHERE DNO IS NULL;-------Allowed

SQL> UPDATE EMP1 SET DNO=30 WHERE DNO IS NULL;-------Not Allowed

Note:

=====

- Once we establish relationship between tables there are two rules are come into picture.

Rule-1(Insertion rule):

==========

- we cannot insert values into child table when those values are not existing in parent table.

i.e No paranet = No child

Ex:

SQL> INSERT INTO EMP1 VALUES(1025,'MARTIN',30);

ERROR at line 1:

ORA-02291: integrity constraint (MYDB9AM.SYS_C009201) violated - parent key not found

Rule-2(Deletion rule):

==========

- we cannot delete a row from parent table when those parent rows are having the corresponding child rows in child table without addressing to child.

EX:

SQL> DELETE FROM DEPT1 WHERE DNO=10

ERROR at line 1:

ORA-02292: integrity constraint (MYDB9AM.SYS_C009201) violated - child record found

How to address to child table if we want to delete a row from parent table:

==========

i) ON DELETE CASCADE

ii) ON DELETE SET NULL

i) ON DELETE CASCADE:

==========

- when we delete a row from parent table then the corresponding childe rows also deleted from child table automatically.

Ex:

SQL> CREATE TABLE DEPT2(DNO NUMBER(2) PRIMARY KEY,DNAME VARCHAR2(10));----parent table

SQL> INSERT INTO DEPT2 VALUES(10,'ORACLE');

SQL> INSERT INTO DEPT2 VALUES(20,'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 table

SQL> INSERT INTO EMP2 VALUES(1021,'SMITH',10);

SQL> INSERT INTO EMP2 VALUES(1022,'ALLEN',20);

TESTING:

SQL> DELETE FROM DEPT2 WHERE DNO=20;-----allowed

ii) ON DELETE SET NULL:

==========

- when we delete a row from a parent table the the corresponding child table foreign key column values are converting into NULL automatically.

Ex:

Ex:

SQL> CREATE TABLE DEPT3(DNO NUMBER(2) PRIMARY KEY,DNAME VARCHAR2(10));----parent table

SQL> INSERT INTO DEPT3 VALUES(10,'ORACLE');

SQL> INSERT INTO DEPT3 VALUES(20,'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 table

SQL> INSERT INTO EMP3 VALUES(1021,'SMITH',10);

SQL> INSERT INTO EMP3 VALUES(1022,'ALLEN',20);

TESTING:

SQL> DELETE FROM DEPT3 WHERE DNO=20;-----allowed

==========

How to add constraints to an existing table:

==========

syntax:

======

ALTER TABLE <TN> ADD <CONSTRAINT> <CONSTRAINT KEY ID> <CONSTRAINT TYPE>(<COLUMN NAME>);

EX:

SQL> CREATE TABLE PARENT(EID NUMBER(4),ENAME VARCHAR2(10),SAL NUMBER(8,2));

Table created.

Adding Primary key:

==========

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

NOTE:

=====

- if we want to view constraint name along with column name of a specific table in oracle then we use a pre-defined table(i.e datadictionary) is "user_cons_columns".

Ex:

SQL> DESC USER_CONS_COLUMNS;

SQL> SELECT CONSTRAINT_NAME,COLUMN_NAME FROM USER_CONS_COLUMNS

2 WHERE TABLE_NAME='PARENT';

CONSTRAINT_NAME	COLUMN_NAME
PK_EID	EID

Adding Unique constraint:

==========

SQL> ALTER TABLE PARENT ADD CONSTRAINT UQ_ENAME UNIQUE(ENAME);

Adding Check constraint:

==========

SQL> ALTER TABLE PARENT ADD CONSTRAINT CHK_SAL CHECK(SAL>15000);

Adding "Not Null" constraint:

==========

syntax:

=======

ALTER TABLE <TN> MODIFY <COLUMN NAME> <CONSTRAINT> <CONSTRAINT KEY ID> NOT NULL;

EX:

SQL> ALTER TABLE PARENT MODIFY ENAME CONSTRAINT NN_ENAME NOT NULL;

Adding Foreign key constraint:

==========

syntax:

======

ALTER TABLE <TN> ADD CONSTRAINT <CONSTRAINT KEY ID>

FOREIGN KEY(COMMON COLUMN OF CHILD TABLE) REFERENCES

<PARENT TABLE NAME>(COMMON COLUMN OF PARENT TABLE)

ON DELETE CASCADE / ON DELETE SET NULL;

EX:

SQL> CREATE TABLE CHILD(DNAME VARCHAR2(10),EID NUMBER(4));

SQL> ALTER TABLE CHILD ADD CONSTRAINT FK_EID FOREIGN KEY(EID)

REFERENCES PARENT(EID) ON DELETE CASCADE;

SQL> SELECT CONSTRAINT_NAME,COLUMN_NAME FROM USER_CONS_COLUMNS

WHERE TABLE_NAME='CHILD';

CONSTRAINT_NAME COLUMN_NAME

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

FK_EID EID

How to drop a constraint from an existing table:

==========

syntax:

======

ALTER TABLE <TN> DROP <CONSTRAINT> <CONSTRAINT KEY ID>;

Dropping a Primary key:

==========

Case-1: With relationship:

==========

SQL> ALTER TABLE PARENT DROP CONSTRAINT PK_EID CASCADE;

Case-2: Without relationship:

==========

SQL> ALTER TABLE PARENT DROP CONSTRAINT PK_EID;

Dropping Unique,Check,Not Null constraint:

==========

SQL> ALTER TABLE PARENT DROP CONSTRAINT UQ_ENAME;

SQL> ALTER TABLE PARENT DROP CONSTRAINT CHK_SAL;

SQL> ALTER TABLE PARENT DROP CONSTRAINT NN_ENAME;

How to rename a constraint name:

==========

syntax:

=======

ALTER TABLE <TN> RENAME CONSTRAINT <OLD CONSTRAINT NAME> TO <NEW CONSTRAINT NAME>;

EX:

SQL> CREATE TABLE TEST7(SNO NUMBER(2) PRIMARY KEY);

SQL> SELECT CONSTRAINT_NAME,COLUMN_NAME FROM USER_CONS_COLUMNS

WHERE TABLE_NAME='TEST7';

CONSTRAINT_NAME COLUMN_NAME

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

SYS_C009215 SNO

SQL> ALTER TABLE TEST7 RENAME CONSTRAINT SYS_C009215 TO SNO_PK;

SQL> SELECT CONSTRAINT_NAME,COLUMN_NAME FROM USER_CONS_COLUMNS

WHERE TABLE_NAME='TEST7';

CONSTRAINT_NAME COLUMN_NAME

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

SNO_PK SNO

How to add / remove default value to / from an existing table:

==========

syntax:

======

alter table <table name> modify <column name> default <value>;

Ex:

SQL> CREATE TABLE TEST8(SNO NUMBER(2),LOC VARCHAR2(10));

Adding default value:

==========

SQL> ALTER TABLE TEST8 MODIFY LOC DEFAULT 'HYD';

NOTE:

=====

- if we want to view default value of a column in particular table

then use a datadictionary is "user_tab_columns".

EX:

```
SQL> DESC USER_TAB_COLUMNS;
SQL> SELECT COLUMN_NAME,DATA_DEFAULT FROM USER_TAB_COLUMNS
     WHERE TABLE_NAME='TEST8';

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

Removing default value:

```
====================
SQL> ALTER TABLE TEST8 MODIFY LOC DEFAULT NULL;

SQL> SELECT COLUMN_NAME,DATA_DEFAULT FROM USER_TAB_COLUMNS
     WHERE TABLE_NAME='TEST8';

COLUMN_NAME                    DATA_DEFAULT
-----------------------------------  ---------------------------------------------
LOC                            NULL
```

```
=========================================================================
SUBQUERY / NESTED QUERY:
=========================
```

- a query inside another query is called as "subquery/nested query".

syntax:

```
======
select * from <table name> where <condition>(select * from ......(select * from.....));
```

- a subquery statement is having two more queries those are,

    i) inner query / subquery / child query

    ii) outer query / main query / parent query

- as per the execution process of subquery statement it again classified into two types those are,

    1) Non-corelated subquery:

    ==========

        - in this mechanism first inner query is executed and later outer query will execute.

    2) Co-related subquery:

    ==========

        - in this mechanism first outer query is executed and later inner query will execute.

1) Non-corelated subquery:

==========

- in this mechanism first inner query is executed and later outer query will execute.

- it again classified into three types.

    i) single row subquery

    ii) multiple row subquery

    iii) multiple column subquery

i) single row subquery:

==========

- when a subquery return a single value is known as "SRSQ".

- in SRSQ we will use the following operators are " = , < , > , <= , >= , !=(or) < > ".

Ex:

waq to display employees details who getting the first 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 <use 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 find out the second highest salary from emp table?

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

MAX(SAL)

3000

Ex:

waq to display employees details who are earning the second 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 whose salary is more than to 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 the 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 is known as "MRSQ".**
- **in multiple row subquery we will use the following operators are "IN,ANY,ALL".**

Ex:

waq to display employees whose job is same as the job of the employees "SMITH","MARTIN"?

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

Ex:

waq to display employees details who are earning minimum,maximum salaries from emp table?

```
SQL> SELECT * FROM EMP WHERE SAL IN
  2  (
  3  SELECT MIN(SAL) FROM EMP
  4  UNION
  5  SELECT MAX(SAL) FROM EMP
  6  );
```

ANY operator:

==========

- **it return a value if any one value is satisfied with the given conditional value.**

Ex:

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

- **i) X=09 ===> FALSE**
- **ii) X=25 ===> TRUE**
- **iii) X=40 ===> TRUE**

ALL operator:

==========

- it return a value if all values are satisfied with the given conditional value.

Ex:

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

i) X=09 ===> FALSE

ii) X=25 ===> FALSE

iii) X=40 ===> TRUE

Ex:

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

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

Ex:

waq to display employees details whose salary is more than to all "salesman" salaries?

```
SQL> SELECT * FROM EMP WHERE SAL>ALL(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:

==========

- multiple columns values of inner query is comparing with multiple columns values

of outer query is known as "MCSQ".

syntax:

======

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

Ex:

waq to display employees details whose job,mgr are same the job,mgr of the employee "ALLEN"?

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

Ex:

waq to display employees who are getting 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 the senior most employees details from each deptno wise?

SQL> SELECT * FROM EMP WHERE(DEPTNO,HIREDATE)IN

2 (SELECT DEPTNO,MIN(HIREDATE) FROM EMP GROUP BY DEPTNO);

2) Co-related subquery:

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

- in this mechanism first outer query is executed and later

inner query will execute.

Syntax for to find out "Nth" high / low salary:

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

SELECT * FROM <TABLE NAME> <TABLE ALIAS NAME1> WHERE N-1=(SELECT COUNT(DISTINCT <COLUMN NAME>)

FROM <TABLE NAME> <TABLE ALIAS NAME2> WHERE <TABLE ALIAS NAME2>.<COLUMN NAME>

< (or) > <TABLE ALIAS NAME1>.<COLUMN NAME>);

Here,

< - finding lowest salary

> - finding highest salary

EX:

waq to find out employees who are getting 1st highest salary?

DEMO_TABLE:

==========

SQL> SELECT * FROM TEST;

ENAME	SAL
ALLEN	85000
JONES	55000
ADAMS	85000
SMITH	23000
MILLER	68000

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 employees who are getting 4th highest salary?

Solution:

========

If N=4

===> 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 employees who are getting 1st lowest salary?

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);
```

Syntax for to find out "TOP n" high / low salaries:

==

SELECT * FROM <TABLE NAME> <TABLE ALIAS NAME1> WHERE N>(SELECT COUNT(DISTINCT <COLUMN NAME>)

FROM <TABLE NAME> <TABLE ALIAS NAME2> WHERE <TABLE ALIAS NAME2>.<COLUMN NAME>

< (or) > <TABLE ALIAS NAME1>.<COLUMN NAME>);

Here,

< - finding lowest salary

> - finding highest salary

Ex:

waq to display top 3 highest salaries employees details?

```
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?

```
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 a special operator which is used in co-related subquery.**
- **it is used to check the required row is existing in a table or not.**
  - **> if a row is existing in a table then it return "TRUE".**
  - **> if a row is not existing in a table then it return "FALSE".**

syntax:

======

```
WHERE EXISTS(<subquery>);
```

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);
```

How to delete multiple duplicate rows except one row from a table:

==========

- when we want to delete multiple duplicate rows from a table then we should use a pseudo column in oracle is "ROWID".

What is ROWID:

==========

- it is a pseudo column in oralce.which is used to generate row identification / row address to each row wise in a table automatically.these ROWID's are saved in database permanently.

EX:

```
SQL> SELECT EMP.*,ROWID FROM EMP;

SQL> SELECT MIN(ROWID)FROM EMP;

MIN(ROWID)
------------------
AAAWZoAAHAAAAHeAAA

SQL> SELECT MAX(ROWID)FROM EMP;

MAX(ROWID)
------------------
AAAWZoAAHAAAAHeAAN
```

Deleting duplicate rows:

==========

EX:

```
SQL> SELECT * FROM TEST;

     SNO NAME
---------- ----------
```

1 A

1 A

1 A

2 B

3 C

3 C

4 D

4 D

4 D

5 E

5 E

Solution:

========

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

SQL> SELECT * FROM TEST;

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

==========

Views:

======

- it is a virtual object / subset of a base table(i.e main table).

- it does not store data but it can access data from a base table.

- whenever we perform DML operations on a view internally those operations

are executed on a base table and refelected in view table to user.

Types of views:

==========

1. simple view

2. complex view

1. simple view:

==========

- when we create a view to access the required data from a single base table is known as "simple view".

- by default simple view allowed DML operations on a base table.

syntax:

=======

create view <view name> as <select query>;

Ex:

create a view to access all departments details from DEPT table?

SQL> SELECT * FROM DEPT;----> main table

SQL> CREATE VIEW V1 AS SELECT * FROM DEPT;

SQL> SELECT * FROM V1;------> view table

TESTING:

SQL> INSERT INTO V1 VALUES(50,'SAP','HYD');-------------------------> Allowed

SQL> UPDATE V1 SET LOC='MUMBAI' WHERE DEPTNO=50;------> Allowed

SQL> DELETE FROM V1 WHERE DEPTNO=50;-----------------------------> Allowed

VIEW OPTIONS:

==========

- view can be created with two options.

i) WITH READ ONLY

ii) WITH CHECK OPTION

i) WITH READ ONLY:

==========

- if we want to restricted DML operations on a base table through a view object then we should use "with read only" statement at the time creating a view.

Ex:

SQL> SELECT * FROM DEPT;----> main table

SQL> CREATE VIEW V2 AS SELECT * FROM DEPT WITH READ ONLY;

SQL> SELECT * FROM V2;------> view table

TESTING:

SQL> INSERT INTO V2 VALUES(50,'SAP','HYD');----------> Not Allowed

SQL> UPDATE V2 SET LOC='MUMBAI' WHERE DEPTNO=50;------> Not Allowed

SQL> DELETE FROM V2 WHERE DEPTNO=50;----------> Not Allowed

ii) WITH CHECK OPTION:

==========

- to restricted unwanted data into a base table through a view object.

EX:

SQL> CREATE VIEW V3 AS SELECT * FROM DEPT WHERE LOC='HYD' WITH CHECK OPTION;

TESTING:

SQL> INSERT INTO V3 VALUES(10,'ACCOUNTING','MUMBAI');------Not Allowed

SQL> INSERT INTO V3 VALUES(10,'ACCOUNTING','HYD');----------Allowed

SQL> SELECT * FROM DEPT;

SQL> SELECT * FROM V3;

2. complex view:

==========

- **when we create a view based on :**
  - **> multiple tables**
  - **> by using group by**
  - **> by using aggregative / grouping functions**
  - **> by using having**
  - **> by using distinct keyword**
  - **> by using set operators**
  - **> by using joins**
- **by default complex views are not allowed DML operations on base tables.**

syntax:

=======

create a view <view name> as <select query>;

EX:

```
SQL> CREATE VIEW V4 AS
  2  SELECT * FROM EMP_HYD
  3  UNION
  4  SELECT * FROM EMP_MUMBAI;
```

- **On complex view we cannot perform DML operations.**

EX:

```
SQL> CREATE VIEW V5 AS
  2  SELECT DEPTNO,SUM(SAL) AS SUM_OF_SALARY
  3  FROM EMP GROUP BY DEPTNO;
```

TESTING:

=========

SQL> SELECT * FROM EMP;------>(before updating)

SQL> UPDATE EMP SET SAL=SAL+1000 WHERE DEPTNO=10;

SQL> SELECT * FROM V5;--------->(after updating)

==========

SEQUENCE:

==========

- it is a database object which is used to generate sequence numbers on a specific column in the table automatically.

- it will provide "auto incremental value" facility on a table.

syntax:

=======

create sequence <sequence name>

[start with n]

[minvalue n]

[increment by n]

[maxvalue n]

[no cycle / cycle]

[no cache / cache n];

start with n:

==========

- to specify starting value of the sequence.

- here "n" is number.

minvalue n:

==========

- to show minimum value of the sequence.

- here "n" is number.

increment by n:

==========

- to specify incremental value in between sequence numbers.

- here "n" is number.

maxvalue n:

==========

- to show maximum value of the sequence.

- here "n" is number.

no cycle:

=======

- it is a default attribute of sequence.

- when we created a sequence object with "no cycle" then the set of sequence numbers are not repeated again and again.

cycle:

=====

- when we created a sequence object with "cycle" then the set of sequence numbers are repeated again and again.

```
create seq s1                    create seq s2
sv 1                             sv 1
miv 1                            miv 1
incr by 1                              incr by 1
maxv 3;                          maxv 3
                                 cycle;
output:
=======                                output:
        1                        =======
        2                        1
        3 (stop)                 2
                                 3 (continue)
```

1

2

3

1

2

3

no cache:

========

- it is a default attribute of sequence object.
- cache is a temporary file memory created by user at the time of creating a sequence object.
- when we created a sequence object with "no cache" then the set of sequence numbers are saved in database directly.so that each and every user request is going to database and fetching data from database to client application. it leads the burdon on database and reduce the performance of database.

cache n:

=======

- when we created a sequence object with "cache" then the set of sequence numbers are saved in database and also the copy of data keep in cache memory. so that each and every user request is going to cache instead of database and fetching data from cache to client application.so that it reduce the burdon on database and improve the performance of database.
- cache file minimum size is 2kb and maximum size is depends on ram.

Nextval:

=======

- it is a pseudo column which is used to generate next by next sequence number on a specific column in the table.

syntax:

=======

<sequence name>.nextval

EX:

```
SQL> CREATE SEQUENCE SQ1
  2  START WITH 1
  3  MINVALUE 1
  4  INCREMENT BY 1
  5  MAXVALUE 3;
```

TESTING:

```
SQL> CREATE TABLE TEST21(SNO NUMBER(2),NAME VARCHAR2(10));

SQL> INSERT INTO TEST21 VALUES(SQ1.NEXTVAL,'&NAME');
Enter value for name: A

SQL> /
Enter value for name: B

SQL> /
Enter value for name: C
```

OUTPUT:

========

```
SQL> SELECT * FROM TEST21;

       SNO NAME
---------- ----------
         1 A
         2 B
         3 C
```

EX:

```
SQL> CREATE SEQUENCE SQ2
  2  START WITH 3
  3  MINVALUE 1
  4  INCREMENT BY 1
  5  MAXVALUE 5
  6  CYCLE
  7  CACHE 2;
```

TESTING:

```
SQL> CREATE TABLE TEST22(SNO NUMBER(3),NAME VARCHAR2(10));

SQL> INSERT INTO TEST22 VALUES(SQ2.NEXTVAL,'&NAME');
Enter value for name: A

SQL> /
.....

SQL> /
.....
```

OUTPUT:

```
=======
SQL> SELECT * FROM TEST22;

       SNO NAME
---------- ----------
         3 A
         4 S
         5 D
         1 F
```

2 G

3 H

4 J

5 K

==========

INDEXES:

========

- it is a db object which is used to retrieve a specific row from a table fastly.

- all databases are supporting the following two types of searching mechanisms those are,

1. Table scan

2. Index scan

1. Table scan:

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

- it is a default searching mechanism of any database.

- in this scan oracle server is scanning the entire table for required data.so that it leads time consume.

Ex:

SQL> SELECT * FROM EMP WHERE SAL=3000;

SAL

800

1600

1250

2975

1250

2850

2450 **SAL=3000**

3000

5000

1500

1100

950

3000

1300

2. Index scan:

==========

- in this scan oracle server is scanning based on an indexed column wise for required data.

How to create an index object:

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

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;

- when we created an index object based on SAL column internally system will arrange SAL column values in the form B-Tree format like below,

B-TREE INDEX FORMAT

||

(lp) < || 3000 || >= (rp) ----------> root level

|| ||

(lp)<||2975||>=(rp) (lp) < || 5000 || >= (rp) ----------> parent level

|| ||

2850|*,2450|*,1600|*, 3000|*(h),*(m)

1500|*,1300|*,1250|*,*,

1100|*,950|*,800|*

NOTE:

=====

- to view index name along with column name of a particular table in oracle database then use a datadictionary is "user_ind_columns".

EX:

```
SQL> DESC USER_IND_COLUMNS;
SQL> SELECT COLUMN_NAME,INDEX_NAME FROM USER_IND_COLUMNS
     WHERE TABLE_NAME='EMP';
```

How to drop index:

==========

syntax:

======

```
DROP INDEX <INDEX NAME>;
```

EX:

```
DROP INDEX I1;
```

==========

TRANSACTION CONTROL LANGUAGE(TCL):

==========

Transaction:

==========

- to perform some operation over database.

- to control transactions on database then we use TCL commands.

1) COMMIT

2) ROLLBACK

3) SAVEPOINT

1) COMMIT:

==========

- to make a transaction is permanent.

- there are two types of commit transactions.

i) Implicit commit:

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

- these transactions are committed by the system

- automatically.

EX: DDL commands (Auto committed)

ii) Explicit commit:

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

- these transactions are committed by user.

Ex: DML commands

syntax:

======

commit;

EX:

```
SQL> CREATE TABLE TEST(SNO NUMBER(2),NAME VARCHAR2(10));

SQL> INSERT INTO TEST VALUES(1,'A');
SQL> COMMIT;

SQL> UPDATE TEST SET SNO=101 WHERE SNO=1;
SQL> COMMIT;

SQL> DELETE FROM TEST WHERE SNO=101;
SQL> COMMIT;
```

(OR)

```
SQL> INSERT INTO TEST VALUES(1,'A');
SQL> UPDATE TEST SET SNO=101 WHERE SNO=1;
SQL> DELETE FROM TEST WHERE SNO=101;
SQL> COMMIT;
```

2) ROLLBACK:

==========

- to cancel a transaction.
- once we committed then we cannot rollback.

syntax:

=======

```
rollback;
```

EX:

```
SQL> INSERT INTO TEST VALUES(1,'A');
SQL> ROLLBACK;

SQL> UPDATE TEST SET SNO=101 WHERE SNO=1;
SQL> ROLLBACK;

SQL> DELETE FROM TEST WHERE SNO=101;
SQL> ROLLBACK;
```

(OR)

```
SQL> INSERT INTO TEST VALUES(1,'A');
SQL> UPDATE TEST SET SNO=101 WHERE SNO=1;
SQL> DELETE FROM TEST WHERE SNO=101;
SQL> ROLLBACK;
```

3) SAVEPOINT:

==========

- to rollback a specific row / rows.

How to create a savepoint:

==========

syntax:

=======

savepoint <pointer name>;

How to rollback a savepoint:

==========

syntax:

=======

rollback to <pointer name>;

EX:

SQL> SELECT * FROM TEST;

SNO	NAME
1	A
2	B
3	C
4	D

SQL> DELETE FROM TEST WHERE SNO=1;

SQL> SAVEPOINT P1;

SQL> DELETE FROM TEST WHERE SNO=3;

CASE-1:

=======

SQL> ROLLBACK TO P1; --------(sno=3 is rollback)

CASE-2:

=======

SQL> ROLLBACK/COMMIT;

EX:

SQL> DELETE FROM TEST WHERE SNO=1;

SQL> SAVEPOINT P1;

SQL> DELETE FROM TEST WHERE SNO IN(3,4);

CASE-1:

=======

SQL> ROLLBACK TO P1; --------(sno=3,4 is rollback)

CASE-2:

=======

SQL> ROLLBACK/COMMIT;

==========

DATA CONTROL LANGUAGE(DCL):

==========

- **these commands are used for providing "security to database".**
- **thes DCL commands are handling by DBA only.**
    1) **Grant**
    2) **Revoke**

1) Grant:

=======

- **this command is used to grant permissions to users.**

syntax:

======

GRANT <privilege name> TO <username>;

EX:

SQL> CONN

Enter username : SYSTEM/TIGER

connected.

SQL> GRANT CONNECT TO MYDB9AM;-------> for connecting to oracle.

SQL> GRANT CREATE TABLE TO MYDB9AM; ----> for creating a table.

SQL> GRANT UNLIMITED TABLESAPCE TO MYDB9AM;------> for inserting data.

2) Revoke:

=========

- this command is used to cancel permissions of user.

syntax:

======

REVOKE <privilege name> FROM <username>;

EX:

SQL> CONN

Enter username : SYSTEM/TIGER

connected.

SQL> REVOKE CONNECT FROM MYDB9AM;-------> for cancel connecting to oracle.

SQL> REVOKE CREATE TABLE FROM MYDB9AM; ----> for cancel creating a table.

SQL> REVOKE UNLIMITED TABLESAPCE FROM MYDB9AM;------> for cancel inserting data.

==========

NORMALIZATION:

==========

What is Normalization?

==========

- Normalization is a technique to decompose(divide) a table data into multiple tables.

Where we use Normalization?

==========

- use at DB designing levels.

Why Normalization?

==========

Ex: Student_Branch_Details (De-Normalizaed Table)

==========

STID	SNAME	BRANCH	HOD	OFFICE_NUMBER
1021	smith	cse	Mr.x	040-22334455
1022	allen	cse	Mr.x	040-22334455
1023	ward	cse	Mr.x	040-22334455
1024	jones	cse	Mr.x	040-22334455

Disadvantages:

==========

> Dataredundancy problem.

> Occupied more memory.

> Datainconsistency problem.

> Insertion problem.

> Updation problem.

> Deletion problem.

- To overcome the above problems then we use a technique is called as "Normalization".

By using Normalization:

==========

(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	jones	1
				1025	miller	1

Advantages of Normalization?

==========

- **To avoid dataredundancy problem.**
- **Occupied less memory.**
- **To avoid datainconsistency problem.**
- **To avoid Insertion problem.**
- **To avoid Updation problem.**
- **To avoid Deletion problem.**

Types of Normalization 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

==========

1022 smith 1

1021 allen 2

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 (A, B) / COMPOSITE PRIMARY KEY>

EX: STU_ID SUB_ID STU_MARKS TEACHER

- Here, "STU_ID and SUB_ID " IS A KEY COLUMNS - " 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) |----------(cpk)----------|

SUB_ID SUB_NAME TEACHER STU_ID SUB_ID STU_MARKS

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

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.

|--------CPK----------|

STUDENT_ID SUBJECT_ID STU_MARKS EXAM_NAME TOTAL_MARKS

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

(Composite Primary key)

EX: STU_ID SUB_ID 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 (cpk) Score_Table
(fk)

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

EXAM_ID EXAM_NAME TOTAL_MARKS STUDENT_ID SUBJECT_ID STU_MARKS EXAM_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 match 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
1	c#

Hobbies_details(4NF)

STUDENT_ID	HOBBY
1	cricket
1	reading
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

==========

PL/SQL

======

EX:

waq to retrieve ENAME,SALARY details from emp table whose EMPNO is 7788?

IN SQL:

======

SQL> SELECT ENAME,SAL FROM EMP WHERE EMPNO=7788;

output:

======

ENAME SAL

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

SCOTT 3000

EX:

write a PL/SQL program to retrieve ENAME,SALARY details from emp table whose EMPNO is 7788?

IN PL/SQL:

=========

```
SQL> DECLARE
  2  v_ENAME  VARCHAR2(10);
  3  v_SAL NUMBER(8,2);
  4  BEGIN
  5  SELECT ENAME,SAL INTO v_ENAME,v_SAL FROM EMP
  6  WHERE EMPNO=7788;
  7  DBMS_OUTPUT.PUT_LINE(v_ENAME||','||v_SAL);
  8  END;
  9  /
```

OUTPUT:

========

SCOTT,3000

==========

CURSOR:

========

- **it is a temporary memory / sql private area / workspace.**
- **there are two types of cursors in pl/sql.**
    1. **Explicit cursor**
    2. **Implicit cursor**

1. Explicit cursor :

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

- **these cursors are created by user for retrieving multiple rows from a table.**
- **to create an explicit cursor then we need to follow the following 4 steps are,**

step1: Declare cursor variable:

==========

syntax:

======

declare cursor <cursor name> is <select query>;

step2: Open cursor connection:

==========

syntax:

=======

open <cursor name>;

step3: Fetching rows from a cursor:

==========

syntax:

=======

fetch <cursor name> into <variables>;

step4: Close cursor connection:

==========

syntax:

======

close <cursor name>;

Attributes of 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 open successfully otherwise it returns "false".

ii) %notfound:

==========

- it returns "true" when cursor is not having data otherwise "false".

iii) %found:

==========

- it returns "true" when cursor is having data otherwise "false".

iv) %rowcount:

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

- it return how many no.of fetch statements are executed.

- it return type is "number".

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

- Generally cursors are holding multiple rows but can access only one row at a time so that we used no.of fetch statements for fetching multiple rows from a cursor table.

- To overcome the above problem we must 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
  2  CURSOR C1 IS SELECT ENAME,SAL FROM EMP;
  3  v_ENAME VARCHAR2(10);
  4  v_SAL NUMBER(8,2);
  5  BEGIN
  6  OPEN C1;
  7  FETCH C1 INTO v_ENAME,v_SAL;
  8  WHILE(C1%FOUND)
  9  LOOP
 10  DBMS_OUTPUT.PUT_LINE(v_ENAME||','||v_SAL);
 11  FETCH C1 INTO v_ENAME,v_SAL;
 12  END LOOP;
 13  CLOSE C1;
```

```
 14  END;
 15  /
```

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

2) Implicit cursor:

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

- these cursors are declaring by oracle server by default whenever we perform DML operations on database table.

- implicit cursor memory is used by oracle for checking the status of DML operation is executed successfully or not.

==========

EXCEPTION HANDLING:

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

i) What is an Exception?

- it is a runtime error / execution error.

ii) What is an Exception Handling?

- to avoid abnormal terimination of a program execution process.

- Oracle supports the following two types exceptions.

1) Pre-defined exceptions

2) User-defined exceptions

1) Pre-defined exceptions:

==========

- these are defined by oracle server by default.which are used whenever the runtime error is occurred in a pl/sql program then we use the suitable exception name to avoid runtime error in a program.

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
```

SQLCODE & SQLERRM:

==========

- these are built-in properties in oracle which are used to handle any type of exception in a pl/sql program and display the information about an exception.

- when we use these properties we must use "others" exception name.

SQLCODE : it return exception number.

SQLERRM : it return exception message.

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  EXCEPTION
 11  WHEN OTHERS THEN
 12  DBMS_OUTPUT.PUT_LINE(SQLCODE);
 13  DBMS_OUTPUT.PUT_LINE(SQLERRM);
 14  END;
 15  /
Enter value for x: 10
Enter value for y: 2
5

SQL> /
Enter value for x: 10
Enter value for y: 0
-1476
ORA-01476: divisor is equal to zero
```

2) USER DEFINED EXCEPTION:

==========

- 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')
------------------------------------------------------------
```

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));
SQL> CREATE TABLE EMP44_AUDIT(EID NUMBER(4),AUDIT_INFR VARCHAR2(100));

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