

ORACLE

BY

MR.SUDHAKAR.L

NARESH TECHNOLOGIES

MYTHRI XEROX

ALL SOFTWARE INSTITUTE MATERIALS AVAILABLE

{ ADD:GAYATHRI NAGAR,SAP STREET,BEHIND HUDA MYTHRIVANAM,AMEERPET}

CELL.7036303535,7995810201

{GOODS ONCE SOLD WILL NOT BE TAKEN BACK}

ORACLE19c

Introduction of DBMS:

DATA: IT IS A RAW FACT (i.e. CHARACTERS & NUMBERS)

EX: EMPID IS DATA, ENAME IS DATA, SALARY IS DATA, DOJ IS DATA.....etc.

NOTE: DATA IS NEVER GIVES ACCURATE MEANINGFULL INFORMATION.

INFORMATION: PROCESSING DATA IS CALLED AS INFORMATIONS.

EX:

EMPID	ENAME	SALARY	DOJ	DEPTNAME
=====	=====	=====	====	=====
1021	X	25000.00	05-08-2020	DB
1022	Y	45000.00	24-12-2019	HR

NOTE: INFORMATION IS ALWAYS PROVIDES ACCURATE MEANINGFULL DATA OF PARTICULAR EMPLOYEE, CUSTOMER, STUDENT and PRODUCT..... etc.

DATA STORAGES: IT A LOCATION WHERE WE CAN STORE DATA / INFORMATION. WE HAS DIFFERENT TYPES OF DATA STORAGES.

1. BOOKS & PAPERS
2. FLAT FILE (FILE MANAGEMENT SYSTEM)
3. DBMS / DATABASE

1. DISADVANTAGES OF BOOKS & PAPERS:

- > IT IS COMPLETE MANUAL PROCEE / SYSTEM.
- > REQUIRED MORE MAN POWER.
- > COSTLY IN MAINTANANCE
- > THERE IS NO SECURITY
- > HANDLING A VERY SMALL DATA
- > RETRIEVING DATA WILL BE TIME CONSUME.

2. FLAT FILE (FILE MANAGEMENT S/STEM):

IN FILE MANAGEMENT DATA CAN BE STORED IN FILES.

DISADVANTAGES:

1. DATA REDUNDANCY & DATA INCONSISTANCY:

===== THESE PROBLEMS COMES INTO PICTURE WHEN WE STORE DATA IN MULTIPLE FILES. WHERE THE CHANGES ARE MADE IN ONE FILE WILL NOT BE REFLECTED TO ANOTHER COPY OF FILE .

BUT IN CASE OF DATABASE WE CAN MAINTAIN NO.OF COPIES OF SAME DATA AND STILL THE CHANGES MADE IN ONE COPY THEN REFLECTED TO OTHER COPY BECAUSE INTERNALLY MAINTAIN ACID PROPERTIES BY DEFAULT IN DATABASE.

2. DATA INTEGRITY :

===== THIS IS ABOUT MAINTAINING PROPER DATA IN EVERY ORGANIZATION IMPOSE SET INTEGRITY RULES ON DATA AND WE WILL CALL THESE RULES ARE BUSINESS RULES.

DATABASE PROVIDES AN OPTIONS FOR IMPOSING THE BUSINESS RULES WITH THE HELP OF CONSTRAINTS AND TRIGGERS.

3. DATA RETRIEVE:

===== IT IS PROCESS OF DATA RETRIEVING FROM DATA SOURCES. WHICH IS VERY COMPLEX WHILE RETRIEVING DATA FROM FILES WHICH WAS ADDRESSED WITH HIGH LEVEL LANGUAGE.

WHERE AS IF YOU WANT TO RETRIEVE DATA FROM DATABASE THEN WE ARE USING SQL LANGUAGE.

4. DATA SECURITY:

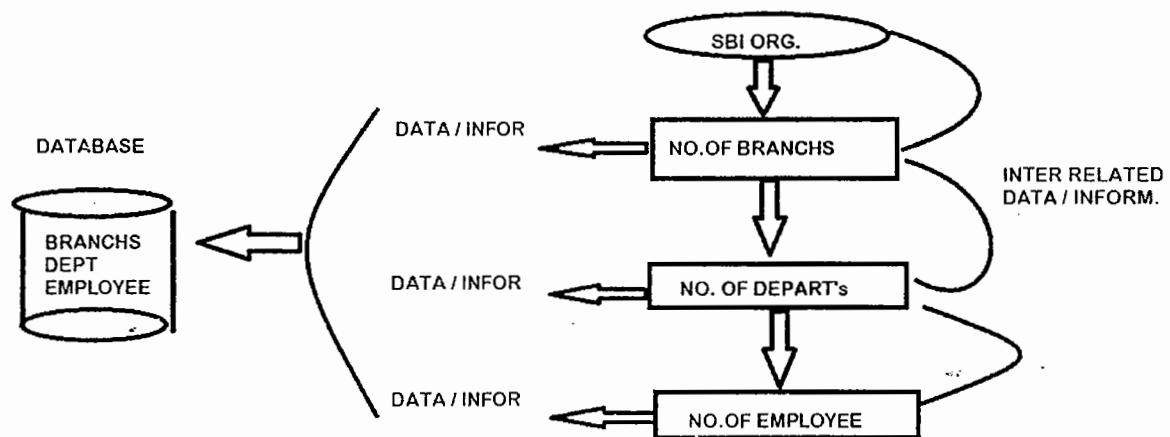
===== DATA IS NEVER SECURE UNDER BOOKS AND FLAT FILE WHERE AS DATABASE ARE PROVIDING AN EXCELLENT CONCEPT IS CALLED AS ROLE BASED SECURITY MECHANISM FOR ACCESSING DATA FROM DATABASE WITH SECURITY MANNER WITH THE HELP OF AUTHENTICATION AND AUTHORIZATION.

5. DATA INDEXING:

===== INDEXES ARE USING FOR ACCESSING DATA MUCH MORE FASTER BUT FLAT FILES DOES NOT PROVIDE ANY INDEX MECHANISM WHERE AS DATABASE WILL PROVIDE INDEXING MECHANISM.

3. DBMS / DATABASE:

DATABASE: IT IS COLLECTION OF INTER RELATED INFORMATION.BY USING DATABASE WE CAN STORE, MODIFY, SELECT AND DELETE DATA FROM DATABASE WITH SECURITY MANNER.



TYPES DATABASES:

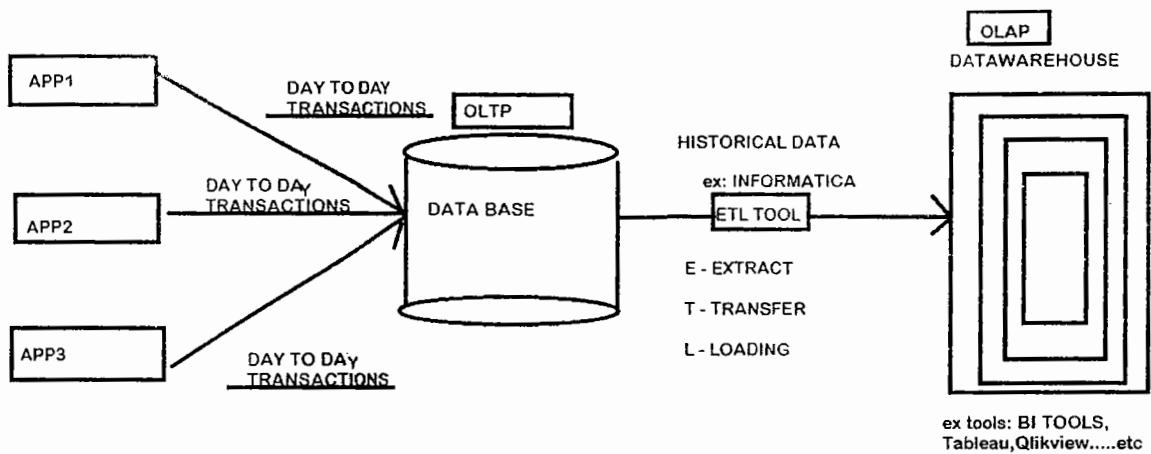
- 1) OLTP (ONLINE TRANSACTION PROCESSING)
- 2) OLAP (ONLINE ANALYTICAL PROCESSING)

OLTP: ORGANIZATIONS ARE MAINTAINING OLTP FOR STORING "DAY - TO - DAY TRANSACTIONS INFORMATION ". USING FOR "RUNNING BUSINESS ".

EX: SQLSERVER, ORACLE, MYSQL,etc.

OLAP: USED FOR DATA ANALYSIS (OR) DATA SUMMARIZED (OR) HISTORY OF DATA OF PARTICULAR BUSSINESS.

EX: DATAWAREHOUSE.



DBMS: IT IS A SOFTWARE WHICH IS USED TO MANAGE DATA IN DATABASE.

WHY DBMS:

EX:

BUSSINESS ----> (COLLECTION OF ENTITIES)

BRANCH		EMPLOYEE		PRODUCTS		CUSTOMERS	---->	(ENTITIES)
< starting >								
1 Bran		10 Emp		10 Pro		NO Cust.	-->(Gen. a very small data)	
< after 5 years >								
25 Bran		500 Emp		25 Pro		50000 Cust.	---->(Gen. small data)	
< after 10 years >								
100 Bran		10000 Emp		100 Pro		5 Lak's Cust.	----->(Gen. big/large data)	

ADVANTAGES OF DBMS:

1. TO REDUCE DATA REDUNDANCY.
2. TO AVOID DATA INCONSISTENCY.
3. EASY TO ACCESS DATA.
4. EASY TO MANIPULATE DATA.
5. MORE SECURITY (AUTHENTICATION & AUTHORIZATION)
6. IT SUPPORTS DATA INTEGRITY RULES.
7. SUPPORTING DATA SHARING
8. SUPPORTS TRANSACTIONS AND "ACID" PROPERTIES.

DBMS MODELS / DATABASE MODELS:

HOW DATA CAN BE ORGANIZED / STORE IN DIFFERENT DATABASE MODELS. THERE ARE THREE TYPES OF DATABASE MODELS ARE,

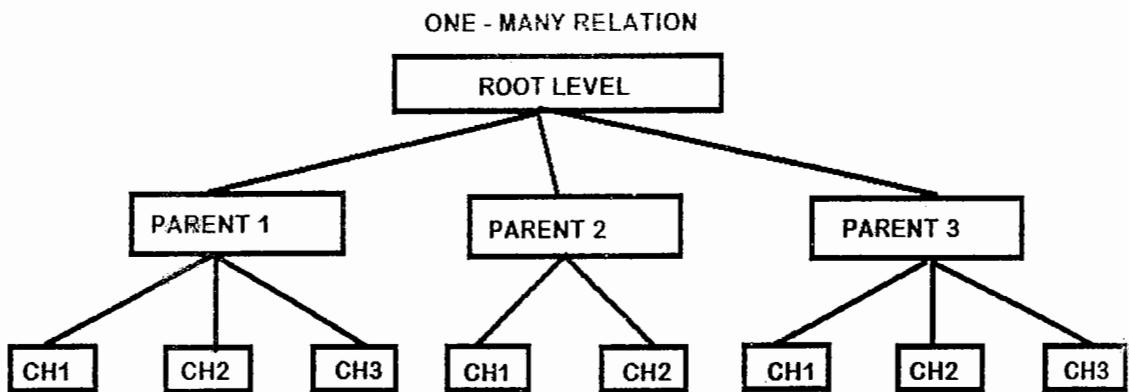
1. HIERARCHICAL DATABASE MANAGEMENT SYSTEM(HDBMS)
2. NETWORK DATABASE MANAGEMENT SYSTEM(NDBMS)
3. RELATIONAL DATABASE MANAGEMENT SYSTEM(RDBMS)
 - i) OBJECT RELATIONAL DBMS(ORDBMS)
 - ii) OBJECT ORIENTED RELATIONAL DBMS(OORDBMS)

HDBMS:

IT IS A FIRST MODEL OF DATABASE THAT CAME INTO EXISTANCE IN THE 1960's WHICH WILL ORGANIZE THE DATA IN THE FORM OF A "TREE STRUCTURE" AND WHICH WAS DESIGN BASED ON "ONE – MANY RELATION"

IN ONE – MANY RELATION EVERY CHILD IS HAVING ONLY ONE PARENT. THIS TREE IS CONTAINS THE FOLLOWING THREE LEVEL ROOT, PARENT AND CHILD LEVEL.

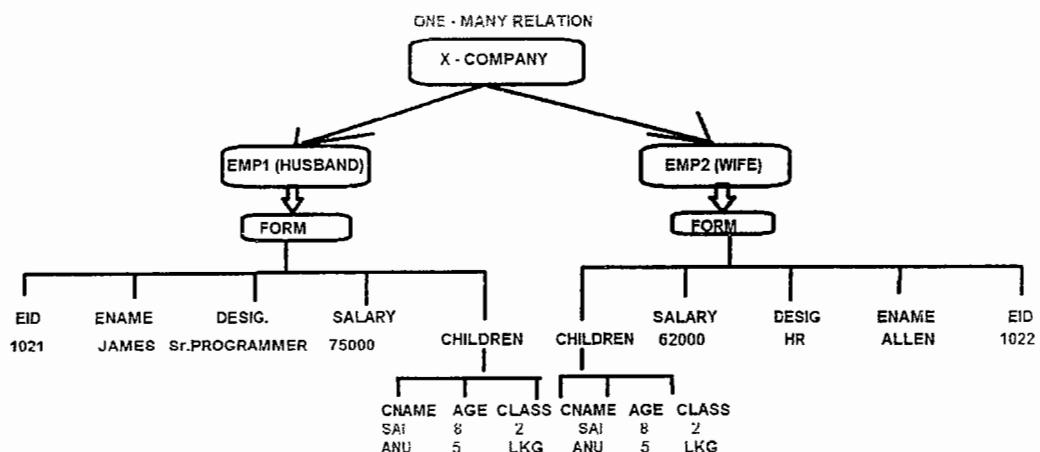
EX: IMS SOFTWARE (INFORMATION MANAGEMENT SYSTEM)



DISADVANTAGES:

1. WHEN WE WANT TO ADD A NEW LEVEL (PARENT / CHILD) TO AN EXISTING STRUCTURE THEN USER HAS TO RE CONSTRUCT THE ENTIRE STRUCTURE SO THAT IT LEADS TIME CONSUME.
2. WHEN WE WANT TO ACCESS DATA FROM THIS MODEL THEN WE NEED TO TRAVEL FROM ROOT LEVEL TO CHILD LEVEL WHICH WILL TIME TAKEN PROCESS.
3. THIS MODEL DESIGNED BASED ON ONE – MANY RELATION i.e EVER CHILD IS HAVING ONLY ONE PARENT SO THAT THERE IS A CHANCES TO OCCURE DATA DUPLICATE.

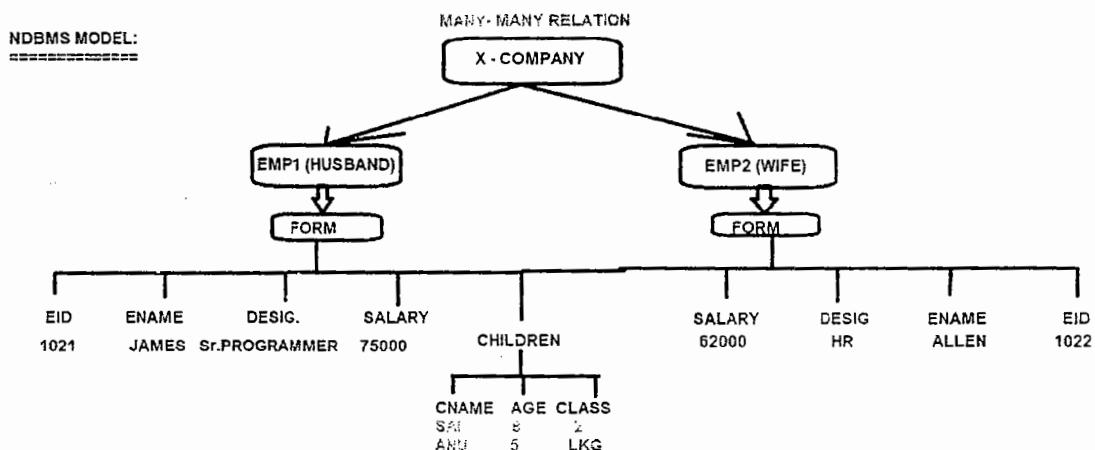
EX:



NDBMS:

THIS MODEL IS A MODIFICATION OF AN EXISTING HIERARCHICAL MODEL BRINGING "MANY - TO - MANY" RELATION SO THAT A CHILD CAN HAVE MORE THAN ONE PARENT WHICH WILL REDUCE DUPLICATE DATA. IN 1969 THE FIRST NDBMS SOFTWARE LAUNCHED WITH THE NAME AS "IDBMS" (INTEGRATED DATABASE MANAGEMENT SYSTEM).

EX:



ADVANTAGES OF NDBMS:

1. TO REDUCE DUPLICATE DATA BECAUSE SUPPORTING MANY - MANY RELATION (A CHILD CAN HAVE MULTIPLE PARENTS)
2. BY USING POINTERS MECHANISM WE CAN ADD NEW LEVEL (PARENT / CHILD) TO AN EXISTING STRUCTURE WITHOUT RECONSTRUCTION.
3. ACCESSING DATA IN THIS MODEL IS VAERY FAST BECAUSE IT USES POINTERS.

DISADVANTAGES OF NDBMS:

1. WHEN WE USE NUMBER OF POINTERS IN AN APPLICATION THEN IT WILL INCREASE COMPLEXITY(DIFFICULT) TO IDENTIFYING WHICH POINTER IS BELONGS TO WHICH PARENT OR WHICH CHILD AND ALSO DEGRADE PERFORMANCE.
2. NDBMS MODEL WAS NOT MORE SUCCESSFUL MODEL IN REAL TIME BECAUSE IMMEDIATE TAKE OVER BY RDBMS MODEL IN 1970's WITH EFFECTIVE FEATURES.

RDBMS:

IN HDBMS AND NDBMS DATA IS ORGANIZED IN THE FORM OF A TREE STRUCTURE WHICH IS LOOKS COMPLEX TO MANAGE AND UNDERSTAND ALSO SO TO OVERCOME THIS PROBLEM IN 1970's Mr.E.F.CODD FROM IBM CAME WITH A NEW CONCEPT ON STORING DATA IN A TABLE STRUCTURE i.e. ROWS AND COLUMNS FORMAT.

E.F.CODD WITH ALL THESE IDEAS FOR THE NEW MODEL CALLED AS "RELATIONAL MODEL" HAS PUBLISHED AN ARTICLE WITH THE TITLE AS "A RELATIONAL MODEL OF DATA FOR LARGE SHARED DATA BANK".

BASING ON THIS ABOVE ARTICLE MANY COMPANIES CAME FORWARD LIKE IBM, RELATIONAL SOFTWARE INC (PRESENT IT IS ORACLE COR.).....etc. HAS STARTED THE DESIGNED FOR THE NEW DATABASE MODEL i.e. RDBMS.

RDBMS IS MAINLY BASED ON TWO MATHEMATICAL PRINCIPLES ARE "RELATIONAL ALGEBRA" AND "CALCULATIONS".IN THE YEAR 1970's IBM HAS GIVEN THE PROTOTYPE FOR RDBMS KNOWN AS "SYSTEM R".

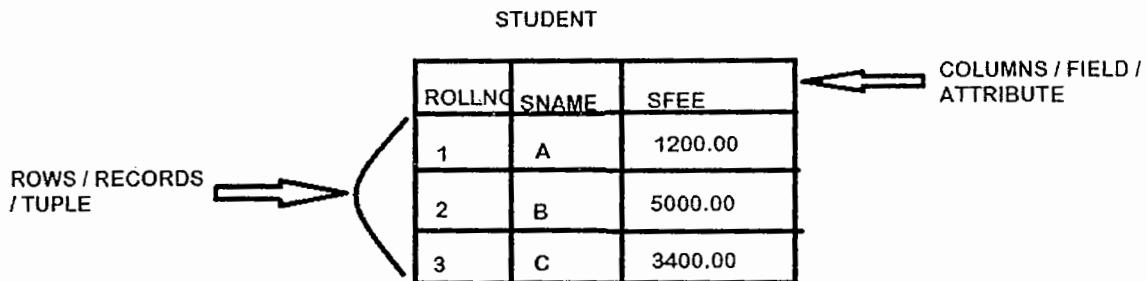
IN THE YEAR 1974 IBM HAS LAUNCHED A LANGUAGE FOR COMMUNICATION WITH RDBMS KNOWN AS "SEQUEL" AND LATER CHANGED AS "SQL".

FEATURES OF RDBMS:

- DATA CAN BE ORGANIZED IN TABLE FORMAT.
- MORE SECURITY WITH THE HELP OF "AUTHENTICATION & AUTHORIZATION".
- REDUCE DATAREDUNDANCY & DATAINCONSISTANCY USING NORMALIZATION.
- EASY TO MANIPULATION DATA USING DML COMMANDS.
- EASY TO ACCESS DATA FROM DB WITH THE HELP OF "SQL QUERY (SELECT)".
- FASTLY RETRIEVE DATA USING "INDEXES ".
- DATA SHARING USING "VIEWS ".
- SUPPORTING TRANSACTIONS WITH "ACID PROPERTIES".
- SUPPORTING DATATYPES, OPERATORS, FUNCTIONS / PROCEDURE, CLAUSES .etc
- SUPPORTING ALL RELATIONSHIPS THOSE ARE "ONE – ONE","ONE – MANY /MANY – ONE" AND "MANY-MANY".
- SUPPORTING DATAINTEGRITY RULES WITH CONSTRAINTS & TRIGGERS
- SUPPORTING SQL & PL/SQL LANGUAGES.

EX.OF AN RDBMS PRODUCTS:

ORACLE, SQL SERVER, MYSQL, DB2, SYBASE, INFORMIX, INGRES, TERADATA,
MAXDB, POSTGRESQL ...etc



DATABASE : COLLECTION OF TABLES

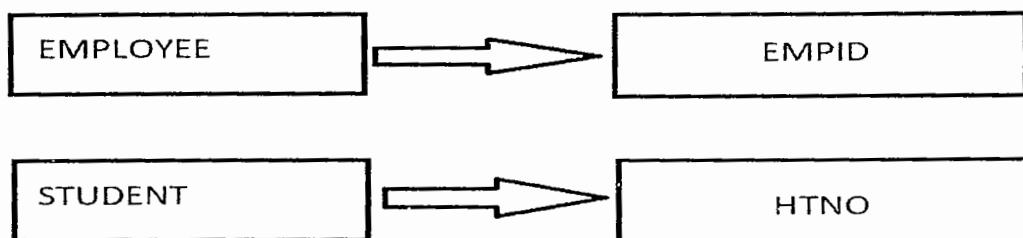
TABLE : COLLECTION OF ROWS AND COLUMNS

ROW : COLLECTION OF COLUMNS

- Here relation can be defined as commonness between objects these relations are classified into 3 types
 - One to One relation
 - One to Many relation / Many to One relation
 - Many to Many relation

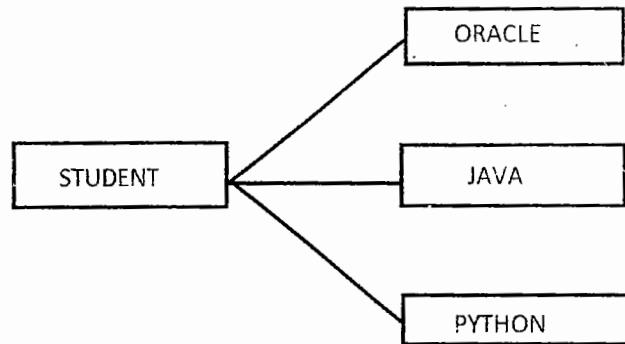
One – One relationship:

- In this relationship one object can have a relationship with another object



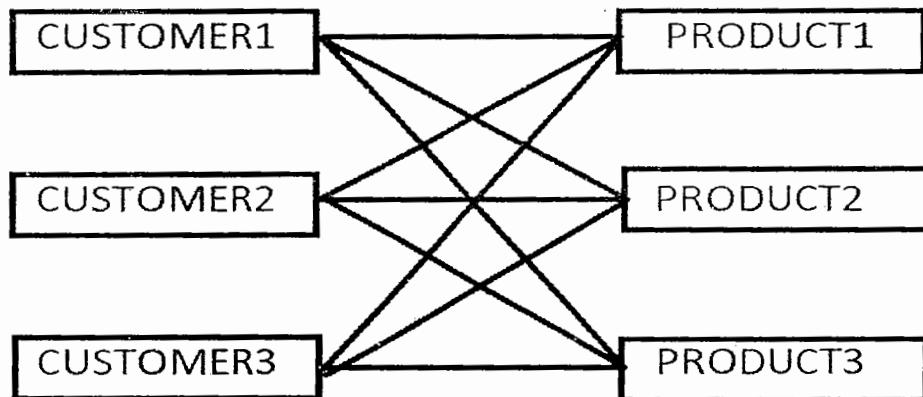
One – Many / Many – One relationship:

- In this relationship one object can have a relationship with many objects



Many – Many relationship:

- In this relationship many vendors (or) many objects can have the relationship with many other objects



INTRODUCTION TO ORACLE:

ORACLE IS A RELATIONAL DATABASE AN RDBMS PRODUCT FROM ORACLE CORPORATION IN 1979.WHICH IS USED TO STORE DATA (OR) INFORMATION PERMANANTLY i.e., IN HARD DISK ALONG WITH SECURITY.

ORACLE IS A PLATFORM INDEPENDENT AN RDBMS PRODUCT.IT MEANS THAT IT CAN DEPLOYEE (INSTALL) IN ANY OS LIKE WINDOWS, LINUX, UNIX, SOLARIES, MAC.... etc.

PLATFORM:

- IT A COMBINATION OF OPERATING SYSTEM AND MICRO PROCESSOR.THESE ARE AGAIN CLASSIFIED INTO TWO TYPES.

1) PLATFORM INDEPENDENT:

- IT SUPPORTS ANY OS WITH THE COMBINATION OF ANY MICRO PROCESSOR.

EX: ORACLE, MYSQL, JAVA, .NET.... etc.

2) PLATFORM DEPENDENT:

- IT SUPPORTS ONLY ONE OS WITH COMBINATION OF ANY MICRO PROCESSOR.

EX: C - LANGUAGE.

Versions of ORACLE:

Year	Version	Features
1979	Oracle 1.0	Not Public released
1980	Oracle 2.0	First Public released, Basic SQL functionalities.
1982	Oracle 3.0	First Portable DB.
1984	Oracle 4.0	Introduced read consistency.
1986	Oracle 5.0	Introduced client-server

		architecture.
1988	Oracle 6.0	Introduced PL/SQL
1992	Oracle 7.0	Integrity Constraints introduced, Varchar data type changed into Varchar2, Stored procedures, functions and triggers
1997	Oracle 8.0	Object Oriented Features, Table partitioning, Instead Triggers
1998	Oracle 8i(Internet)	Rollup, cube methods, Columns increased per a table up to 1000
2001	Oracle 9i	Renaming Column, Ansi Joins
2004	Oracle 10g(grid technologies)	Introduced Admin side operations, flashback query, Indicate of clauses, regular expressions
2007	Oracle 11g	Read only tables, virtual tables, integer data type, using sequence, enables and disables triggers.
2013	Oracle12c (cloud technology)	Truncate table cascade, multiple indexes, invisible column, sequence session, new auto increment by

		using Identity.
2018	Oracle18c	Polymorphic Table Functions, Active Directory Integration
2019	Oracle19c	Active Data Guard DML Redirection, Automatic Index Creation,SQL Queries on Object Stores.

WORKING WITH ORACLE:

**WHEN WE INSTALL ORACLE SOFTWARE INTERNALLY TWO
COMPONENTS ARE INSTALLED.THOSE ARE,**

1. ORACLE CLIENT

2. ORACLE SERVER

1. ORACLE CLIENT:

**BY USING ORACLE CLIENT TOOL USER CAN PERFORM THE
FOLLOWING THREE OPERATIONS ARE**

- **USER CAN CONNECT TO ORACLE SERVER**
- **USER CAN SEND REQUEST TO ORACLE SERVER**
- **USER CAN RECEIVE RESPONSE FROM ORACLE SERVER.**

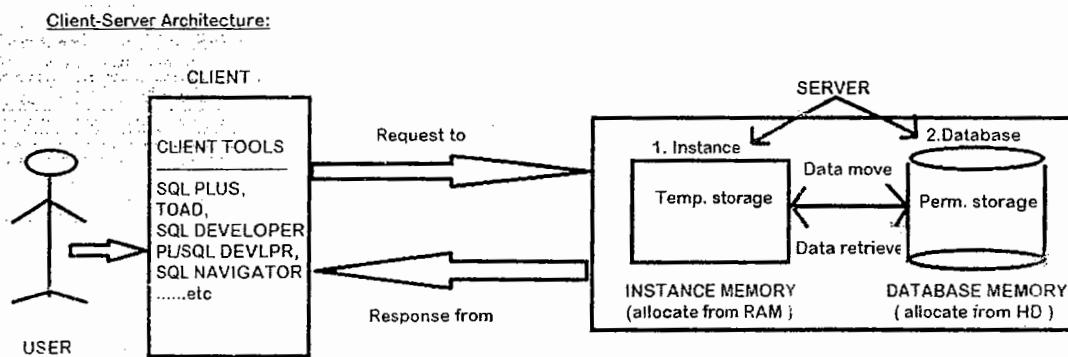
**Ex: SQLPLUS, TOAD, SQL DEVELOPER, SQL NAVIGATOR.....
etc.**

2. ORACLE SERVER:

ORACLE SERVER MANAGE TWO MORE SUB COMPONENTS INTERNALLY THOSE ARE,

- INSTANCE
- DATABASE

INSTANCE WILL ACT AS TEMPORARY MEMORY WHICH WILL ALLOCATE FROM RAM AND STORED DATA / INFORMATION TEMPORARY WHERE AS DATABASE IS A PERMANENT MEMORY WHICH WILL ALLOCATE FROM HARDDISK AND STORED DATA PERMANENTLY.



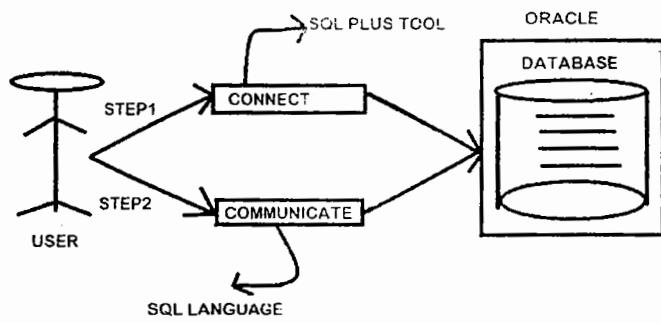
NOTE: WHEN WE WANT TO WORK ON ORACLE DATABASE THEN WE FOLLOW THE FOLLOWING TWO STEPS PROCEDURE

1) CONNECT TO ORACLE:

IF USER WANTS TO CONNECT TO ORACLE THEN WE REQUIRED A DATABASE TOOL IS CALLED AS "SQLPLUS" WHICH WAS INBUILTED IN ORACLE SOFTWARE.

2) COMMUNICATE WITH DATABASE:

IF USER WANTS TO COMMUNICATE WITH DATABASE THEN WE NEED A DATABASE COMMUNICATION LANGUAGE IS CALLED AS "SQL".



HOW TO CONNECT TO ORACLE:

BEFORE CONNECT TO ORACLE DATABASE WE NEED TO KNOW THE TYPES OF EDITIONS IN ORACLE SOFTWARE.EVERY ORACLE SOFTWARE IS HAVING TWO TYPES OF EDITIONS THOSE ARE

- 1) ORACLE EXPRESS EDITION (PARTIAL SUPPORTING FEATURES)**
- 2) ORACLE ENTERPRISE EDITION (FULLY SUPPORTING FEATURES)**

THE ABOVE TWO EDITIONS ARE HAVING DEFAULT USERNAME IS "SYSTEM" AND PASSWORD IS CREATED AT INSTALLATION OF ORACLE SOFTWARE.

STEPS TO CONNECT TO ORACLE:

- > GO TO ALL PROGRAMS**
- > GO TO ORACLE19c_HOME1 FOLDER**
- > CLICK ON SQL PLUSE ICON**
- > ENTER USERNAME: SYSTEM**
- > ENTER PASSWORD: MANAGER (AT INSTALLATION TIME PASSWORD)**

TO CREATE A NEW USERNAME & PASSWORD IN ORACLE DB:

SYNTAX: CREATE USER <USERNAME> IDENTIFIED BY <PASSWORD>;

EX: CREATE USER SUDHAKAR IDENTIFIED BY SUDHAKAR;

NOTE: USER IS CREATED BUT THIS USER IS DUMMY USER BECAUSE IS NOT HAVING PERMISSION TO CONNECT AND CREATE NEW TABLE IN DB.SO PERMISSIONS MUST BE GIVEN TO USER(SUDHAKAR) BY USING "GRANT" COMMAND BY DBA(SYSTEM).

NOTE: EVERY USER IN ORACLE SERVER IS CALLED AS "SCHEMA".

GRANTING PERMISSIONS TO USER:

STEP1: USERNAME: SYSTEM

PASSWORD: MANAGER

CONNECTED.

STEP2:

GRANT CONNECT, CREATE TABLE TO SUDHAKAR;

Here,

CONNECT ----- TO CONNECT TO ORACLE DB

CREATE TABLE ----- TO CREATE NEW TABLES IN DB.

NOTE: WHEN WE CONNECT TO ORACLE DB SOME TIMES, WE WILL FACE A PROBLEM IS,

ERROR: ORA-28000: THE ACCOUNT IS LOCKED.

TO OVERCOME THE ABOVE ERROR THEN WE FOLLOW THE FOLLOWING STEPS ARE

SOLUTION:

STEP1: CONNECT TO ORACLE WITH SYSTEM DATABASE ADMIN:

SYNTAX:

ENTER USERNAME: SYSTEM

ENTER PASSWORD: MANAGER

CONNECTED.

STEP2: TO UNLOCK USER:

SYNTAX:

SQL> ALTER USER <USER NAME> ACCOUNT UNLOCK / LOCK;

EX:

SQL> ALTER USER SUDHAKAR ACCOUNT UNLOCK;

**STEP3: NOW CONNECT TO ORACLE WITH EITHER SYSTEM (OR)
SUDHAKAR USER:**

ENTER USERNAME: SUDHAKAR

ENTER PASSWORD: SUDHAKAR

CONNECTED.

HOW TO CHANGE A PASSWORD:

SQL> PASSWORD

Changing password for SUDHAKAR

Old password: SUDHAKAR

New password:123

Retype new password:123

Password changed

SQL> CONN

Enter user-name: SUDHAKAR / SUDHAKAR

ERROR: ORA-01017: invalid username/password; logon denied

Warning: You are no longer connected to ORACLE.

SQL> CONN

Enter user-name: SUDHAKAR / 123

Connected.

HOW TO CREATE A NEW PASSWORD IF WE FORGOT A PASSWORD:

SYNTAX:

**ALTER USER <USER NAME> IDENTIFIED BY
<NEW PASSWORD>;**

EX:

Enter user-name: SYSTEM / MANAGER

Connected.

SQL> ALTER USER SUDHAKAR IDENTIFIED BY SUDHAKAR;

User altered.

SQL> CONN

Enter user-name: SUDHAKAR / 123

ERROR:

ORA-01017: invalid username/password; logon denied

Warning: You are no longer connected to ORACLE.

SQL> CONN

Enter user-name: SUDHAKAR / SUDHAKAR

CONNECTED.

**NOTE: WHEN WE WANT TO CONNECT TO ORACLE DB SERVER
SOME TIMES, WE FACED ANOTHER PROBLEM IS CALLED AS**

"TNS PROTOCOL ADAPTER ERROR".

Enter user-name: SUDHAKAR / SUDHAKAR

ERROR:

ORA-12560: TNS: protocol adapter error

Enter user-name: SYSTEM / MANAGER

ERROR:

ORA-12560: TNS: protocol adapter error

NOTE: TO OVERCOME THE ABOVE PROBLEM THEN WE FOLLOW THE FOLLOWING STEPS ARE,

STEP1: GO TO SERVICES

STEP2: GO TO ORACLESERVICEorcl AND CLICK ON IT

STEP3: SELECT STARTUP TYPE IS AUTOMATIC

STEP4: CLICK ON START BUTTON

STEP5: CLICK ON OK

Enter user-name: SYSTEM / MANAGER

Connected to: Oracle Database 19c Enterprise Edition Release
19.0.0.0.0 - Production

Version 19.3.0.0.0

SQL> conn

Enter user-name: SUDHAKAR / SUDHAKAR

Connected.

DATATYPES:

DATATYPE IS AN ATTRIBUTE WHICH SPECIFIES WHAT TYPE OF DATA IS STORED INTO A COLUMN. ORACLE SUPPORTS THE FOLLOWING DATATYPES ARE

- 1) NUMERIC DATATYPES
- 2) CHARACTER DATATYPES/ STRING DATATYPES
- 3) LONG DATATYPE
- 4) DATE DATATYPES
- 5) RAW & LONG RAW DATATYPES
- 6) LOB DATATYPES (LARGE OBJECTS DATATYPES)

1) NUMERIC DATATYPES:

- i) INT
- ii) NUMBER (P, S)

INT: STORING INTEGER FORMAT VALUES ONLY.

INT = NUMBER (38)

NOTE: WHEN WE USE "INT" DATATYPE ON COLUMN AT THE TIME OF TABLE CREATION THEN INTERNALLY ORACLE SERVER WILL CONVERT INTO "NUMBER" DATATYPE WITH MAXIMUM SIZE IS 38 DIGITS.

NUMBER (P, S): STORING BOTH INTEGER & FLOAT FORMAT VALUES. HERE THIS DATATYPE IS HAVING FOLLOWING TWO ARGUMENTS ARE PRECISION(P), SCALE(S).

WHEN WE USE ----> NUMBER(P)-----> STORE INTEGER VALUES

WHEN WE USE ----> NUMBER (P, S) -----> STORE FLOAT VALUES

PRECISION(P):

> COUNTING ALL DIGITS INCLUDING LEFT & RIGHT SIDES OF GIVEN FLOAT EXPRESSION.

Ex: 25.12

PRECISION = 4

Ex: 856.45

PRECISION = 5

Ex: 9999.99

PRECISION = 6

SCALE(S):

> COUNTING ONLY RIGHT DIGITS OF A FLOAT EXPRESSION.

Ex: 25.12

SCALE = 2

PRECISION = 4

Ex:

7456.123

SCALE = 3

PRECISION = 7

2) CHARACTER DATATYPES: STORING "STRING" FORMAT DATA ONLY. IN DATABASE STRING IS REPRESENT WITH SINGLE QUOTES '**<STRING>**'. CHARACTER DATATYPES ARE STORING TWO TYPES STRING FORMAT DATA. THOSE ARE

- i) CHARACTERS ONLY STRING DATA ii) ALPHANUMERIC STRING DATA.

Ex:

CHARACTER DATATYPES

< STRING FORMAT DATA >

CHARACTERS ONLY

ALPHANUMERIC CHAR's

STRING DATA

STRING DATA

[A - Z (or) a - z]

[A - Z (or) a - z & 0 - 9 &

@, #, \$, %, &, _,etc]

Ex: 'SAI', 'ALLEN ',etc

Ex: 'sai123@gmail.com ',etc.

NOTE: CHARACTER DATATYPES ARE AGAIN CLASSIFIED INTO TWO CATEGORIES THOSE ARE

1) NON - UNICODE DATATYPES: SUPPORTING TO STORE LOCALIZED DATA (ONLY ENGLISH LANGUAGE) THESE ARE AGAIN TWO TYPES.

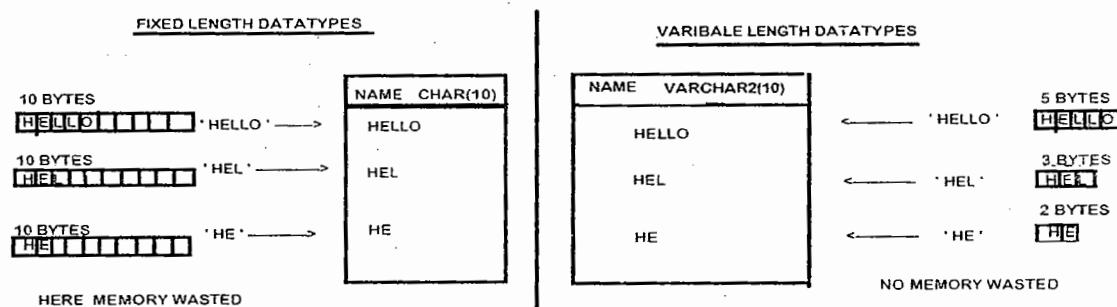
1) CHAR(SIZE):

- IT IS A FIXED LENGTH DATATYPE(STATIC).
- WILL STORE NON-UNICODE CHAR's IN THE FORM OF 1 CHAR = 1 BYTE.
- MAXIMUM SIZE OF CHAR DATATYPE IS 2000 BYTES (2000 CHAR's).

2) VARCHAR2(SIZE):

- IT IS A VARIABLE LENGTH DATATYPE(DYNAMIC).
- WILL STORE NON-UNICODE CHAR's IN THE FORM OF 1 CHAR = 1 BYTE.
- MAXIMUM SIZE OF VARCHAR2 DATATYPE IS 4000 BYTES (4000 CHAR's).

DIFFERENCES BETWEEN FIXED LENGTH DATATYPES AND VARIABLE LENGTH DATATYPES:



2) UNICODE DATATYPES: THESE DATATYPES ARE STORING "GLOBALIZED DATA" SUPPORTING "ALL NATIONAL LANGUAGES". THESE ARE TWO TYPES,

1) NCHAR(SIZE):

- IT IS FIXED LENGTH DATATYPE(STATIC).
- WILL STORE UNICODE CHAR's (ALL NATIONAL LANG's) IN THE FORM OF 1 CHAR = 1 BYTE.
- MAXIMUM SIZE OF NCHAR DATATYPE IS 2000 BYTES (2000 CHAR's).

2) NVARCHAR2(SIZE):

- IT IS A VARIABLE LENGTH DATATYPE(DYNAMIC).
- WILL STORE UNICODE CHAR's IN THE FORM OF 1 CHAR = 1 BYTE.
- MAXIMUM SIZE OF NVARCHAR2 DATATYPE IS 4000 BYTES (4000 CHAR's).

3) LONG:

- IT IS VARIABLE LENGTH DATATYPE(DYNAMIC).
- WILL STORE NON-UNICODE & UNICODE CHAR's IN THE FORM OF 1 CHAR = 1BYTE.
- MAXIMUM SIZE OF LONG DATATYPE IS 2GB.

4) DATE DATATYPES:

- STORING DATE AND TIME INFORMATION OF A PARTICULAR DAY.
- RANGE OF DATE DATATYPES IS FROM "01-JAN-4712 BC " TO "31-DEC-9999 AD ".

i) **DATE:** STORING DATE & TIME INFORMATION BUT TIME IS OPTIONAL.IF NOT ENTERED TIME BY USER, THEN ORACLE "12:00:00AM".DEFAULT FORMAT OF DATE DATATYPE IS 'DD-MON-YYYY / YY HH: MI: SS'.IT OCCUPIED 7 BYTES OF MEMORY (FIXED MEMORY).

ii) **TIMESTAMP:** STORING DATE & TIME INFORMATION ALONG WITH MILLISECONDS.DEFAULT FORMAT OF DATE DATATYPE IS 'DD-MON-YYYY / YY.HH: MI: SS.MS'. IT OCCUPIED 11 BYTES OF MEMORY (FIXED MEMORY).

5) RAW & LONG RAW: STORING IMAGE / AUDIO / VIDEO FILES IN THE FORM OF 010010101001 BINARY FORMAT.THE MAXIMUM SIZE OF RAW DATATYPE IS 2000 BYTES AND LONG RAW SIZE IS 2 GB.

6) LOB DATATYPES (LARGE OBJECTS):

i) **BLOB (BINARY LARGE OBJECT):** STORING IMAGE / AUDIO / VIDEO FILES IN THE FORM 010010101001 BINARY FORMAT.THE MAXIMUM SIZE IS 4GB.

ii) **CLOB (CHARACTER LARGE OBJECT):** STORING NON-UNICODE CHARACTERS.THE MAXIMUM SIZE IS 4GB.

iii) **NCLOB (NATIONAL CHARACTERS LARGE OBJECT):** STORING UNICODE CHARACTERS.THE MAXIMUM SIZE IS 4GB.

NOTE:

- | | |
|-------------------------------|--------------------------------|
| > CHAR IS UPTO 2000 BYTES | > NCHAR IS UPTO 2000 BYTES |
| > VARCHAR2 IS UPTO 4000 BYTES | > NVARCHAR2 IS UPTO 4000 BYTES |
| > CLOB IS UPTO 4 GB | > NCLOB IS UPTO 4 GB |
| > RAW IS UPTO 2000 BYTES | > LONG RAW IS UPTO 2GB |
| > BLOB IS UPTO 4 GB. | |

STRUCTURE QUERY LANGUAGE(SQL): SQL IS A DATABASE LANGUAGE WHICH WAS USED TO COMMUNICATE WITH DATABASE. INTRODUCED BY "IBM" AND INITIAL NAME OF THIS LANGUAGE WAS "SEQUEL" AND LATER RENAMED WITH "SQL".

"SQL" IS ALSO CALLED AS "CLI"(COMMON LANGUAGE INTERFACE) BECAUSE THIS IS THE LANGUAGE WHICH IS USED TO COMMUNICATE WITH ANY RDBMS PRODUCTS SUCH AS ORACLE, SQLSERVER, MYSQL, DB2.....etc.

SQL PRE-DEFINE QUERIES ARE NOT A CASE - SENSITIVE (WRITE QUERIES IN EITHER UPPER & LOWER-CASE CHARACTERS) BUT EVERY SQL QUERY SHOULD ENDS WITH ";".

SUB - LANGUAGES OF SQL:

1) DDL (DATA DEFINITION LANGUAGE):

- > CREATE, ALTER, RENAME, TRUNCATE, DROP
- > RECYCLEBIN, FLASHBACK, PURGE (LATEST FEATURES)

2) DML (DATA MANIPULATION LANGUAGE):

- > INSERT, UPDATE, DELETE
- > INSERT ALL, MERGE (NEW COMMANDS)

3) DQL / DRL (DATA QUERY / DATA RETRIVE LANGUAGE):

- > SELECT

4) TCL (TRANSACTION CONTROL LANGUAGE):

- > COMMIT, ROLLBACK, SAVEPOINT

5) DCL (DATA CONTROLL LANGUAGE):

- > GRANT, REVOKE

1) DDL (DATA DEFINITION LANGUAGE):

- > CREATE
- > ALTER
- > RENAME
- > TRUNCATE
- > DROP

1. CREATE: CREATE A NEW TABLE IN ORACLE DB.

SYNTAX:

```
CREATE TABLE <TABLE NAME> (<COLUMN NAME1> <DATATYPE>[SIZE],  
<COLUMN NAME2> <DATATYPE>[SIZE],  
.....);
```

EX:

```
CREATE TABLE STUDENT (STID INT, SNAME CHAR (10), SFEE NUMBER  
(6,2));
```

TO VIEW THE STRUCTURE OF A TABLE IN ORACLE DB:

SYNTAX:

```
SQL> DESC <TABLE NAME>;
```

EX:

```
SQL> DESC STUDENT;
```

2. ALTER:

IT IS USED TO MODIFY THE STRUCTURE OF A TABLE IN DATABASE. THIS COMMAND IS HAVING THE FOLLOWING FOUR SUB COMMANDS ARE

i) ALTER - MODIFY: TO CHANGE DATATYPE AND ALSO SIZE OF DATATYPE OF A PARTICULAR COLUMN.

SYNTAX:

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



EX:

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

ii) ALTER - ADD: ADDING 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(30);

iii) ALTER - RENAME: TO CHANGE A COLUMN NAME IN A TABLE.

SYNTAX:

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

EX:

SQL> ALTER TABLE STUDENT RENAME COLUMN SNAME TO STUDENTNAMES;

iv) ALTER - DROP: TO DROP AN EXISTING COLUMN FROM A TABLE.

SYNTAX:

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

EX:

SQL> ALTER TABLE STUDENT DROP COLUMN SFEE;

3. RENAME: IT IS USED TO CHANGE A TABLE NAME IN DATABASE.

SYNTAX:

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

EX:

SQL> RENAME STUDENT TO STUDENTDETAILS;

4. TRUNCATE: TO DELETE ALL ROWS FROM A TABLE AT A TIME. BY USING TRUNCATE COMMAND, WE CANNOT DELETE A SPECIFIC ROW FROM A TABLE BECAUSE TRUNCATE DOES NOT SUPPORTS "WHERE CLAUSE" CONDITION. IS DELETING ROWS BUT NOT COLUMNS OF A TABLE.

SYNTAX:

TRUNCATE TABLE <TABLE NAME>;

EX:

SQL> TRUNCATE TABLE STUDENT;

5. DROP: TO DROP A TABLE (i.e., ROWS AND COLUMNS) FROM DATABASE.

SYNTAX:

DROP TABLE <TABLE NAME>;

EX:

SQL> DROP TABLE STUDENT;

NOTE: FROM ORACLE 10g ENTERPRISE EDITION ONCE WE DROP A TABLE FROM DATABASE THEN IT WILL DROP TEMPORARILY. AND USER HAS A CHANCE TO RESTORE DROPPED TABLE AGAIN INTO DATABASE BY USING THE FOLLOWING COMMANDS ARE,

1) RECYCLEBIN

2) FLASHBACK

3) PURGE

1) RECYCLEBIN: IT IS A PRE-DEFINE TABLE WHICH IS USED TO STORE INFORMATION ABOUT DROPPED TABLES. IT WILL WORK AS A WINDOWS RECYCLEBIN IN SYSTEM.

HOW TO VIEW THE STRUCTURE OF RECYCLEBIN:

SYNTAX:

SQL> DESC RECYCLEBIN;

Name	Null?	Type
OBJECT_NAME	NOT NULL	VARCHAR2(30)
ORIGINAL_NAME		VARCHAR2 (32)

HOW TO VIEW INFORMATION ABOUT DROPPED TABLES IN RECYCLEBIN:

SYNTAX:

SQL> SELECT OBJECT_NAME, ORIGINAL_NAME FROM RECYCLEBIN;

OBJECT_NAME	ORIGINAL_NAME
-------------	---------------

BIN\$EenuRFtsT7ahnHV7rbI71Q==\\$0	STUDENT
-----------------------------------	---------

2) FLASHBACK: THIS COMMAND IS USED TO RESTORE A DROPPED TABLE FROM RECYCLEBIN.

SYNTAX:

SQL> FLASHBACK TABLE <TABLE NAME> TO BEFORE DROP;

EX:

SQL> FLASHBACK TABLE STUDENT TO BEFORE DROP;

PURGE: THIS COMMAND IS USED TO DROP A TABLE FROM RECYCLEBIN PERMANENTLY (OR) TO DROP A TABLE FROM DATABASE PERMANENTLY.

SYNTAX1: (DROPPING A SPECIFIC TABLE FROM RECYCLEBIN)

SQL> PURGE TABLE <TABLE NAME>;

EX:

SQL> PURGE TABLE TEST1;

SYNTAX2: (DROPPING ALL TABLES FROM RECYCLEBIN)

SQL> PURGE RECYCLEBIN;

EX:

SQL> PURGE RECYCLEBIN;

SYNTAX3: (DROP A TABLE FROM DATABASE PERMANENTLY)

SQL> DROP TABLE <TABLE NAME> PURGE;

EX:

SQL> DROP TABLE STUDENT PURGE;

2) DML (DATA MANIPULATION LANGUAGE)

> INSERT

> UPDATE

> DELETE

1. INSERT: INSERTING A NEW ROW DATA INTO A TABLE.

SYNTAX1:

INSERT INTO <TN> VALUES (VALUE1, VALUE2,);

EX:

**SQL> CREATE TABLE STUDENT (STID INT, SNAME VARCHAR2(10), SFEE
NUMBER (10));**

SQL> INSERT INTO STUDENT VALUES (1021,'SAI',2500);

1 row created.

**NOTE: IN THIS METHOD WE SHOULD INSERT VALUES TO ALL COLUMNS IN
A TABLE.**

SYNTAX2:

**INSERT INTO <TN> (REQ. COLUMN NAMES) VALUES (VALUE1, VALUE2,
.....);**

EX:

SQL> INSERT INTO STUDENT (STID, SNAME) VALUES (1022,'SMITH');

1 row created.

NOTE: IN THIS METHOD WE CAN INSERT VALUES FOR REQUIRED COLUMNS ONLY.AND REMAINING COLUMNS WILL TAKE "NULL" BY DEFAULT.

How to insert NULLs into a table:

METHOD1:

INSERT INTO EMP VALUES (NULL, NULL, NULL);

METHOD2:

INSERT INTO EMP (EID, ENAME, EADDRESS) VALUES (NULL, NULL, NULL);

SUBSTITUTIONAL OPERATORS: THESE OPERATORS ARE USED TO INSERT MULTIPLE ROWS DATA INTO A TABLE CONTINUALLY.THESE ARE TWO TYPES,

i) &: WE CAN INSERT VALUES TO COLUMNS DYNAMICALLY.

ii) &&: WE CAN INSERT VALUES TO COLUMNS IN FIXED MANNER.IF WE WANT CHANGE A FIXED VALUE OF COLUMN THEN WE SHOULD "EXIT" FROM ORACLE DATABASE.

SYNTAX1 (&):

INSERT INTO <TN> VALUES (&<COLUMN NAME1>,&<COLUMN NAME2>,.....);

EX:

SQL> INSERT INTO STUDENT VALUES (&STID,'&SNAME',&SFEE);

Enter value for stid: 1023

Enter value for sname: ALLEN

Enter value for sfee: 1500

SQL> / ----- →(HERE " / " IS USED TO RE-EXECUTE THE LAST EXECUTED SQL QUERY IN SQLPLUS EDITOR)

Enter value for stid: 1024

Enter value for sname: WARD

Enter value for sfee: 4500

SYNTAX2 (&):

```
INSERT INTO <TN> (REQ.COLUMN NAMES) VALUES (&<COLUMN NAME1>,
.....);
```

EX:

```
SQL> INSERT INTO STUDENT (STID)VALUES(&STID);
```

Enter value for stid: 1026

```
SQL> /
```

Enter value for stid: 1027

```
SQL> /
```

Enter value for stid: 1028

SYNTAX1 (&&):

```
INSERT INTO <TN> VALUES (&&<COLUMN NAME1>,&&<COLUMN
NAME2>,.....);
```

EX:

```
SQL> INSERT INTO STUDENT VALUES (&STID,'&SNAME',&&SFEE);
```

Enter value for stid: 1029

Enter value for sname: SCOTT

Enter value for sfee: 8000

```
SQL> /
```

Enter value for stid: 1030

Enter value for sname: WARNER

```
SQL> /
```

.....

.....

SYNTAX2 (&&):

INSERT INTO <TN>(REQ.COLUMN NAMES)VALUES(&&<COLUMN NAME1>,.....);

**UPDATE: UPDATING ALL ROWS DATA AT A TIME IN A TABLE (OR)
UPDATING A SINGLE ROW DATA IN A TABLE BY USING "WHERE CLAUSE" CONDITION.**

SYNTAX:

UPDATE <TN> SET <COLUMN NAME1> = <VALUE1>,<COLUMN NAME2>=<VALUE2>,.....[WHERE <CONDITION>];

EX1:

SQL> UPDATE STUDENT SET SNAME='JONES', SFEE=6500 WHERE STID=1027;

SQL> UPDATE EMP SET COMM=500;

SQL> UPDATE EMP SET SAL=NULL;

SQL> UPDATE EMP SET SAL=5000 WHERE SAL IS NULL;

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 <TN> [WHERE <CONDITION>];

EX:

SQL> DELETE FROM STUDENT WHERE STID=1023;

EX:

SQL> DELETE FROM STUDENT;

SQL> DELETE FROM EMP WHERE COMM IS NULL;

DIFFERENCE BETWEEN DELETE & TRUNCATE COMMAND:

DELETE

1. IT IS A DML COMMAND.
2. IT CAN DELETE A SPECIFIC ROW FROM A TABLE.
3. IT SUPPORTS "WHERE CLAUSE" "WHERE CONDITION."
4. IT TEMPORARY DATA DELETION.
5. WE CAN RESTORE DELETED DATA BY USING "ROLLBACK" COMMAND. "ROLLBACK".
6. EXECUTION SPEED IS SLOW.

FAST

(DELETING ROWS IN ONE BY ONE MANNER)

TRUNCATE

1. IT IS A DDL COMMAND.
2. IT CANNOT DELETE ALL ROWS FROM A TABLE.
3. IT DOES NOT SUPPORTS CLAUSE" CONDITION.
4. IT IS PERMANENT DATA DELETION.
5. WE CANNOT RESTORE DATA BY USING
6. EXECUTION SPEED IS

Fast

(DELETING GROUP OF ROWS AT A TIME)

3) DQL / DRL (DATA QUERY LANGUAGE / DATA RETRIEVE LANGUAGE):

> SELECT

SELECT: TO RETRIEVE ALL ROWS FROM A TABLE AT A TIME (OR) TO RETRIEVE A SPECIFIC ROW FROM A TABLE BY USING "WHERE CLAUSE" CONDITION.

SYNTAX:

SELECT * FROM <TABLE NAME> [WHERE <CONDITION>];

Here, " * " IS REPRESENT ALL COLUMNS IN A TABLE.

EX:

SQL> SELECT * FROM DEPT;

(OR)

SQL> SELECT DEPTNO, DNAME, LOC FROM DEPT;

EX:

SQL> SELECT * FROM EMP WHERE JOB='CLERK';

SQL> SELECT * FROM EMP WHERE COMM IS NULL;

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

TO VIEW ALL LIST OF TABLES IN ORACLE DATABASE:

SYNTAX:

SQL> SELECT * FROM TAB;

TO VIEW DATA OF A PARTICULAR TABLE:

SYNTAX:

SQL> SELECT * FROM <TABLE NAME>;

EX:

SQL> SELECT * FROM EMP;

NOTE: WHEN WE WANT TO DISPLAY THE INFORMATION / DATA OF A PARTICULAR TABLE IN PROPER SYSTEMATICALLY THEN WE NEED TO SET THE FOLLOWING TWO PROPERTIES ARE,

1) PAGESIZE n:

- NUMBER OF ROWS DISPLAYED PER A PAGE. HERE "n" IS REPRESENTED NO. OF ROWS. BY DEFAULT, A SINGLE PAGE IS DISPLAYED 14 ROWS. RANGE IS 0 TO 50000.

SYNTAX:

SQL> SET PAGESIZE n;

EX:

SQL> SET PAGESIZE 100;

2) LINES n:

- NUMBER OF BYTES IN A SINGLE LINE. HERE "n" IS REPRESENT NO. OF BYTES. RANGE IS 1 TO 32767.

SYNTAX:

SQL> SET LINES n;

EX:

SQL> SET LINES 100;

TO CLEAR SQL PLUS EDITOR SCREEN:

SYNTAX:

SQL> CL SCR;

(OR)

SHIFT+DELETE (FROM KEYBOARD)

TO DISCONNECT / EXIT FROM ORACLE DATABASE:

SQL> EXIT;

ALIAS NAMES: IT IS AN ALTERNATE (OR) TEMPORARY NAME. USER CAN CREATE ALIAS NAMES ON TWO LEVELS IN DB.

i) COLUMN LEVEL:

- IN THIS LEVEL WE ARE CREATING ALIAS NAMES ON COLUMNS.

SYNTAX:

<COLUMN NAME> <COLUMN ALIAS NAME>

EX:

DEPTNO X

ii) TABLE LEVEL:

- IN THIS LEVEL WE ARE CREATING ALIAS NAMES ON TABLE.

SYNTAX:

<TABLE NAME> <TABLE ALIAS NAME>

EX:

DEPT D

SYNTAX TO COMBINE COLUMN + TABLE LEVEL ALIAS NAMES BY USING "SELECT" QUERY:

SELECT <COLUMN NAME1> <COLUMN NAME1 ALIAS NAME>, <COLUMN NAME2> <COLUMN NAME2 ALIAS NAME>, FROM <TN> <TABLE ALIAS NAME>;

EX:

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

CONCATENATION OPERATOR (||):

- THIS OPERATOR IS USED TO JOIN TWO STRING VALUES (OR) TWO EXPRESSIONS IN A SELECT QUERY.

EX:

SQL> SELECT 'WELCOME'||' '| '|TO ORACLE' FROM DUAL;

OUTPUT:

WELCOME TO ORACLE

EX:

SQL> SELECT 'Mr.'||ENAME||' '| '|IS WORKING AS A'||'|JOB FROM EMP;

OUTPUT:

Mr. SMITH IS WORKING AS A CLERK

DISTINCT KEYWORD:

- THIS KEYWORD IS USED TO ELIMINATE DUPLICATE VALUES AND DISPLAY

UNIQUE VALUES IN QUERY RESULT.

EX:

SQL> SELECT DISTINCT JOB FROM EMP;

SQL> SELECT DISTINCT DEPTNO FROM EMP;

How to create a new table from the old table:

syntax1:

create table <new table name> as select * from <old table name>;

EX:

CREATE TABLE NEWEMP AS SELECT * FROM EMP;

NOTE: created a new table with copy of all rows & columns from the old table.

syntax2:

create table <new table name> as select * from <old table name> where <false condition>;

Ex:

CREATE TABLE DUMMYEMP AS SELECT * FROM EMP WHERE 1=2;

NOTE: created a new table without copy rows from old table. (Columns copy)

EX:

CREATE TABLE SPECEMP AS SELECT EID, EADDRESS FROM EMP;

NOTE: created a new table with specific columns from the old table.

EX:

CREATE TABLE SPECROWS AS SELECT * FROM EMP WHERE EADDRESS='HYD';

NOTE: created a new table with specific rows from the old table.

How to copy data from one table to another table:

syntax:

insert into <destination table name> select * from <source table name>;

EX:

CREATE TABLE DESTEMP (EMPNO INT, NAME CHAR (10), LOC VARCHAR2(10));

INSERT INTO DESTEMP SELECT * FROM EMP;

EX:

```
CREATE TABLE DEMP AS SELECT * FROM EMP WHERE 1=0;
```

INSERT INTO DEMP SELECT * FROM EMP;

INSERT ALL:

- IT IS A DML COMMAND (ORACLE 9i). WHICH IS USED TO INSERT ROWS INTO MULTIPLE TABLES AT A TIME.BUT THE ROWS SHOULD BE AN EXISTING TABLE.

SYNTAX:

INSERT ALL INTO <TN1> VALUES (<COLUMN NAME1>, <COLUMN NAME2>,)

INTO <TN2> VALUES (<COLUMN NAME1>, <COLUMN NAME2>,)

INTO <TN3> VALUES (<COLUMN NAME1>, <COLUMN NAME2>,)

INTO <TN n> VALUES (<COLUMN NAME1>, <COLUMN NAME2>, ...)

SELECT * FROM <OLD TABLE NAME>;

Ex-

STEP1:

SOL> SELECT * FROM DEPT; -----OLD TABLE

STEP2: CREATING EMPTY TABLES:

SOL> CREATE TABLE TEST1 AS SELECT * FROM DEPT WHERE 1=0;

SOL> CREATE TABLE TEST2 AS SELECT * FROM DEPT WHERE 1=0;

```
SOL> CREATE TABLE TEST3 AS SELECT * FROM DEPT WHERE 1=0;
```

STEP3:

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

STEP4:

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

MERGE COMMAND:

- IT IS A DML COMMAND (ORACLE 9i). IT IS USED TO TRANSFER DATA FROM SOURCE TABLE TO DESTINATION TABLE.

- IF DATA IS MATCHING IN BOTH TABLES THEN THOSE MATCHING DATA / ROWS ARE OVERRIDE ON DESTINATION TABLE BY USING "UPDATE COMMAND" WHERE AS DATA IS NOT MATCHING THEN THOSE UNMATCHING DATA / ROWS ARE TRANSFERRING FROM SOURCE TABLE TO DESTINATION TABLE BY USING "INSERT COMMAND".

SYNTAX:

```
MERGE INTO <DESTINATION TABLE NAME> <ALIAS NAME> USING
<SOURCE TABLE NAME> <ALIAS NAME> ON (<JOINING CONDITION>)
```

WHEN MATCHED THEN

```
UPDATE SET <DEST.TABLE ALIAS NAME>. <COLUMN
NAME1>=<SOUR.TABLE ALIAS NAME>. <COLUMN NAME1>,
.....
```

WHEN NOT MATCHED THEN

```
INSERT (<DESTINATION TABLE COLUMNS>) VALUES (<SOURCE TABLE
COLUMNS>);
```

EX:

STEP1:

SQL> SELECT * FROM DEPT;

STEP2:

SQL> CREATE TABLE NEWDEPT AS SELECT * FROM DEPT;

STEP3:

SQL> INSERT INTO NEWDEPT VALUES (50,'DBA','HYD');

SQL> INSERT INTO NEWDEPT VALUES (60,'SAP','MUMBAI');

STEP4:

SQL> SELECT * FROM NEWDEPT; -----SOURCE TABLE

SQL> SELECT * FROM DEPT; -----OLD TABLE

STEP5:

SQL> MERGE INTO DEPT D USING NEWDEPT S ON (D. DEPTNO=S.DEPTNO)

WHEN MATCHED THEN

UPDATE SET D. DNAME=S.DNAME, D.LOC=S.LOC

WHEN NOT MATCHED THEN

INSERT (D. DEPTNO, D. DNAME, D.LOC) VALUES (S. DEPTNO,

S. DNAME, S.LOC);

Operators in Oracle

Operators are used to express the conditions in Select statements. Operator manipulates individual data items and returns a result. The data items are called operands or arguments.

The different types of Operators available in Oracle SQL are:-

- Arithmetic operators
- Assignment operator
- Relational operators
- Logical operators
- Special Operators
- Set Operators

Arithmetic operators:-

- The arithmetic operations can be used to create expressions on number and date data.
- The arithmetic operations can be used to perform any Arithmetic Operations like Addition, subtraction, Multiplication and Divided by.
- The arithmetic operators can be used in any clause of a sql statement.
- Sql * plus ignores the blank spaces before and after the arithmetic operator .

Example:-

Display salary of employees with 2000 increment in their salary.

Sql> SELECT ename,sal,sal + 2000 "Incremented salary" FROM emp;

Explanation:-in emp table every employee salary sum it 2000.

Arithmetic Operator Subtraction (-):-

- used to perform subtraction between two numbers and dates.

Example:

Display the details of employees decreasing their salary by 200.

Sql> select ename,sal,sal-200 from emp;

Explanation:-in emp table every employee salary subtracted with 200.

Arithmetic Operator Multiplication(*) :-Used to perform multiplication.

Example:-

Display the details of the employees Incrementing their salary two times.

Sql> SELECT sal * 2 FROM emp;

Explanation:-every emp table salary is multiplied by 2.

Arithmetic Operator Division (/):-

Used to perform Division test. Division will display only the Quotient value not the remainder value. Example 6/2 gives 3 because 2 divides 6 by 3 times.

Example:-

Display half of the salary of employees.

Sql> SELECT sal, sal/2 FROM emp;

Examples:-

Sql> select empno,ename,sal,12*sal+100 from emp;

Sql> select empno,ename,sal,(12*sal)+100 from emp;

Sql> select empno,ename,sal,12*(sal+100) from emp;

Assignment operators:-

This operator is used for equality test. Used to test the equality of two operands.

Example:-

Display the details of Employees whose salary is equal to 2000.

Sql> SELECT *FROM emp WHERE sal=950;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM
DEPTNO						
7900	JAMES	CLERK	7698	03-DEC-81	950	
30						

Relational Operator Lessthan(<):-

This operator is used for less than test. Example a<b checks that operand 'a' is less than 'b' or not.

Example: Display the details of the employees whose salary is less than 3000.

Sql> SELECT * FROM emp WHERE sal < 3000;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM
DEPTNO						
7369	SMITH	CLERK	7902	17-DEC-80	800	20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500
7566	JONES	MANAGER	7839	02-APR-81	2975	20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400
30						
7698	BLAKE	MANAGER	7839	01-MAY-81	2850	30
7782	CLARK	MANAGER	7839	09-JUN-81	2450	10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0
30						
7876	ADAMS	CLERK	7788	23-MAY-87	1100	20

7900 JAMES	CLERK	7698 03-DEC-81	950	30
7934 MILLER	CLERK	7782 23-JAN-82	1300	10

Here if you observe we got a result values whose salary is less than the operand 3000.

Relational Operator Greater than(>):-

This operator is used for Greater than test. For example $a > b$ checks the operand 'a' is greater than 'b' or not.

Example:-

Display the details of Employees whose salary is greater than 3000

Sql> SELECT * FROM emp WHERE sal > 3000;

EMPNO ENAME	JOB	MGR HIREDATE	SAL	COMM	DEPTNO
7839 KING	PRESIDENT	17-NOV-81	5000		10

Relational Operator Less than or Equals to(<=):

This operator is used for Less than or Equal to test. For example $a \leq b$, here checks whether operand 'a' is less than or equals to operand 'b'. If $a < b$ then condition is true and if $a = b$ then also condition is true but if $a > b$ then condition is false.

Example :-

Display the details of Employees whose salary is less than or equal to 3000.

Sql> SELECT * FROM EMP WHERE sal <= 3000;

Relational Operator Greater than or Equals to(>=):

This operator is used to check the Greater than or equal test. For example $a \geq b$ checks the operand 'a' is greater than operand 'b' or operand 'a' is equals to the operand 'b'.

Example:-

Display the details of Employees whose salary is greater than or equal to 3000.

Sql> SELECT * FROM emp WHERE sal >= 3000;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM
DEPTNO						
7788	SCOTT	ANALYST	7566	19-APR-87	3000	20
7839	KING	PRESIDENT		17-NOV-81	5000	
10						
7902	FORD	ANALYST	7566	03-DEC-81	3000	20

Relational Operator Not Equals to(!= or ^= or <>):

This operator is used for inequality test.

Examples:

Display the details of employees whose salary is not equals to 2000.

Sql> SELECT * FROM emp WHERE sal != 3000;

Sql> SELECT * FROM emp WHERE sal ^= 2000;

Sql> SELECT * FROM emp WHERE sal <> 2000;

Logical operators:-

AND operator:-

Returns 'True' if both component conditions are true. Returns 'False' if any one component condition or Both Component conditions are False.

Example:-

Display the details of Employees whose salary is Greater than 1000 AND also whose salary is less than 2000.

```
Sql> SELECT *FROM emp WHERE sal > 1000 AND sal < 2000;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250		500
	30						
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250		1400
	30						
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7876	ADAMS	CLERK	7788	23-MAY-87		1100	
	20						
7934	MILLER	CLERK	7782	23-JAN-82		1300	
	10						

```
Sql> select ename,sal,job from emp
```

```
where (sal>=1500 and sal<=5000) and  
job='MANAGER';
```

The Logical OR operator:-

- Returns True if either component conditions become TRUE. Returns False if both the component conditions becomes False.

Example:

Display the details of Employees whose salary is Greater than 1000 OR also whose salary is less than 2000.

```
Sql> SELECT *FROM emp WHERE sal> 1000 OR sal < 2000;
```

Explanation:-whose salaries more than 1000 or less than 2000 that all emp table display.

```
SQL> select empno,ename,job,hiredate from emp
```

```
where job='MANAGER' or deptno=20;
```

```
sql> select empno,ename,job,hiredate from emp
```

```
        where (job='MANAGER' or deptno=10);

sql> select empno,ename,job,hiredate from emp

        where (job='CLERK' or job='SALESMAN' or job='ANALYST');

SQL> select empno,ename,job,hiredate from emp

        where (sal<=2500 or sal>=5000) or job='MANAGER';

sql> select ename,job ,sal from emp

        where job='CLERK' or job='MANAGER' and sal>1500;
```

The Logical NOT operator:-

The NOT operator returns 'True' if the condition is False and returns 'False' if the following condition is True.

Example:

Display the details of employees whose salary is Greater than or Equals to 3000.

```
Sql> SELECT * FROM emp WHERE sal < 3000;
```

Explanation:-whose salary less than 3000 that salaries all are comming.

```
SQL> select empno,ename,job,sal from emp

        where not ename='SMITH';

SQL> select empno,ename,job,sal from emp

        where not sal>=5000;

sql> select empno,ename,job,sal,deptno from emp

        where not job='CLERK' and deptno=20;
```

Special operators:-

IN operator:-

- Returns true if value is available in given list of values
- Supports with all types of data (data types)

In the below example only employees whose empno is (7125,7369,7782) are fetched.

```
Sql> SELECT *FROM emp WHERE empno IN (7125, 7369, 7782);
```

EMPNO DEPTNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM
7369	SMITH	CLERK	7902	17-DEC-80	800	20
7782	CLARK	MANAGER	7839	09-JUN-81	2450	10

Inside DML statements:-

```
Sql> UPDATE emp SET sal=sal+200 WHERE ename IN ('SMITH','ALLEN','WARD');
```

```
Sql> DELETE FROM emp WHERE hiredate IN ('22-DEC-82','17-NOV-81');
```

Not in operator:-

'not in' operator is quite opposite to 'IN' clause.

```
Sql> SELECT *FROM emp WHERE empno NOT IN (7125, 7369,7782);
```

Inside DML statements:-

```
Sql> UPDATE emp SET sal=sal+200 WHERE ename NOT IN ('SMITH','ALLEN','WARD');
```

```
Sql> DELETE FROM emp WHERE hiredate NOT IN ('22-DEC-82',' 17-NOV-81');
```

BETWEEN Operator:-

- Returns true if value specified is within the specified range.
- Supports with numbers and date values.
- Returns all values from given range including source & destination values.
- Always apply on low value to high value.

Example:- in this example all employee records are fetched whose salary is between 2000 and 3000

Sql> SELECT *FROM emp WHERE sal BETWEEN 2000 AND 3000;

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		
10							
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20
7902	FORD	ANALYST	7566	03-DEC-81	3000		20

Whenever lower bound value is larger than upper bound then it shows 'no rows selected'

Example:-

Sql> SELECT *FROM emp WHERE sal BETWEEN 3000 AND 2000;

Output:

-- No rows selected

SQL> select ename,sal,job from emp

where job between 'MANAGER' and 'SALESMAN';

sql> select ename,sal,job,hiredate from emp

where hiredate between '17-DEC-81' and '20-JUN-83';

Not between operator:-

- Returns true if value specified is not within the specified range.
- Supports with numbers and date values.

- Not between is an exclusive operator which eliminates range limits from Output.

Example:-

```
Sql> SELECT *FROM emp WHERE sal NOT BETWEEN 2000 AND 3000;
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800	20	
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250		500
	30						
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250		1400
	30						
7839	KING	PRESIDENT		17-NOV-81	5000	10	
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7876	ADAMS	CLERK	7788	23-MAY-87		1100	
	20						
7900	JAMES	CLERK	7698	03-DEC-81		950	30
7934	MILLER	CLERK	7782	23-JAN-82		1300	
	10						

Note:-

Lower bound – 'value' must be lower when compare to 'upper bound 'value

Upper bound- 'value' must be higher when compare to 'lower bound 'value

```
Sql> select ename,sal,job from emp
      where job not between 'MANAGER' and 'SALESMAN';
sql> select ename,sal,job,hiredate from emp
      where hiredate not between '17-DEC-81' and '20-JUN-83';
```

LIKE operator: -

Used to search for specific pattern in a given input.

% (percentage) and _ (underscore) are two wildcard characters.

% (percentage) represents "remaining group of characters "in the given input

_ (underscore) represents "one character" in given input.

Syntax:-

Select *From <tableName>

Where <character data type column> like '<value>';

Example:- Display the employees whose name is starting with 'S' in EMP table.

Sql> SELECT * FROM emp WHERE ename LIKE 'S%'

EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO

7369	SMITH	CLERK	7902	17-DEC-80	800	20
7788	SCOTT	ANALYST	7566	19-APR-87	3000	20

Display the employees whose name ends with 'S' in EMP table

Sql> SELECT * FROM emp WHERE ename LIKE '%S'

EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO

7566	JONES	MANAGER	7839	02-APR-81	2975	20
7876	ADAMS	CLERK	7788	23-MAY-87	1100	20
7900	JAMES	CLERK	7698	03-DEC-81	950	30

Display the employees whose names are having second letter as 'L' in EMP table

Sql> SELECT * FROM emp WHERE ename LIKE '_L%'

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10

Sql> select ename ,hiredate from emp where hiredate like '%JAN%';

Sql> select empno,ename,job from emp where job like '_____';

EX:

WHOSE EMPLOYEE NUMBER STARTS WITH 7 AND ENDS WITH 8 DIGIT?

SELECT * FROM EMP WHERE EMPNO LIKE '7%8';

EX:

WHO ARE JOINED IN THE YEAR 1981?

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

EX:

WHO ARE JOINED IN THE MONTH OF DEC?

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

EX:

WHO ARE JOINED IN MONTH OF "FEB" , "MAY" ?

SELECT * FROM EMP WHERE HIREDATE LIKE '%FEB%' OR HIREDATE LIKE '%MAY%';

```
SQL> select empno,ename,job ,hiredate from emp where hiredate like '%-FEB-81';
```

Like operator with special char's:

EX: WHOSE EMPLOYEE NAME IS HAVING "#" CHAR?

```
SELECT * FROM EMP77 WHERE ENAME LIKE '%#%';
```

EX: WHOSE EMPLOYEE NAME IS HAVING "@" CHAR?

```
SELECT * FROM EMP77 WHERE ENAME LIKE '%@%';
```

EX: WHOSE EMPLOYEE NAME IS HAVING "_" CHAR?

```
SELECT * FROM EMP77 WHERE ENAME LIKE '%\_%'ESCAPE '\';
```

EX: WHOSE EMPLOYEE NAME IS HAVING "%" CHAR?

```
SELECT * FROM EMP77 WHERE ENAME LIKE '%\%%'ESCAPE '\';
```

NOTE:

Generally oracle db server will treat these symbol(%, _) as "wildcard operators" but not "special char's". To overcome this problem we must use a pre-defined statement is "escape '\'".

Not like operator:-

syntax:-

```
select *from <table Name>
```

```
where <character data type column> not like '<value>';
```

Display the employees whose name is not ends with 'S' in EMP table?

```
Sql> SELECT *FROM emp WHERE ename NOT LIKE '%S';
```

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20

7369	SMITH	CLERK	7902	17-DEC-80	800		20
------	-------	-------	------	-----------	-----	--	----

7499 ALLEN	SALESMAN	7698 20-FEB-81	1600	300	30
7521 WARD	SALESMAN	7698 22-FEB-81	1250	500	30
7654 MARTIN 30	SALESMAN	7698 28-SEP-81	1250	1400	
7698 BLAKE	MANAGER	7839 01-MAY-81	2850		30
7782 CLARK 10	MANAGER	7839 09-JUN-81		2450	
7788 SCOTT	ANALYST	7566 19-APR-87	3000		20
7839 KING	PRESIDENT	17-NOV-81	5000		10
7844 TURNER	SALESMAN	7698 08-SEP-81	1500		30
7902 FORD	ANALYST	7566 03-DEC-81	3000		20
7934 MILLER	CLERK	7782 23-JAN-82	1300		10

Display the employees whose names are not having second letter as 'L' in EMP table?

Sql> SELECT *FROM emp WHERE ename NOT LIKE '_L%';

Display the employees whose names are not start with 'S' in EMP table.?

Sql> SELECT *FROM emp WHERE ename NOT LIKE 'S%';

Sql> select ename ,hiredate from emp where hiredate not like '%JAN%';

Sql> select empno,ename,job from emp where ename not like '_O%';

Display the employees whose names are second letter start with 'R' from ending.?

Sql> SELECT *FROM emp WHERE ename LIKE '%R_';

EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO

7521 WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7782 CLARK 10	MANAGER	7839	09-JUN-81	2450		
7902 FORD	ANALYST	7566	03-DEC-81	3000		20

Display the names in EMP table whose names having 'LL'.?

Sql> SELECT *FROM emp WHERE ename LIKE '%LL%';

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM
DEPTNO						

7499 ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
------------	----------	------	-----------	------	-----	----

7934 MILLER	CLERK	7782	23-JAN-82	1300		10
-------------	-------	------	-----------	------	--	----

Is null Operator: Comparing nulls in a table. NULL is unknow (or) undefined value in database, NULL != 0 & NULL != space, so use " IS " keyword.

syntax:

where <column name> is null;

Ex:

Waq to display employee whose commission is NULL ?

SELECT * FROM EMP WHERE COMM IS NULL;

Ex:

Waq to display employee whose commission is NOT NULL ?

SELECT * FROM EMP WHERE COMM IS NOT NULL;

Ex:

Waq to display ename,job,sal,comm and also sal+comm from emp table whose ename is "SMITH" ?

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

ENAME	JOB	SAL	COMM	COMM+SAL
SMITH	CLERK	800		

Note: If any arithmetic operator is performing some operation with NULL then it again returns NULL only.

Ex: If $x=1000$;

- i) $x+null \rightarrow 1000+0 \rightarrow 1000$
- ii) $x-null \rightarrow 1000-null \rightarrow null$
- iii) $x*null \rightarrow 1000*null \rightarrow null$
- iv) $x/null \rightarrow 1000/null \rightarrow null$

- To overcome the above problem we should use a predefined function is called as "NVL()"

Nvl(Exp1,Exp2): Nvl stands for NULL VALUE. It is pre-defined function. is used to replace a user defined value in place of NULL in the expression. This function is having two arguments are Expression1 and Expression2.

- > If Exp1 is null \rightarrow returns Exp2 value (user defined value)
- > If Exp1 is not null \rightarrow returns Exp1 only.

Ex:

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

NVL(NULL,0)

0

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

NVL(NULL,100)

100

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

NVL(0,100)

0

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

ENAME	JOB	SAL	COMM	NVL(COMM,0)+SAL
-------	-----	-----	------	-----------------

SMITH	CLERK	880		880
-------	-------	-----	--	-----

Nvl2 (Exp1, Exp2, Exp3): Pre-defined function which is an extension of NVL () having 3 arguments are Exp1, Exp2, Exp3.

- If Exp1 is Null -----> Exp3 value (User Defined value)
- If Exp1 is not null -----> Exp2 value (User Defined value)

Ex: Waq to update all employee commissions in a table based on the following conditions?

- i) IF employee comm is null then update those employees comm as 600.
- ii) IF employee comm is not null then update those employees comm as comm+500.

Sol:- UPDATE EMP SET COMM=NVL2 (COMM, COMM+500,600);

Set Operators:

SQL set operators allows combine results from two or more SELECT statements. At first sight this looks similar to SQL joins although there is big difference. SQL joins tends to combine columns i.e. with each additionally joined table it is possible to select more and more columns. SQL set operators on the other hand combine rows from different queries with strong preconditions - all involved SELECTS must. Joins we are collecting the data from two tables when there is a common data. But in set operators the data is not joined, in this the data is merged

- Retrieve the same number of columns and

- The data types of corresponding columns in each involved SELECT must be compatible (either the same or with possibility implicitly convert to the data types of the first SELECT statement).

Operator	Returns
UNION	All distinct rows selected by either query
UNION ALL	All rows selected by either query, including all duplicates
INTERSECT	All distinct rows selected by both queries
MINUS	All distinct rows selected by the first query but not the second

You can combine multiple queries using the set operators UNION, UNION ALL, INTERSECT, and MINUS. All set operators have equal precedence. If a SQL statement contains multiple set operators, then Oracle Database evaluates them from the left to right unless parentheses explicitly specify another order.

The corresponding expressions in the select lists of the component queries of a compound query must match in number and must be in the same datatype group

If component queries select character data, then the datatype of the return values are determined as follows:

- If both queries select values of datatype CHAR of equal length, then the returned values have datatype CHAR of that length. If the queries select values of CHAR with different lengths, then the returned value is VARCHAR2 with the length of the larger CHARvalue.
- If either or both of the queries select values of datatype VARCHAR2, then the returned values have datatype VARCHAR2.

In queries using set operators, Oracle does not perform implicit conversion across datatype groups. Therefore, if the corresponding expressions of

A
B
C

component queries resolve to both character data and numeric data, Oracle returns an error.

Set Operator Guidelines:-

- The expressions in the SELECT lists must match in number and data type.
- Parentheses can be used to alter the sequence of execution.
- The ORDER BY clause:
 - Can appear only at the very end of the statement
 - Will accept the column name, aliases from the first SELECT statement, or the positional notation
- Column names from the first query appear in the result.

Advantage of set operator:-

- Use a set operator to combine multiple queries into a single query.
- These operators are used to combine the information of similar data type from One or more than one table.

Restrictions on the Set Operators:-

The set operators are subject to the following restrictions:

- The ORDER BY clause doesn't recognize the column names of the second SELECT
- The set operators are not valid on columns of type BLOB, CLOB, BFILE, VARRAY, or nested table.
- The UNION, INTERSECT, and MINUS operators are not valid on LONG columns.
- Set operations are not allowed on SELECT statements containing TABLE collection expressions.
- SELECT statements involved in set operations can't use the FOR UPDATE clause.

SQL statements containing these set operators are referred to as compound queries, and each SELECT statement in a compound query is referred to as a component query. Two SELECTs can be combined into a compound query by a set operation only if they satisfy the following two conditions:

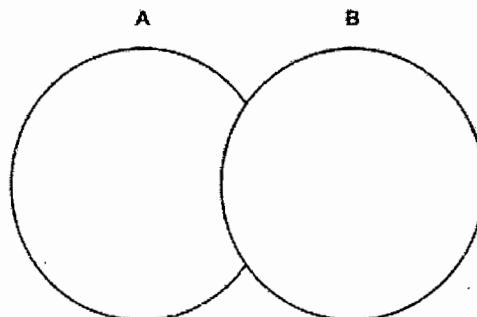
- 1. The result sets of both the queries must have the same number of columns.**
- 2. The datatype of each column in the second result set must match the datatype of its corresponding column in the first result set.**

The generic syntax of a query involving a set operation is:

```
<component query>
{UNION | UNION ALL | MINUS | INTERSECT}
<component query>
```

- **UNION:- UNION Operator combines the results of two select statements into one result set, and then eliminates any duplicates rows from the final result set.**

UNION Operator



The UNION operator returns results from both queries after eliminating duplications.

Example:-

- **Sql> select empno,ename from emp where deptno=20**
- **union**

- **select empno,ename from emp where deptno=30 order by 1;**

Explanation:- The above statement combines the results of two queries with the UNION operator, which eliminates duplicate selected rows. This statement shows that you must match datatype (using the TO_CHAR function) when columns do not exist in one or the other table:

```
Sql> select empno,ename,job from emp
```

```
where deptno=(select deptno from dept
```

```
Where dname='SALES')
```

```
union
```

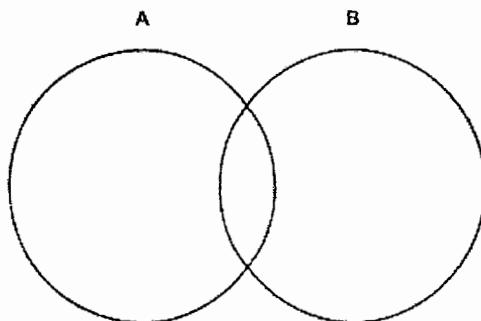
```
select empno,ename,job from emp
```

```
where deptno=(select deptno from dept where  
dname='ACCOUNTING') order by 1;
```

Union All:-

UNION ALL Operator combines the results of two select statements into one result set including Duplicates.

UNION ALL Operator



The UNION ALL operator returns results from both queries, including all duplications.

Example:-

```
Sql> select empno,ename from emp where deptno=10
```

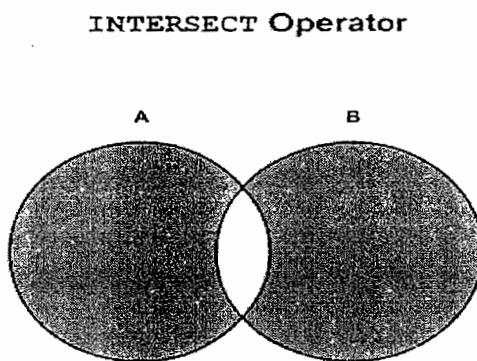
```
union all  
select empno,ename from emp where deptno=30  
order by 1;
```

Explanation:- The UNION operator returns only distinct rows that appear in either result, while the UNION ALL operator returns all rows. The UNION ALL operator does not eliminate duplicate selected rows:

```
sql> select job from emp where deptno=20  
union all  
select job from emp where deptno=30;
```

INTERSECT:-

INTERSECT Operator returns only those rows that are common in both tables.



The INTERSECT operator returns rows that are common to both queries.

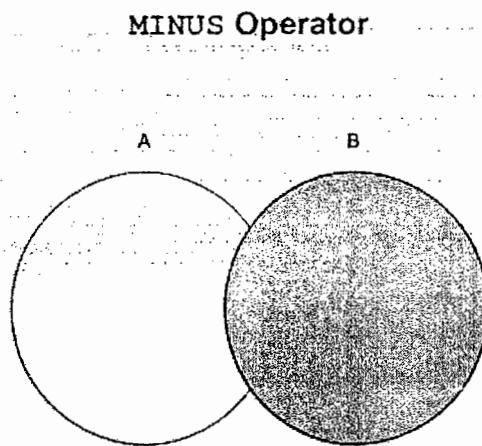
```
Sql> select empno,ename from emp where deptno=10  
intersect  
select empno,ename from emp where deptno=30  
order by 1;
```

Explanation:- The above statement combines the results with the INTERSECT operator, which returns only those rows returned by both queries.

```
Sql> select job from emp where deptno=20  
      intersect  
      select job from emp where deptno=30;
```

MINUS:-

MINUS Operator takes the result set of first select statement and removes those rows that are returned by a second select statement.



The MINUS operator returns rows in the first query that are not present in the second query.

Example:-

```
Sql> select empno,ename from emp  
      Where  deptno=10  
      minus  
      select empno,ename from emp  
      where deptno=30 order by 1;
```

Explanation:- The above statement combines results with the MINUS operator, which returns only unique rows returned by the first query but not by the second:

```
Sql>select job from emp where deptno=20
```

```
minus
```

```
select job from emp where deptno=30;
```

Using Set Operations to Compare Two Tables:-

Developers, and even DBAs, occasionally need to compare the contents of two tables to determine whether the tables contain the same data. The need to do this is especially common in test environments, as developers may want to compare a set of data generated by a program under test with a set of "known good" data. Comparison of tables is also useful for automated testing purposes, when we have to compare actual results with a given set of expected results. SQL's set operations provide an interesting solution to this problem of comparing two tables.

The following query uses both MINUS and UNION ALL to compare two tables for equality. The query depends on each table having either a primary key or at least one unique index.

Example:-

```
Sql>DESC CUSTOMER_KNOWN_GOOD
```

Name	Null?	Type
------	-------	------

CUST_NBR	NOT NULL	NUMBER(5)
----------	----------	-----------

NAME	NOT NULL	VARCHAR2(30)
------	----------	--------------

```
SELECT * FROM CUSTOMER_KNOWN_GOOD;
```

CUST_NBR	NAME
----------	------

1	Sony
---	------

1	Sony
---	------

2 Samsung

3 Panasonic

3 Panasonic

3 Panasonic

Sql>DESC CUSTOMER_TEST

Name	Null?	Type
------	-------	------

CUST_NBR	NOT NULL	NUMBER(5)
----------	----------	-----------

NAME	NOT NULL	VARCHAR2(30)
------	----------	--------------

Sql>SELECT * FROM CUSTOMER_TEST;

CUST_NBR	NAME
----------	------

1	Sony
---	------

1	Sony
---	------

2	Samsung
---	---------

2	Samsung
---	---------

3	Panasonic
---	-----------

As we can see the CUSTOMER_KNOWN_GOOD and CUSTOMER_TEST tables have the same structure, but different data. Also notice that none of these tables has a primary or unique key; there are duplicate records in both. The following SQL will compare these two tables effectively.

Example:-

```
Sql>(SELECT * FROM CUSTOMER_KNOWN_GOOD  
      MINUS  
      SELECT * FROM CUSTOMER_TEST)
```

```
UNION ALL  
(SELECT * FROM CUSTOMER_TEST  
MINUS  
SELECT * FROM CUSTOMER_KNOWN_GOOD);
```

Explanation:-Let's talk a bit about how this query works. We can look at it as the union of two compound queries. The parentheses ensure that both MINUS operations take place first before the UNION ALL operation is performed. The result of the first MINUS query will be those rows in CUSTOMER_KNOWN_GOOD that are not also in CUSTOMER_TEST. The result of the second MINUS query will be those rows in CUSTOMER_TEST that are not also in CUSTOMER_KNOWN_GOOD. The UNION ALL operator simply combines these two result sets for convenience. If no rows are returned by this query, then we know that both tables have identical rows. Any rows returned by this query represent differences between the CUSTOMER_TEST and CUSTOMER_KNOWN_GOOD tables.

```
Sql>(SELECT C1.*, COUNT(*)  
FROM CUSTOMER_KNOWN_GOOD C1  
GROUP BY C1.CUST_NBR, C1.NAME  
MINUS  
SELECT C2.*, COUNT(*)  
FROM CUSTOMER_TEST C2  
GROUP BY C2.CUST_NBR, C2.NAME)  
UNION ALL  
(SELECT C3.*, COUNT(*)  
FROM CUSTOMER_TEST C3  
GROUP BY C3.CUST_NBR, C3.NAME  
MINUS  
SELECT C4.*, COUNT(*)  
FROM CUSTOMER_KNOWN_GOOD C4
```

```
GROUP BY C4.CUST_NBR, C4.NAME);
```

Explanation:- These results indicate that one table (**CUSTOMER_KNOWN_GOOD**) has one record for "Samsung", whereas the second table (**CUSTOMER_TEST**) has two records for the same customer. Also, one table (**CUSTOMER_KNOWN_GOOD**) has three records for "Panasonic", whereas the second table (**CUSTOMER_TEST**) has one record for the same customer. Both the tables have the same number of rows (two) for "Sony", and therefore "Sony" doesn't appear in the output.

TIP: Duplicate rows are not possible in tables that have a primary key or at least one unique index. Use the short form of the table comparison query for such tables.

Type conversion functions:-

If corresponding expressions does not belongs to same data type also we can restrictive data from multiple queries ,in this case we use appropriate type conversion function.

Example:-

```
Sql> select ename "name",to_number(null) "deptno" from emp  
union  
select to_char(null),deptno from dept;
```

```
sql> select empno,ename,sal,to_number(null) from emp
```

```
where deptno=20
```

```
union
```

```
select empno,ename,to_number(null),hiredate from emp  
where deptno=30;
```

Using NULLs in Compound Queries

We discussed union compatibility conditions at the beginning of this chapter. The union compatibility issue gets interesting when NULLs are involved. As we know, NULL doesn't have a datatype, and NULL can be used in place of a value of any datatype. If we purposely select NULL as a

column value in a component query, Oracle no longer has two datatypes to compare in order to see whether the two component queries are compatible. For character columns, this is no problem. For example:

```
Sql>SELECT 1 NUM, 'DEFINITE' STRING FROM DUAL  
      UNION  
      SELECT 2 NUM, NULL STRING FROM DUAL;  
      NUM STRING
```

```
-----  
      1 DEFINITE
```

```
      2
```

Notice that Oracle considers the character string 'DEFINITE' from the first component query to be compatible with the NULL value supplied for the corresponding column in the second component query. However, if a NUMBER or a DATE column of a component query is set to NULL, we must explicitly tell Oracle what "flavor" of NULL to use. Otherwise, we'll encounter errors. For

example:

```
sql>SELECT 1 NUM, 'DEFINITE' STRING FROM DUAL  
      UNION  
      SELECT NULL NUM, 'UNKNOWN' STRING FROM DUAL;
```

ERROR at line 1:

ORA-01790: expression must have same datatype as corresponding expression

Note that the use of NULL in the second component query causes a datatype mismatch between the first column of the first component query, and the first column of the second component query. Using NULL for a DATE column causes the same problem, as in the following

Example:-

```
sql>SELECT 1 NUM, SYSDATE DATES FROM DUAL  
      UNION
```

SELECT 2 NUM, NULL DATES FROM DUAL;

SELECT 1 NUM, SYSDATE DATES FROM DUAL

ERROR at line 1:

ORA-01790: expression must have same datatype as corresponding expression

In these cases, we need to cast the NULL to a suitable datatype to fix the problem, as in the following examples:-

SqI>SELECT 1 NUM, 'DEFINITE' STRING FROM DUAL

UNION

SELECT TO_NUMBER(NULL) NUM, 'UNKNOWN' STRING FROM DUAL;

NUM STRING

1 DEFINITE

UNKNOWN

SqI>SELECT 1 NUM, SYSDATE DATES FROM DUAL

UNION

SELECT 2 NUM, TO_DATE(NULL) DATES FROM DUAL;

NUM DATES

**-----
1 06-JAN-02**

2

This problem of union compatibility when using NULLs is encountered in Oracle8i. However, there is no such problem in Oracle9i, as we can see in the following examples generated from an Oracle9i database.

SqI>SELECT 1 NUM, 'DEFINITE' STRING FROM DUAL

UNION
SELECT NULL NUM, 'UNKNOWN' STRING FROM DUAL;

NUM STRING

1 DEFINITE

UNKNOWN

SQL>SELECT 1 NUM, SYSDATE DATES FROM DUAL
UNION
SELECT 2 NUM, NULL DATES FROM DUAL;

NUM DATES

1 06-JAN-02

2

EX:-

- sql>select job from emp where deptno=20
union
select job from emp where deptno=30;

EX:-

SQL> SELECT * FROM EMP_HYD;

EID	ENAME	SAL
1021	SAI	85000
1022	JONES	48000
1023	ALLEN	35000

SQL> SELECT * FROM EMP_CHENNAI;

EID	ENAME	SAL
1021	SAI	85000
1024	WARS	36000
1025	MILLER	28000

EX: Waq to display all employee details who are working in NARESHIT organization?

SELECT * FROM EMP_HYD UNION SELECT * FROM EMP_CHENNAI;

(OR)

SELECT * FROM EMP_HYD UNION ALL SELECT * FROM EMP_CHENNAI;

EX: Waq to display employee details who are working in both branches ?

SELECT * FROM EMP_HYD INTERSECT SELECT * FROM EMP_CHENNAI;

EX: Waq to display employee details who are working in HYD BUT NOT IN CHENNAI branches ?

SELECT * FROM EMP_HYD MINUS SELECT * FROM EMP_CHENNAI;

EX: Waq to display employee details who are working in CHENNAI BUT NOT IN HYD branches ?

SELECT * FROM EMP_CHENNAI MINUS SELECT * FROM EMP_HYD;

Functions in oracle

- > To Perform Task & Must Return Value.
- > Oracle Supports Two Types Functions. Those Are

- 1) Pre-Define / Built in Functions (Use in Sql & Pl/Sql)
- 2) User Define Functions (Use in Pl/Sql)

1) Pre-Define Functions:

- > These Are Again Classified into Two Categories.

- A) Single Row Functions (Scalar Functions)
- B) Multiple Row Functions (Grouping Functions)

Single Row Functions:

- > These Functions Are Returns A Single Row (Or) A Single Value.

- > Numeric Functions
- > String Functions
- > Date Functions
- > Conversion Functions

How To Call a Function:

Syntax:

Select <Fname>(Values) From Dual;

What Is Dual:

- > Pre-Define Table In Oracle.
- > Having Single Column & Single Row
- > Is Called As Dummy Table In Oracle.
- > Testing Functions (Pre-Define & User Define) Functionalities.

To View Strc.Of Dual Table:

Sql> Desc Dual;

To View Data Of Dual Table:

Sql> Select * From Dual;

Numeric Functions:

1) Abs():

> Converts (-Ve) Value Into (+Ve) Value.

Syntax:

Abs(Number)

Ex:

Sql> Select Abs(-12) From Dual; -----> 12

Sql> Select Ename,Sal,Comm,Abs(Comm-Sal) From Emp;

2) Ceil():

> Returns A Value Which Is Greater Than Or Equal To Given Value.

Syntax:

Ceil(Number)

Ex:

Sql> Select Ceil(9.0) From Dual;-----9

Sql> Select Ceil(9.3) From Dual;-----10

3) Floor():

Syntax:

Floor(Number)

Ex:

Sql> Select Floor(9.0) From Dual;-----9

Sql> Select Floor(9.8) From Dual;-----9

4) Mod():
Returns Remainder Value.

Syntax:
 Mod(M,N)

Ex:
Sql> Select Mod(10,2) From Dual;-----0

5) Power():
The Power Of Given Expression

Syntax:
 Power(M,N)

Ex:
Sql> Select Power(2,3) From Dual;-----8

Round():
> Nearest Value Given Expression.

Syntax:
 Round(Number,[Decimal Places])

Ex:
Sql> Select Round(5.50) From Dual;-----6
Sql> Select Round(32.456,2) From Dual;-----32.46

Trunc:

> Returns A Value Which Will Specified Number Of Decimal Places.

Syntax:
 Trunc(Number,Decimal Places)

Ex:
Sql> Select Trunc(5.50) From Dual;-----5
Sql> Select Trunc(32.456,2) From Dual;----32.45

String Functions:

Length():

> Length Of Given String.

Syntax:

Length(String)

Ex:

**Sql> Select Length('Hello') From Dual;-----5
Sql> Select Length('Good Morning') From Dual;-----12**

Sql> Select Ename,Length(Ename) From Emp;

Sql> Select * From Emp Where Length(Ename)=4;

Lower():

To Convert Upper Case Char's Into Lower Case Char's.

Syntax:

Lower(String)

Ex:

**Sql> Select Lower('Hello') From Dual;
Sql> Update Emp Set Ename=Lower(Ename) Where Job='Clerk';**

Upper():

Syntax:

Upper(String)

Ex:

Sql> Select Lower('Hello') From Dual;

Initcap():

To Convert First Char. Is Capital.

Syntax:

Initcap(String)

Ex:

```
Sql> Select Initcap('Hello') From Dual;  
Sql> Select Initcap('Good Morning') From Dual;
```

Ltrim():

To Remove Unwanted Spaces (Or) Unwanted Characters From Left Side Of Given String.

Syntax:

```
Ltrim(String1[,String2])
```

Ex:

```
Sql> Select Ltrim('Sai') From Dual;  
Sql> Select Ltrim('Xxxxxxsai','X') From Dual;  
Sql> Select Ltrim('123SAI','123') From Dual;
```

Rtrim():

To Remove Unwanted Spaces (Or) Unwanted Characters From Right Side Of Given String.

Syntax:

```
Rtrim(String1[,String2])
```

Ex:

```
Sql> Select Rtrim('Saixxxxxx','X') From Dual;
```

Trim():

To Remove Unwanted Spaces (Or) Unwanted Characters From Both Sides Of Given String.

Syntax:

```
Trim('Trimming Char' From 'String')
```

Ex:

```
Sql> Select Trim('X' From 'Xxxxxxsaixxxx') From Dual;
```

Lpad():

To Fill A String With Specific Char. On Left Side Of Given String.

Syntax:

Lpad(String1,Length,String2)

Ex:

Sql> Select Lpad('Hello',10,'@') From Dual;
@@@@@@Hello

Rpad():

To Fill A String With Specific Char. On Right Side Of Given String.

Syntax:

Rpad(String1,Length,String2)

Ex:

Sql> Select Rpad('Hello',10,'@') From Dual;
Hello@@@@@

Concat():

Adding Two String Expressions.

Syntax:

Concat(String1,String2)

Ex:

Sql> Select Concat('Good','Bye') From Dual;

Replace():

To Replace One String With Another String.

Syntax:

Replace(String1,String2,String3)

Ex:

Sql> Select Replace('Hello','Ell','Xyz') From Dual;
Hxyz

Sql> Select Replace('Hello','L','Abc') From Dual;
Heabcabco

Translate():

To Translate A Single Char With Another Single Char.

Syntax:

Translate(String1, String2, String3)

Ex:

Sql> Select Translate('Hello','Elo','Xyz') From Dual;
Hxxyz

**Sol: E = X , L=Y , O=Z
Hello => Hxxyz**

Ex:

Sql> Select Ename,Sal,Translate(Sal,'0123456789','\$B@Gh*V#T%')
Salary From Emp;

Ename	Sal	Salary
Smith	800	T\$\$

Sol: 0=\$,1=B,2=@,3=G,4=H,5=*,6=V,7=#,8=T,9=%.

Substr():

It Returns Req.Substring From Given String Expression.

Syntax:

Substr(String1,<Starting Position Of Char.>,<Length Of Char's>)

Ex:

Sql> Select Substr('Hello',2,3) From Dual;
Ell

Sql> Select Substr('Welcome',4,2) From Dual;
Co

Sql> Select Substr('Welcome',-6,3) From Dual;
Elc

Instr():

Returns Occurrence Position Of A Char. In The Given String.

Syntax:

Instr(String1, String2,<Starting Position Of Char.>,<Occurrence Position Of Char.>)

Ex:

**Sql> Select Instr('Hello Welcome','O') From Dual;-----> 5
Sql> Select Instr('Hello Welcome','Z') From Dual;-----> 0
Sql> Select Instr('Hello Welcome','O',1,2) From Dual;----11
Sql> Select Instr('Hello Welcome','E',5,2) From Dual;----13
Sql> Select Instr('Hello Welcome','E',1,4) From Dual;-----8**

Note:

Position Of Char's Always Fixed Either Count From Left To Right (Or) Right To Left.

**Sol: Hello Welcome
12345 6 78910111213**

Ex:

**Sql> Select Instr('Hello Welcome','E',-1,3) From Dual;-----2
Sql> Select Instr('Hello Welcome','L',-4,3) From Dual;-----3
Sql> Select Instr('Hello Welcome','L',-6,3) From Dual;-----0**

Date Functions:

1) Sysdate:

> Current Date Information Of The System.

Ex:

```
Sql> Select Sysdate From Dual;  
Sql> Select Sysdate+10 From Dual;  
Sql> Select Sysdate-10 From Dual;
```

Add_Months():

> Adding No.Of Months To The 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():

> Returns The Last Day Of The Month.

Syntax:

```
Last_Day(Date)
```

Ex:

```
Sql> Select Last_Day(Sysdate) From Dual;
```

Next_Day():

> Returns The Next Specified Day From The Given Date.

Syntax:

```
Next_Day(Date,'<Day Name>')
```

Ex:

Sql> Select Next_Day(Sysdate,'Sunday') From Dual;

Months_Between():

> Returns No.Of Months Between Two Date Expressions.

Syntax:

Months_Between(Date1,Date2)

Ex:

Sql> Select Months_Between('05-Jan-81','05-Jan-80') From Dual;---

-- 12

Sql> Select Months_Between('05-Jan-80','05-Jan-81') From Dual;---

-- -12

**Note: Here, Date1 Is Always Greater Than Date2 Otherwise
Oracle Returns Nagative Value.**

Conversion Functions:

- 1. To_Char()**
- 2. To_Date()**

To_Char():

> Date Type To Char Type To Display Date In Different Fromat.

Syntax:

To_Char(Date,[<Format>])

Year Formats:

Yyyy -	2020
Yy -	20
Year -	Twenty Twenty
Cc -	Centuary 21
Ad / Bc -	Ad Yaer / Bc Year

Ex:

Sql> Select To_Char(Sysdate,'Yyyy Yy Year Cc Ad') From Dual;

To_Char(Sysdate,'Yyyyyyyearccad')

2020 20 Twenty Twenty 21 Ad

**Q: To Display Employee Who Are Joined In Year 1982
By Using To_Char() Function ?**

Sol:

Sql> Select * From Emp Where To_Char(Hiredate,'Yyyy')=1982;

**Q: To Display Employee Who Are Joined In Year 1980,1982,1987
By Using To_Char() Function ?**

Sol:

**Sql> Select * From Emp Where To_Char(Hiredate,'Yyyy')
In(1980,1982,1987);**

Month Format:

Mm - Month Number

Mon - First Three Char From Month Spelling

Month - Full Name Of Month

Ex:

Sql> Select To_Char(Sysdate,'Mm Mon Month') From Dual;

To_Char(Sysdate,

08 Aug August

Sql> Select To_Char(Sysdate,'Mm Mon Month') From Dual;

To_Char(Sysdate,

08 Aug August

**Q: To Display Employee Who Are Joined In Feb,May,Dec Months
By Using To_Char() ?**

Sol:

**Sql> Select * From Emp Where To_Char(Hiredate,'Mm')
In(02,05,12);**

**Q: To Display Employee Who Are Joined In Feb 1981
By Using To_Char() ?**

Sol:

```
Sql> Select * From Emp Where  
To_Char(Hiredate,'Myyyy')='021981';
```

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 Day Dy') From Dual;

```
To_Char(Sysdate,'Ddddd')
```

220 07 6 Friday Fri

**Q: To Display Employee Who Are Joined On "Friday" By Using
To_Char() ?**

Sol:

```
Sql> Select * From Emp Where To_Char(Hiredate,'Day')='Friday';
```

Q: To Display Employee On Which Day Employees Are Joined ?

Sol:

```
Sql> Select Ename||' ||'Joined On'||' ||To_Char(Hiredate,'Day')  
From Emp;
```

Note:

**In Oracle Whenever We Using To_Char() And Also Within To_Char()
When We use Day / Month Format Then Oracle Server Internally
Allocate Some Extra Memory For Day/Month Format Of Data.**

**To Overcome The Above Problem That Is To Remove Extra
Memory Which Was Allocate By Oracle Server Then We Use A Pre-
Define Specifier Is
Called "Fm" (Fill Mode).**

Ex:

Select * From Emp Where To_Char(Hiredate,'Fmday')='Friday';

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;

T

3

Q : Who Are Joined In 2ND Quater Of 1981 ?

Sol:

**Sql> Select * From Emp Where To_Char(Hiredate,'Yyyy')='1981'
And To_Char(Hiredate,'Q')=2;**

Week Format:

Ww - Week Of The Year
W - Week Of Month

Ex:

Sql> Select To_Char(Sysdate,'Ww W') From Dual;

To_C

32 2

Time Format:

Hh - Hour Part

Hh24- 24 Hrs Fromat

Mi - Minute Part

Ss - Seconds Part

Am / Pm - Am Tme (Or) Pm Time

Ex:

Sql> Select To_Char(Sysdate,'Hh:Mi:Ss Am') From Dual;

To_Char(Sys

12:04:21 Pm

To_Date():

To Convert Char Type To Oracle Date Format Type.

Syntax:

To_Date(String[,Fromat])

Ex:

Sql> Select To_Date('08/August/2020') From Dual;

To_Date('

08-Aug-20

Sql> Select To_Date('08-Aug-2020')+10 From Dual;

To_Date('

18-Aug-20

Multiple Row Functions:

**These Functions Are Returns Either Group Of Values
(Or) A Single Value.**

Sum():

> It Returns Sum Of A Specific Column Values.

Ex:

**Sql> Select Sum(Sal) From Emp;
Sql> Select Sum(Sal) From Emp Where Job='Clerk';**

Avg():

> It Returns Average Of A Specific Column Values.

Ex:

**Sql> Select Avg(Sal) From Emp;
Sql> Select Avg(Sal) From Emp Where Deptno=10;**

Min():

> It Returns Min.Value From Group Of Values.

Ex:

**Sql> Select Min(Hiredate) From Emp;
Sql> Select Min(Hiredate) From Emp Where Job='Manager';
Sql> Select Min(Sal) From Emp;**

Max():

> It Returns Max.Value From Group Of Values.

Ex:

Sql> Select Max(Sal) From Emp;

Count():

**> It Returns No.Of Rows In A Table / No.Of Values In A Column
> Three Types,
 I) Count(*)
 II) Count(<Column Name>)
III) Count(Distinct <Column Name>)**

Ex:

Test

Sno	Name
101	A
102	B
103	
104	C
105	A
106	C

Count(*):

> Counting All Rows (Duplicates & Nulls) In A Table.

Ex:

Sql> Select Count(*) From Test;

Count(*)

6

Count(<Column Name>):

> Counting All Values Including Duplicate Values But Not Null Values From A Column.

Ex:

Sql> Select Count(Name) From Test;

Count(Name)

5

Count(Distinct <Column Name>):

> Counting Unique Values From A Column. Here "Distinct" Keyword Is Eliminating Duplicate Values.

Ex:

Sql> Select Count(Distinct Name) From Test;-----> 3

CLAUSES:

- Clause Is Statement Which Is Used To Add To Sql Pre-Define Query For Providing Additional Facilities Are Like Filtering Rows, Sorting Values, Grouping Similar Values, Finding

Sub Total and Grand Total Based On the Given Values Automatically.

- Oracle Supports The Following Clauses. Those Are,

- > Where - Filtering Rows (Before Grouping Data)
- > Order by - Sorting Values
- > Group by - Grouping Similar Data
- > Having - Filtering Rows (After Grouping Data)
- > Rollup - Finding Sub Total & Grand Total (Single Column)
- > Cube - Finding Sub Total & Grand Total (Multiple Columns)

Syntax:

<Sql Per-Define Query> + <Clauses>;

Where Clause:

> Filtering Rows in One By One Manner before Grouping Data in Table.

Syntax:

Where <Filtering Condition>

Ex:

Select * From EMP Where Empno=7788;

Update EMP Set Sal=8500 Where Job='Clerk';

Delete From EMP Where Deptno=10;

Note: "Where" Clause Can Be Used In "Select" , "Update" And "Delete" Commands Only.

Order by Clause:

> Sorting Values Based On Columns. It Can Be Used In "Select" Command Only.

> By Default Order By Clause Arrange Values In Ascending Order But If We Want To Arrange Values In Descending Order Then We Use "Desc" Keyword.

Syntax:

Select * / <List Of Column Names> From <Tn> Order By <Column Name1> <Asc / Desc>, <Column Name2> <Asc/Desc>,.....;

Ex1:

Waq to Display Employee Salaries In Ascending Order?

Sol:

Sql> Select * From EMP Order by Sal;

(Or)

Sql> Select Sal from EMP Order by Sal;

Ex2:

Waq to Arrange Employee Names In Descending Order?

Sol:

Sql> Select Ename from EMP Order by Ename Desc;

Ex3:

Waq to Display Employee Who Are Working In The Deptno Is 20 and Arrange Those Employee Salaries In Descending Order?

Sol:

Sql> Select * From EMP Where Deptno=20 Order By Sal Desc;

Ex4:

Waq to Arrange Employee Deptno's In Ascending Order
And Those Employee Salaries in Descending Order From
Each Deptno Wise?

Sol:

Sql> Select * From EMP Order By Deptno, Sal Desc;

Note:

Order by Clause Not Only On Column Names Even though
We Can Apply On Position Of Column In Select Query.

Ex:

Sql> Select * From EMP Order by 6;

Sql> Select Ename, Job, Sal from EMP Order by 3;

Sql> Select Ename, Sal from EMP Order by 2;

Sql> Select Sal from EMP Order by 1;

Note:

Using Order by Clause On "Null" Values Column Then Oracle Returns "Null" Values
Last In Ascending Order And "Null" Values Are Displayed First In Descending By Default.

If We Want To Change This Default Order Of "Null" Then We Use Null Clauses
Are "Nulls First" And " Nulls Last ".

Ex:

Sql> Select * From EMP Order By Comm Nulls First;

Sql> Select * From Emp Order By Comm Desc Nulls Last;

Group By:

> Grouping Similar Data Based On Columns.

> When We Use "Group by" We Must Use "Aggregative Functions" Are
Sum(),Avg(),Min(),Max(),Count().

> Whenever We Implement "Group By" Clause In Select Statement Then

First Grouping Similar Data Based Columns And Later An Aggregative Function/(S)
Will Execute On Each Group Of Data To Produce Accurate Result.

Syntax:

```
Select <Column Name1>,<Column Name2>,...,<Aggregative Function  
Name1>,... From <Tn> Group By <Column Name1>,<Column  
Name2>,...;
```

Group By

Aggregative Functions.

Sum (), Avg (), Job (No. Of In Each Job)

Min (), Max(), Count ()

Clerk	Analyst	President	Manager	Salesman
Clerk	Analyst	(1)	Manager	Salesman
Clerk	(2)		Manager	Salesman
Clerk		(3)		Salesman
			(4)	

Ex1:

Waq to Find Out No. Of Employee Working In Each Job?

Sol:

```
Sql> Select Job,Count(*) Num_Of_Employee From Emp Group By Job;
```

Ex2:

Waq To Calculate Department Number Wise Total Salary ?

Sol:

```
Sql> Select Deptno,Sum(Sal) Total_Salary From Emp
```

```
Group By Deptno Order By Deptno;
```

Ex3:

Waq To Display No.Of Employee Working In Each Job Along With Deptno Wise ?

Sol:

```
Sql> Select Job,Deptno,Count(*) Num_Of_Employee From Emp  
Group By Job,Deptno;
```

Ex4:

Waq To Calculate Deptno Wise Totalsalary Where Deptno's Are 10,20 ?

Sol:

```
Sql> Select Deptno,Sum(Sal) Total_Salary From Emp  
Where Deptno In(10,20) Group By Deptno;
```

Ex5:

Waq To Calculate Deptno Wise Avg,Min,Max Salaries ?

Sol:

```
Sql> Select Deptno,Avg(Sal) Avgsal,Min(Sal) Minsal,Max(Sal) Maxsal From Emp  
Group By Deptno Order By Deptno;
```

Having:

> Filtering Rows after Grouping Data in Table. It Can Be Used Along With "Group By" Clause.

Syntax:

```
Select <Column Name1>,<Column Name2>,...,<Aggregative Function  
Name1>,... From <Tn> Group By <Column Name1>,<Column  
Name2>,... Having <Filtering Condition>;
```

Ex1:

Waq To Find Out No.Of Employee Of Each Job In Which Job No.Of Employee Are More Than 3 ?

Sol:

```
Sql> Select Job,Count(*) From Emp Group By Job  
Having Count(*)>3;
```

Ex2:

Waq To Display Sum Of Salary Of Deptno's From Emp Table.If Sum Of Salary
Of Deptno Is Less Than 9000 ?

Sol:

```
Sql> Select Deptno,Sum(Sal) From Emp Group By Deptno  
Having Sum (Sal)<9000;
```

Diff. B/W "Where" And "Having" Clause:

Where	Having
-----	-----
1. Where Clause Condition Is Executed On Each Row Of A Table.	1. Having Clause Condition Is Executed On Group Of Rows Of A Table.
2. It Can Be Apply Before Group by Clause.	2. It Can Be Apply After Group by Clause.
3. It Cannot Support Aggregative Functions.	3. It Can Supports Aggregative Functions.
4. without Group By We Can Use Where Clause.	4. Without Group By We Cannot Use Having Clause.

Using All Clauses In A Single Select Statement:

Syntax:

```
Select <Col1>,<Col2>,...,<Aggregative Function  
Name1>,...  
From <Table Name> [Where <Filtering Condition>  
Group By <Col1>,<Col2>,...  
Having <Filtering Condition>  
Order By <Col1> [Asc/Desc],<Col2> [Asc/Desc],...  
];
```

Ex:

Select Deptno,Count(*) From Emp

Where Sal>1000

Group By Deptno

Having Count(*)>3

Order By Deptno;

Deptno Count(*)

20	4
30	5

Order of Execution:

- > From
- > Where
- > Group By
- > Having
- > Select
- > Order By

Rollup & Cube:

- > Special Clauses.
- > To Finding Sub Total & Grand Total Based On Columns.
- > Working Along With "Group By" Clause.
- > Rollup Will Find Sub & Grand Total Based On a Single Column.
- > Cube Will Find Sub & Grand Total Based On Multiple Columns.

Syntax:

Group By Rollup (<Col1>,<Col2>,<Col3>,...<Col N>)

Ex. On Rollup with a Single Column:

Sql> Select Deptno, Count (*) From Emp Group By Rollup(Deptno);

Deptno	Count(*)
10	3
20	5
30	6
	14

Ex. On Rollup with Multiple Columns:

Sql> Select Deptno, Job, Count (*) From Emp Group By Rollup(Deptno,Job);

Note: In The Above Ex. Rollup Is Finding Sub & Grand Total Based On A Single Column (Deptno). If We Want To Find Sub & Grand Total Then Use "Cube" Clause.

Syntax:

Group By Cube (<Col1>,<Col2>,...<Col N>).

Ex. On Cube with a Single Column:

Sql> Select Deptno, Count (*) From Emp Group By Cube(Deptno) Order By Deptno;

Ex. On Cube with Multiple Columns:

Sql> Select Deptno, Job, Count (*) From Emp Group By Cube(Deptno , Job)

Order by Deptno;

Grouping_ID ():

> It Used More Compact Way To Identify Sub And Grand Total Rows.

- > Id Number 1: To Represent Sub Total Of First Grouping Column.
- 2: To Represent Sub Total Of Second Grouping Column.
- 3: Grand Total row.

Syntax:

Grouping_ID (<Col1>,<Col2>,.....)

Ex:

Select Deptno, Job, Count (*), Grouping_ID (Deptno, Job) From Emp

Group by Cube (Deptno, Job) Order By Deptno;

JOINS:

JOINS ARE USED TO RETRIEVE DATA FROM MULTIPLE TABLES AT A TIME. IN RELATIONAL DATABASES WE ARE STORING RELATED DATA IN MULTIPLE TABLES LIKE EMPLOYEE DETAILS, DEPARTMENT DETAILS, CUSTOMER DETAILS, ORDERSDETAILS, PRODUCTS DETAILS, ETC.

TO COMBINE DATA AND RETRIEVE DATA FROM THOSE MULTIPLE TABLES THEN WE NEED JOINS. JOINS ARE AGAIN CLASSIFIED INTO TWO WAYS

1) NON - ANSI FORMAT JOINS: (ORACLE 8I JOINS)

- EQUI JOIN
- NON-EQUI JOIN
- SELF JOIN

2) ANSI FORMAT JOINS: (ORACLE 9I JOINS)

- INNER JOIN
- OUTER JOIN
 - LEFT OUTER JOIN
 - RIGHT OUTER JOIN
 - FULL OUTER JOIN
- CROSS JOIN (OR) CARTISEAN JOIN
- NATURAL JOIN

- WHEN WE ARE RETRIEVING DATA FROM MULTIPLE TABLES BASED ON "WHERE"

CLAUSE CONDITION THEN WE CALLED AS NON-ANSI FORMAT JOIN.

- WHEN WE ARE RETRIEVING DATA FROM MULTIPLE TABLES WITH "ON" / "USING"

CLAUSE CONDITION THEN WE CALLED AS ANSI FORMAT JOIN.

SYNTAX FOR NON-ANSI JOINS:

SELECT * FROM TABLE NAME1, TABLE NAME2 WHERE <JOIN CONDITION>;

SYNTAX FOR ANSI JOINS:

SELECT * FROM <TABLE NAME1> <JOIN KEY> <TABLE NAME2 > ON <JOIN CONDITION>;

JOINS TABLES:

EX:

STUDENT_TABLE

STID	SNAME	CID
1021	SAI	10
1022	ADAMS	20
1023	JONES	30

COURSE_TABLE

CID	CNAME	CFEE
10	ORACLE	2500
20	JAVA	6000
40	PHP	4500

EQUI JOIN: RETRIEVING DATA FROM MULTIPLE TABLES BASED ON "EQUAL OPERATOR (=) "IS CALLED AS EQUI JOIN.

WHEN WE USE EQUI JOIN BETWEEN TWO (OR) MORE THAN TWO TABLES THERE MUST BE COMMON COLUMN (OR) COMMON FIELD NAME IS NO NEED TO BE SAME NAME (BUT RECOMMEND). COMMON COLUMN (OR) COMMON FIELD DATATYPE MUST BE MATCH.

WHEN WE PERFORM ANY JOIN OPERATION BETWEEN TABLES THERE IS NO NEED TO HAVE RELATIONSHIP(OPTIONAL).(I.E PRIMARY KEY & FOREIGN KEY RELATION).EQUI JOIN ALWAYS RETRIEVING ONLY MATCHING DATA / MATCHNG ROWS.

SYNTAX:

WHERE <TABLE NAME1>. <COMMON COLUMN> = <TABLE NAME2>. <COMMON COLUMN>;

(OR)

WHERE <TN1 ALIAS NAME>. <COMMON COLUMN> = <TN2 ALIAS NAME>. <COMMON COL>;

EX1:

WAQ TO RETRIEVE STUDENT AND THE CORRESPONDING COURSE DETAILS FROM

STUDENT, COURSE TABLES BY USING EQUI JOIN ?

SOL:

SQL> SELECT * FROM STUDENT, COURSE WHERE CID=CID;

ERROR AT LINE 1:

ORA-00918: COLUMN AMBIGUOUSLY DEFINED

NOTE: IN ABOVE EXAMPLE WE GET AN ERROR IS "COLUMN AMBIGUOUSLY DEFINED".

TO OVER COME THIS ERROR THEN WE SHOULD USE A TABLE NAME AS AN IDENTITY

TO AMBIGUOUSE COLUMN CID LIKE BELOW,

SOL:

SQL> SELECT * FROM STUDENT, COURSE WHERE STUDENT.CID=COURSE.CID;

(OR)

SQL> SELECT * FROM STUDENT S, COURSE C WHERE S.CID=C.CID;

RULE OF JOIN:

A ROW IN A FIRST TABLE IS COMPARING THE GIVEN JOIN CONDITION WITH ALL ROWS

OF SECOND TABLE.

EX2:

WAQ TO RETRIEVE STUDENT, COURSE DETAILS FROM TABLES IF CID IS 20?

SOL:

SQL> SELECT * FROM STUDENT S,COURSE C WHERE S.CID=C.CID AND C.CID=20;

EX3:

WAQ TO RETRIEVE LIST OF EMPLOYEES FROM EMP, DEPT TABLES BY USING EQUI JOIN

WHO ARE WORKING IN THE LOCATION IS 'CHICAGO'?

SOL:

SQL> SELECT * FROM EMP E, DEPT D WHERE E. DEPTNO=D.DEPTNO AND LOC='CHICAGO';

EX4:

WAQ TO DISPLAY SUM OF SALARIES OF DEPARTMENTS FROM EMP, DEPT TABLES BY USING EQUI JOIN?

SOL:

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

EX5:

WAQ TO DISPLAY SUM OF SALARIES OF DEPARTMENTS FROM EMP, DEPT TABLES BY USING EQUI JOIN IF SUM OF SALARIES OF DEPARTMENTS ARE MORE THAN 10000?

SOL:

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

USING ROLLUP & CUBE CLAUSES IN JOINS:

EX6:

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

EX7:

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

INNER JOIN:

- INNER JOIN IS SIMILAR TO EQUI JOIN RETRIEVING DATA FROM MULTIPLE

TABLES WITH "ON" CLAUSE CONDITION.

SYNTAX:

```
ON <TABLE NAME1>.<COMMON COLUMN> = <TABLE NAME2>.<COMMON COLUMN>;
```

(OR)

```
ON <TN1 ALIAS NAME>.<COMMON COLUMN> = <TN2 ALIAS NAME>.<COMMON COLUMN>;
```

EX1:

WAQ TO RETRIEVE STUDENT,COURSE DETAILS FROM TABLES BY USING INNER JOIN?

SOL:

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

EX2:

WAQ TO DISPLAY EMPLOYEE FROM EMP,DEPT TABLES BY USING INNER JOIN WHO ARE WORKING IN THE LOCATION IS "CHICAGO" ?

SOL:

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

(OR)

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

WHY ANSI JOINS:

THESE JOINS ARE INTRODUCED IN ORACLE 9I. THE MAIN ADVANTAGE OF ANSI JOINS ARE PORTABILITY. IT MEANS THAT WE CAN MOVE JOIN STATEMENTS FROM ONE DATABASE TO ANOTHER DATABASE WITHOUT MAKING ANY CHANGES AS IT IS THE JOIN STATEMENTS ARE EXECUTED IN OTHER DATABASES.

OUTER JOINS:

- IN THE ABOVE EQUI / INNER JOIN WE ARE RETRIEVING ONLY MATCHING ROWS BUT NOT UN MATCHING ROWS FROM MULTIPLE TABLES. SO TO OVERCOME THIS PROBLEM THEN WE USE "OUTER JOINS" MECHANISM.

THESE ARE AGAIN THREE TYPES:

1. LEFT OUTER JOIN
2. RIGHT OUTER JOIN
3. FULL OUTER JOIN

LEFT OUTER JOIN:

- RETRIEVING ALL ROWS(MATCHING & UN MATCHING) FROM LEFT SIDE TABLE

BUT RETRIEVING MATCHING ROWS FROM RIGHT SIDE TABLE.

ANSI FORMAT:

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

(OR)

NON - ANSI FORMAT:

- WHEN WE WRITE OUTER JOINS IN NON-ANSI FORMAT THEN WE SHOULD USE

JOIN OPERATOR (+).

EX:

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

RIGHT OUTER JOIN:

- RETRIEVING ALL ROWS(MATCHING & UN MATCHING) FROM RIGHT SIDE TABLE BUT RETRIEVING MATCHING ROWS FROM LEFT SIDE TABLE.

ANSI FORMAT:

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

(OR)

NON-ANSI FORMAT:

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

FULL OUTER JOIN:

- RETRIEVING MATCHING AND ALSO UN MATCHING ROWS
FROM BOTH SIDES TABLES.

ANIS FORMAT:

SELECT * FROM STUDENT S FULL OUTER JOIN COURSE C ON
S.CID=C.CID;

(OR)

NON - ANSI FORMAT:

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

ERROR AT LINE 1:

ORA-01468: A PREDICATE MAY REFERENCE ONLY ONE OUTER-
JOINED TABLE

NOTE: NON-ANSI FORMAT IS NOT SUPPORTING FULL OUTER JOIN
MECHANISM. SO THAT WHEN WE WANT TO IMPLEMENT FULL OUTER
JOIN IN NON -ANSI FORMAT THEN WE COMBINED THE RESULTS OF
LEFT OUTER AND RIGHT OUTER JOINS BY USING "UNION"
OPERATOR.

EX:

SELECT * FROM STUDENT S,COURSE C WHERE S.CID=C.CID(+)
UNION

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

NON-EQUI JOIN:

- RETRIEVING DATA FROM MULTIPLE TABLES BASED ON ANY
CONDITION EXCEPT EQUAL OPERATOR CONDITION IS CALLED AS
NON-EQUI JOIN. IN THIE JOIN WE CAN USE THE FOLLOWING
OPERATORS ARE <,>,<=,>=,AND,BETWEEN,.....ETC.

EX1:

NON-ANSI:

SQL> SELECT * FROM TEST1 T1,TEST2 T2 WHERE T1.SNO>T2.SNO;

(OR)

ANSI:

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

EX2:

**WAQ TO DISPLAY ENAME,SALARY,LOW SALARY,HIGH SALARY FROM
EMP,SALGRADE**

TABLES WHOSE SALARY BETWEEN LOW SALARY AND HIGH SALARY ?

SOL:

**SQL> SELECT ENAME,SAL,LOSAL,HISAL FROM EMP,SALGRADE
WHERE SAL BETWEEN LOSAL AND HISAL;**

(OR)

**SQL> SELECT ENAME,SAL,LOSAL,HISAL FROM EMP,SALGRADE
WHERE (SAL>=LOSAL) AND (SAL<=HISAL);**

CROSS JOIN / CARTESIAN JON:

**- JOINING TWO (OR) MORE THAN TWO TABLES WITHOUT ANY
CONDITION IS CALLED AS "CROSS / CARTESIAN JOIN".**

**- IN CROSS JOIN,EACH ROW OF THE FIRST TABLE WILL JOIN
JOINS WITH EACH ROW OF THE SECOND TABLE.THAT MEANS A
FIRST TABLE IS HAVING "M" NO.OF**

**ROWS AND A SECOND TABLE IS HAVING "N" NO.OF ROWS THEN THE
RESULT IS MXN NO.OF ROWS.**

EX1:

SQL> SELECT * FROM STUDENT CROSS JOIN COURSE;---ANSI

(OR)

SQL> SELECT * FROM STUDENT,COURSE;----NON-ANSI

EX2:

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

OUTPUT:

INAME	PRICE	INAME	PRICE
TOTAL_AMOUNT			
PIZZA	250	COCACOLA	25
			275

EX3:

```
SELECT * FROM COLORS CROSS JOIN SIZES;-----ANSI
```

(OR)

```
SELECT * FROM COLORS,SIZES;-----NON - ANSI
```

NATURAL JOIN:

- NATURAL JOIN IS SIMILAR TO EQUI JOIN. WHEN WE USE NATURAL JOIN, WE SHOULD HAVE A COMMON COLUMN NAME. THIS COLUMN DATA TYPE MUST BE MATCH.

- WHENEVER WE ARE USING NATURAL JOIN THERE IS NO NEED TO WRITE A JOINING CONDITION BY EXPLICITLY BECAUSE INTERNALLY ORACLE SERVER IS PREPARING JOINING CONDITION BASED ON AN "EQUAL OPERATOR(=)" WITH COLUMN COLUMN NAME AUTOMATICALLY.

- BY USING NATURAL JOIN WE AVOID DUPLICATE COLUMNS WHILE RETRIEVING DATA FROM MULTIPLE TABLES.

EX:

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

SELF JOIN:

- JOINING A TABLE BY ITSELF IS CALLED AS SELF JOIN. IN SELF JOIN A ROW IN ONE TABLE JOINED WITH THE ROW OF SAME TABLE.

- WHEN WE USE SELF JOIN MECHANISM THEN WE SHOULD CREATE ALIAS NAMES ON A TABLE. ONCE WE CREATE ALIAS NAME ON A TABLE INTERNALLY ORACLE SERVER IS CREATING VIRTUAL TABLE(COPY) ON EACH ALIAS NAME.

- WE CAN CREATE ANY NO.OF ALIAS NAMES ON A SINGLE TABLE BY EACH ALIAS NAME SHOULD BE DIFFERENT NAME.

- SELF JOIN CAN BE IMPLEMENTED AT TWO SITUATIONS:

1. COMPARING A SINGLE COLUMN VALUES BY ITSELF IN THE TABLE.

2. COMPARING TWO DIFFERENT COLUMNS VALUES TO EACH OTHER IN THE TABLE.

EX. ON COMPARING A SINGLE COLUMN VALUES BY ITSELF:

Q: WAQ TO DISPLAY EMPLOYEE WHO ARE WORKING IN THE SAME LOCATION OF THE EMPLOYEE IS "SCOTT" ?

SOL:

```
SQL> SELECT T1.ENAME,T1.LOC FROM TEST T1,TEST T2 WHERE  
T1.LOC=T2.LOC
```

```
AND T2.ENAME='SCOTT';
```

Q: WAQ TO DISPLAY EMPLOYEE WHOSE SALARY IS SAME AS THE SALARY OF THE EMPLOYEE FORD?

SOL:

```
SQL> SELECT E1.ENAME,E1.SAL FROM EMP E1,EMP E2  
WHERE E1.SAL=E2.SAL AND E2.ENAME='FORD';
```

EX. ON COMPARING TWO DIFF.COLUMNS TO EACH OTHER:

EX1:

**WAQ TO DISPLAY MANAGERS AND THEIR EMPLOYEES FROM
EMP TABLE?**

**SSQL> SELECT M.ENAME MANAGER,E.ENAME EMPLOYEES FROM EMP
E,EMP M**

WHERE M.EMPNO=E.MGR;

EX2:

**WAQ TO DISPLAY EMPLOYEE WHO ARE GETTING MORE THAN THEIR
MANAGER SALARY?**

SOL:

**SQL> SELECT M.ENAME MANAGER,M.SAL MSALARY,E.ENAME
EMPLOYEE,E.SAL ESALARY FROM EMP E,EMP M WHERE
M.EMPNO=E.MGR AND E.SAL>M.SAL;**

EX3:

**WAQ TO DISPLAY EMPLOYEE WHO ARE JOINED BEFORE THEIR
MANAGER ?**

SOL:

**SQL> SELECT M.ENAME MANAGER,M.HIREDATE MDOJ,E.ENAME
EMPLOYEE,E.HIREDATE EDOJ FROM EMP E,EMP M WHERE
M.EMPNO=E.MGR AND E.HIREDATE<M.HIREDATE;**

**USING CLAUSE: IN ANSI FORMAT JOINS WHENEVER WE JOIN TWO
OR MORE THAN TWO TABLES INSTEAD OF "ON" CLAUSE WE CAN USE
"USING" CLAUSE ALSO. IT RETURNS COMMON COLUMN ONLY ONE
TIME.**

EX:

**SQL> SELECT * FROM STUDENT S INNER JOIN COURSE C USING
(S.CID);**

ERROR AT LINE 2:

ORA-01748: ONLY SIMPLE COLUMN NAMES ALLOWED HERE

NOTE: WHEN WE USE "USING" CLAUSE WITH COMMON COLUMN NAME THERE IS NO NEED TO PREFIX WITH TABLE ALIAS NAME.

EX:

```
USING(S.CID); -----ERROR
USING(CID); -----ALLOWED
```

EX:

```
SQL> SELECT * FROM STUDENT S INNER JOIN COURSE C
      USING(CID);
```

HOW TO JOIN MORE THAN TWO TABLES:

SYNTAX FOR NON-ANSI JOINS:

```
SELECT * FROM <TN1>,<TN2>,<TN3>,...,...<TN N>
```

```
WHERE <CONDITION1> AND <CONDITION2> AND
<CONDITON3>.....;
```

EQUI JOIN:

```
SQL> SELECT * FROM STUDENT S,COURSE C,REGISTER R
```

```
      WHERE S.CID=C.CID AND C.CID=R.CID;
```

SYNTAX FOR ANSI JOINS:

```
SELECT * FROM <TN1> <JOIN KEY> <TN2> ON <CONDITION1>
<JOIN KEY> <TN3> ON <CONDITION2> <JOIN KEY> <TN4>
.....;
```

INNER JOIN:

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

```
      INNER JOIN REGISTER R ON C.CID=R.CID;
```

(OR)

```
SQL> SELECT * FROM STUDENT S INNER JOIN COURSE C
      USING(CID)
```

```
      INNER JOIN REGISTER R USING(CID);
```

DIFFERENCES BETWEEN JOINS AND SET OPERATORS:

<u>JOINS</u>	<u>SET OPERATORS</u>
1. COMBINED DATA IN COLUMNS WISE.	- COMBINED DATA IN ROWS WISE.
2. COMBINED DATA HORIZONTAL	- COMBINED DATA VERTICALLY
3. DIFFERENT STRUCTURE TABLES TABLES ARE JOINED	- SIMILAR STRUCTURE ARE JOINED.

DATAINTEGRITY:

TO MAINTAIN ACCURATE & CONSISTENCY DATA IN DB TABLES.THIS DATAINTEGRITY AGAIN CLASSIFIED INTO TWO WAYS THOSE ARE

1. DECLARATIVE / PRE -DEFINE DATAINTEGRITY (USING CONSTRAINTS)
2. PROCEDURAL / USER - DEFINE DATAINTEGRITY (USING TRIGGERS)

DECLARATIVE / PRE -DEFINE DATAINTEGRITY:

- THIS DATAINTEGRITY CAN BE IMPLEMENTED WITH HELP OF "CONSTRAINTS". THESE ARE AGAIN THREE TYPES,

- ENTITY INTEGRITY
- REFERENCIAL INTEGRITY
- DOMAIN INTEGRITY

ENTITY INTEGRITY:

- IT ENSURE THAT EACH ROW UNIQUELY IDENTIFY IN A TABLE.TO IMPLEMENT THIS MECHANISM WE USE PRIMARY KEY OR UNIQUE CONSTRAINT.

REFERENCIAL INTEGRITY:

- IT ENSURE THAT TO CREATE RELATIONSHIP BETWEEN TABLES.TO IMPLEMENT THIS MECHANISM THEN WE USE FOREIGN KEY (REFERENCIAL KEY).

DOMAIN INTEGRITY:

- DOMAIN IS NOTHING BUT COLUMN.IT ENSURE THAT TO CHECK VALUES WITH USER DEFINE CONDITION BEFORE ACCEPTING VALUES INTO A COLUMN.TO PERFORM THIS MECHANISM WE USE CHECK, DEFAULT, NOT NULL CONSTRAINTS.

CONSTRAINTS:

- CONSTRAINTS ARE USED TO RESTRICTED UNWANTED(INVALID) DATA INTO TABLE.ALL DATABASES ARE SUPPORTING THE FOLLOWING CONSTRAINT TYPES ARE

- UNIQUE
- NOT NULL
- CHECK
- PRIMARY KEY
- FOREIGN KEY (REFERENCES KEY)
- DEFAULT

- ALL DATABASES ARE SUPPORTING THE FOLLOWING TWO TYPES OF METHODS TO DEFINE CONSTRAINTS.THOSE ARE

I) COLUMN LEVEL:

- IN THIS METHOD WE ARE DEFINING CONSTRAINTS ON INDIVIDUAL COLUMNS.

SYNTAX:

```
CREATE TABLE <TN> (<COLUMN NAME1><DATATYPE>[SIZE] <CONSTRAINT TYPE>, .....);
```

II) TABLE LEVEL:

- IN THIS METHOD WE ARE DEFINING CONSTRAINTS AFTER ALL COLUMNS ARE DECLARED. (END OF THE TABLE DEFINITION)

SYNTAX:

```
CREATE TABLE <TN> (<COLUMN NAME1><DATATYPE>[SIZE], <COLUMN NAME2><DATATYPE>[SIZE], ..... , <CONSTRAINT TYPE>(<COLUMN NAME1>, <COLUMN NAME2>, .....));
```

UNIQUE:

- TO RESTRICTED DUPLICATE VALUES BUT ACCEPTING NULLS INTO A COLUMN.

I) COLUMN LEVEL:

EX:

```
SQL> CREATE TABLE TEST1(SNO INT UNIQUE, NAME VARCHAR2(10) UNIQUE);
```

TESTING:

```
SQL> INSERT INTO TEST1 VALUES(1,'A');---ALLOWED
```

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

```
SQL> INSERT INTO TEST1 VALUES(NULL,NULL);----ALLOWED
```

II) TABLE LEVEL:

EX:

```
SQL> CREATE TABLE TEST2(SNO INT, NAME VARCHAR2(10), UNIQUE (SNO, NAME));
```

TESTING:

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

```
SQL> INSERT INTO TEST2 VALUES(1,'A');---NOT ALLOWED
```

```
SQL> INSERT INTO TEST2 VALUES(2,'A');---ALLOWED
```

NOTE: WHEN WE APPLY UNIQUE CONSTRAINT ON GROUP OF COLUMNS THEN WE CALLED AS "COMPOSITE UNIQUE" CONSTRAINT.IN THIS MECHANISM INDIVIDUAL COLUMNS ARE ACCEPTING DUPLICATE VALUES BUT DUPLICATE COMBINATION OF COLUMNS DATA IS NOT ALLOWED.

NOT NULL:

- TO RESTRICTED NULLS BUT ACCEPTING DUPLICATE VALUES INTO A COLUMN.
- NOT NULL CONSTRAINT NOT SUPPORTS "TABLE LEVEL".

COLUMN LEVEL:

EX:

```
SQL> CREATE TABLE TEST3(STID INT NOT NULL, SNAME VARCHAR2(10) NOT  
NULL);
```

TESTING:

```
SQL> INSERT INTO TEST3 VALUES(101,'A');----ALLOW
```

```
SQL> INSERT INTO TEST3 VALUES(101,'A');---ALLOW
```

```
SQL> INSERT INTO TEST3 VALUES(NULL,NULL);---NOT ALLOW
```

CHECK:

- TO CHECK VALUES WITH USER DEFINED CONDITION BEFORE ACCEPTING VALUES INTO A COLUMN.

I) COLUMN LEVEL:

EX:

```
SQL> CREATE TABLE TEST4(EID INT, SAL NUMBER(10) CHECK(SAL>=10000));
```

TESTING:

```
SQL> INSERT INTO TEST4 VALUES (1021,9500);---NOT ALLOW
```

```
SQL> INSERT INTO TEST4 VALUES (1021,10000);---ALLOW
```

II) TABLE LEVEL:

EX:

```
SQL> CREATE TABLE TEST5(ENAME VARCHAR2(10),SAL NUMBER(10),  
CHECK(ENAME=LOWER(ENAME) AND SAL>8000));
```

TESTING:

```
SQL> INSERT INTO TEST5 VALUES('SAI',7500);---NOT ALLOW
```

```
SQL> INSERT INTO TEST5 VALUES('SAI',9500);---ALLOW
```

PRIMARY KEY:

=====

- TO RESTRICTED DUPLICATES & NULLS INTO A COLUMN.
- A TABLE SHOULD HAVE ONLY "ONE PRIMARY KEY".

I) COLUMN LEVEL:

EX:

```
SQL> CREATE TABLE TEST6(PCODE INT PRIMARY KEY, PNAME VARCHAR2(10)  
PRIMARY KEY);
```

ERROR AT LINE 1:

ORA-02260: TABLE CAN HAVE ONLY ONE PRIMARY KEY.

SOL:

```
SQL> CREATE TABLE TEST6(PCODE INT PRIMARY KEY, PNAME VARCHAR2(10));
```

TESTING:

```
SQL> INSERT INTO TEST6 VALUES(10021,'C');----ALLOW  
SQL> INSERT INTO TEST6 VALUES(10021,'C++');----NOT ALLOW  
SQL> INSERT INTO TEST6 VALUES(NULL,'C++');----NOT ALLOW
```

II) TABLE LEVEL:

EX:

```
SQL> CREATE TABLE TEST7(BCODE INT, BNAME VARCHAR2(10),  
LOC VARCHAR2(10), PRIMARY KEY (BCODE, BNAME));
```

TESTING:

```
SQL> INSERT INTO TEST7 VALUES (1021,'SBI','SRNAGAR');---ALLOW  
SQL> INSERT INTO TEST7 VALUES (1021,'SBI','MADHAPUR');---NOT ALLOW  
SQL> INSERT INTO TEST7 VALUES (1022,'SBI','MADHAPUR');---ALLOW  
SQL> INSERT INTO TEST7 VALUES (1021,'ICICI','SRNAGAR');---ALLOW
```

**NOTE: WHEN WE APPLY PRIMARY KEY CONSTRAINT ON GROUP OF COLUMNS
THEN WE CALLED AS "COMPOSITE PRIMARY KEY" CONSTRAINT. IN THIS
MECHANISM INDIVIDUAL COLUMNS ARE ACCEPTING DUPLICATE VALUES BUT
DUPLICATE COMBINATION OF COLUMNS DATA IS NOT ALLOWED.**

FOREIGN KEY (REFERENCES KEY):

- FOREIGN KEY IS USED TO ESTABLISH RELATIONSHIP BETWEEN TABLES.

BASIC THINGS:

1. WE HAVE A COMMON COLUMN NAME(OPTIONAL) BUT RECOMMENDED.

2. COMMON COLUMN DATATYPE MUST MATCH.

3. ONE TABLE FOREIGN KEY MUST BELONGS TO ANOTHER TABLE PRIMARY KEY.

AND HERE PRIMARY KEY & FOREIGN KEY COLUMN MUST BE COMMON COLUMN.

4. PRIMARY KEY TABLE IS CALLED AS "PARENT TABLE" AND FOREIGN KEY TABLE IS CALLED AS "CHILD TABLE"(I.E PARENT & CHILD RELATIONSHIP).

5. FOREIGN KEY COLUMN VALUES SHOULD BE MATCH WITH PRIMARY KEY COLUMN

VALUES ONLY.

6. GENERALLY PRIMARY KEY IS NOT ALLOWED DUPLICATE AND NULL VALUES WHERE AS FOREIGN KEY IS ALLOWED DUPLICATE & NULL VALUES.

I) COLUMN LEVEL:

SYNTAX:

<COMMON COLUMN NAME OF CHILD> <DT>[SIZE] REFERENCES

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

EX:

STEP1:

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

STEP2:

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

STEP3:

```
SQL> CREATE TABLE EMP1(EID INT PRIMARY KEY, ENAME VARCHAR2(10),  
DEPTNO INT REFERENCES DEPT1(DEPTNO));
```

STEP4:

```
SQL> INSERT INTO EMP1 VALUES (1021,'SAI',10);  
SQL> INSERT INTO EMP1 VALUES (1022,'JONES',10);
```

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

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

1) INSERTION RULE:

- WE CANNOT INSERT VALUES INTO FOREIGN KEY(REFERENCES KEY) COLUMN THOSE VALUES ARE NOT EXISTING UNDER PRIMARY KEY COLUMN OF PARENT TABLE.

EX:

SQL> INSERT INTO EMP1 VALUES (1026,'SCOTT',30);

ERROR AT LINE 1:

ORA-02291: INTEGRITY CONSTRAINT (SCOTT.SYS_C005468) VIOLATED - PARENT KEY NOT FOUND.

2) DELETION RULE:

- WHEN WE TRY TO DELETE A RECORD FROM PARENT TABLE AND THOSE ASSOCIATED RECORDS ARE AVAILABLE IN CHILD TABLE THEN ORACLE RETURNS

AN ERROR IS,

EX:

SQL> DELETE FROM DEPT1 WHERE DEPTNO=20;

ERROR AT LINE 1:

ORA-02292: INTEGRITY CONSTRAINT (SCOTT.SYS_C005468) VIOLATED - CHILD RECORD FOUND.

NOTE:

IF WE WANT TO DELETE A RECORD FROM PARENT TABLE WHEN THEY HAVE CORRESPONDING CHILD RECORDS IN CHILD TABLE THEN WE PROVIDE SOME SET OF RULES TO PERFORM DELETE OPERATIONS ON PARENT TABLE.THOSE RULES ARE CALLED AS "CASCADE RULES".

I) ON DELETE CASCADE

II) ON DELETE SET NULL

I) ON DELETE CASCADE:

- WHENEVER WE ARE DELETING A RECORD FROM PARENT TABLE THEN THAT ASSOCIATED CHILD RECORDS ARE DELETED FROM CHILD TABLE AUTOMATICALLY.

EX:

STEP1:

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

STEP2:

```
SQL> INSERT INTO DEPT2 VALUES (10,'ORACLE');  
SQL> INSERT INTO DEPT2 VALUES (20,'JAVA');
```

STEP3:

```
SQL> CREATE TABLE EMP2(EID INT PRIMARY KEY, ENAME VARCHAR2(10),  
DEPTNO INT REFERENCES DEPT2(DEPTNO) ON DELETE CASCADE);
```

STEP4:

```
SQL>INSERT INTO EMP2 VALUES (1021,'SAI',10);  
SQL>INSERT INTO EMP2 VALUES (1022,'JONES',10);  
SQL>INSERT INTO EMP2 VALUES (1023,'MILLER',20);
```

TESTING:

```
SQL> DELETE FROM DEPT2 WHERE DEPTNO=20; ----ALLOWED
```

II) ON DELETE SET NULL:

WHENEVER WE ARE DELETING A RECORD FROM PARENT TABLE THEN THAT ASSOCIATED CHILD RECORDS ARE SET TO NULL IN CHILD TABLE AUTOMATICALLY.

EX:

STEP1:

```
SQL> CREATE TABLE DEPT3(DEPTNO INT PRIMARY KEY,DNAME  
VARCHAR2(10));
```

STEP2:

```
SQL> INSERT INTO DEPT3 VALUES (10,'ORACLE');
SQL> INSERT INTO DEPT3 VALUES (20,'JAVA');
```

STEP3:

```
SQL> CREATE TABLE EMP3(EID INT PRIMARY KEY, ENAME VARCHAR2(10),
DEPTNO INT REFERENCES DEPT3(DEPTNO) ON DELETE SET NULL);
```

STEP4:

```
SQL>INSERT INTO EMP3 VALUES (1021,'SAI',10);
SQL>INSERT INTO EMP3 VALUES (1022,'JONES',10);
SQL>INSERT INTO EMP3 VALUES (1023,'MILLER',20);
```

TESTING:

```
SQL> DELETE FROM DEPT3 WHERE DEPTNO=10; ----ALLOWED
```

SYNTAX FOR TABLE LEVEL:

```
CREATE TABLE <TN>(<COL1><DT>[SIZE], <COL2><DT>[SIZE],
....., FOREIGN KEY(<COL1>, <COL2>, ..... ) REFERENCES
<PARENT TABLE NAME>(<COL1>, <COL2>, .....);
```

PRE-DEFINE CONSTRAINT NAME:

- WHENEVER WE ARE APPLYING CONSTRAINT ON A PARTICULAR COLUMN THEN DB SERVER(SYSTEM) INTERNALLY GENERATE AN UNIQUE ID NUMBER (OR) AN UNIQUE CONSTRAINT KEY NAME AUTOMATICALLY FOR IDENTIFYING A CONSTRAINT.

EX:

```
SQL> CREATE TABLE TEST8(SNO INT PRIMARY KEY, NAME VARCHAR2(10));
```

TESTING:

```
SQL> INSERT INTO TEST8 VALUES(1,'A');---ALLOWED
SQL> INSERT INTO TEST8 VALUES(1,'B');---NOT ALLOWED
```

ERROR:

```
ORA-00001: UNIQUE CONSTRAINT (SCOTT.SYS_C005475) VIOLATED
```

USER DEFINE CONSTRAINT NAME:

- IN PLACE OF PRE-DEFINE CONSTRAINT NAME WE CAN ALSO CREATE A USER DEFINED CONSTRAINT KEY NAME (OR) CONSTRAINT ID FOR IDENTIFYING A CONSTRAINT.

SYNTAX:

```
<COLUMN NAME> <DT>[SIZE] CONSTRAINT <USER DEFINED CONSTRAINT NAME> <CONSTRAINT TYPE>
```

EX:

```
SQL> CREATE TABLE TEST10(SNO INT CONSTRAINT PK_SNO PRIMARY KEY,  
NAME VARCHAR2(10) CONSTRAINT UQ_NAME UNIQUE);
```

TESTING:

```
SQL> INSERT INTO TEST10 VALUES(1,'A');
```

```
SQL> INSERT INTO TEST10 VALUES(1,'B');
```

ERROR AT LINE 1:

```
ORA-00001: UNIQUE CONSTRAINT (SCOTT.PK_SNO) VIOLATED
```

```
SQL> INSERT INTO TEST10 VALUES(2,'A');
```

ERROR AT LINE 1:

```
ORA-00001: UNIQUE CONSTRAINT (SCOTT.UQ_NAME) VIOLATE
```

DATA DICTIONARIES (OR) READ ONLY TABLES:

- WHENEVER WE ARE INSTALLING ORACLE S/W INTERNALLY ORACLE SERVER IS CREATING SOME PRE-DEFINE TABLES ARE CALLED AS "DATA DICTIONARIES". THESE DATA DICTIONARIES ARE USED TO STORE THE INFORMATION ABOUT DB OBJECTS SUCH AS TABLES,INDEXES, VIEWS, SYNONYMS,..... ETC.

- THESE DATA DICTIONARIES ARE SUPPORTING "SELECT" AND "DESC" COMMANDS ONLY. SO THAT DATA DICTIONARIES ARE ALSO CALLED AS "READ ONLY TABLES" IN ORACLE DB.

- IF WE WANT TO VIEW ALL DATA DICTIONARIES IN ORACLE DB THEN WE FOLLOW THE FOLLOWING SYNTAX IS,

SYNTAX:

```
SQL> SELECT * FROM DICT;
```

NOTE1:

IF WE WANT TO VIEW ALL CONSTRAINTS INFORMATION OF A PARTICULAR TABLE THEN WE USE "USER_CONSTRAINTS" DATA DICTIONARY.

EX:

SQL> DESC USER_CONSTRAINTS;

SQL> SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE FROM USER_CONSTRAINTS WHERE TABLE_NAME='TEST10';

CONSTRAINT_NAME	CONSTRAINT_TYPE
PK_SNO	P
UQ_NAME	U

NOTE2:

- IF WE WANT TO VIEW CONSTRAINT NAME ALONG WITH COLUMN NAME OF A PARTICULAR TABLE THEN WE USE "USER_CONS_COLUMNS" DATA DICTIONARY.

EX:

SQL> DESC USER_CONS_COLUMNS;

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

CONSTRAINT_NAME	COLUMN_NAME
UQ_NAME	NAME
PK_SNO	SNO

NOTE3:

TO VIEW A LOGICAL CONDITION OF CHECK CONSTRAINT THEN WE NEED TO CALL "SEARCH_CONDITION" COLUMN FROM "USER_CONSTRAINTS" DATA DICTIONARY.

EX:

SQL> CREATE TABLE TEST11(SNO INT, SAL NUMBER(10) CHECK(SAL>5000));

EX:

```
SQL> DESC USER_CONSTRAINTS;  
SQL> SELECT SEARCH_CONDITION FROM USER_CONSTRAINTS  
      WHERE TABLE_NAME='TEST11';
```

SEARCH CONDITION

SAL>5000

NOTE4:

**TO VIEW ALL COLUMNS INFORMATION OF A PARTICULAR TABLE THEN
WE USE "USER_TAB_COLUMNS" DATADICTONARY.**

EX:

```
SQL> DESC USER_TAB_COLUMNS;  
SQL> SELECT COLUMN_NAME FROM USER_TAB_COLUMNS  
      WHERE TABLE_NAME='EMP';
```

HOW TO FIND NO. OF ROWS IN A TABLE:

```
SQL> SELECT COUNT (*) FROM EMP;
```

COUNT (*)

14

HOW TO FIND NO. OF COLUMNS IN A TABLE:

```
SQL> SELECT COUNT (*) FROM USER_TAB_COLUMNS WHERE  
      TABLE_NAME='EMP';
```

COUNT (*)

8

HOW TO ADD CONSTRAINTS TO AN EXISTING TABLE:

SYNTAX:

=====

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

EX:

```
SQL> CREATE TABLE TEST12(EID INT, ENAME VARCHAR2(10), SAL  
NUMBER(10));
```

I) ADDING PRIMARY KEY:

```
SQL> ALTER TABLE TEST12 ADD CONSTRAINT PK_EID PRIMARY KEY(EID);
```

II) ADDING UNIQUE, CHECK CONSTRAINT:

```
SQL> ALTER TABLE TEST12 ADD CONSTRAINT UQ_ENAME UNIQUE(ENAME);
```

```
SQL> ALTER TABLE TEST12 ADD CONSTRAINT CHK_SAL CHECK(SAL=10000);
```

III) ADDING "NOT NULL" CONSTRAINT:

SYNTAX:

```
ALTER TABLE <TN> MODIFY <COLUMN NAME> CONSTRAINT <CONSTRAINT  
KEY NAME>
```

```
NOT NULL;
```

EX:

```
SQL> ALTER TABLE TEST12 MODIFY ENAME CONSTRAINT NN_ENAME NOT  
NULL;
```

IV) ADDING FOREIGN KEY CONSTRAINT:

SYNTAX:

```
ALTER TABLE <TN> ADD CONSTRAINT <CONSTRAINT KEY NAME>FOREIGN KEY  
(<COMMON COLUMN OF CHILD TABLE>) REFERENCES<PARENT TABLE>  
(<COMMON COLUMN OF PARENT TABLE>) ON DELETE CASCADE / ON DELETE  
SET NULL;
```

EX:

```
SQL> CREATE TABLE TEST13(DNAME VARCHAR2(10), EID INT);
```

```
TABLE CREATED.
```

EX:

```
SQL> ALTER TABLE TEST13 ADD CONSTRAINT FK_EID FOREIGN KEY(EID)  
REFERENCES TEST12(EID) ON DELETE CASCADE;
```

HOW TO DROP CONSTRAINT FROM AN EXISTING TABLE:

SYNTAX:

```
ALTER TABLE <TN> DROP CONSTRAINT <CONSTRAINT KEY NAME>;
```

I) DROPPING PRIMARY KEY:

METHOD1:

```
SQL> ALTER TABLE TEST13 DROP CONSTRAINT FK_EID; -----FIRST  
SQL> ALTER TABLE TEST12 DROP CONSTRAINT PK_EID; -----LATER
```

METHOD2:

- WHEN WE DROP PRIMARY KEY ALONG WITH FOREIGN KEY CONSTRAINT FROM PARENT AND CHILD TABLES THEN WE USE "CASCADE" STATEMENT.

EX:

```
SQL> ALTER TABLE TEST12 DROP CONSTRAINT PK_EID CASCADE;
```

II) DROPPING UNIQUE, CHECK, NOT NULL CONSTRAINT:

```
SQL> ALTER TABLE TEST12 DROP CONSTRAINT UQ_ENAME;
```

```
SQL> ALTER TABLE TEST12 DROP CONSTRAINT CHK_SAL;
```

```
SQL> ALTER TABLE TEST12 DROP CONSTRAINT NN_ENAME;
```

HOW TO RENAME CONSTRAINT NAME:

SYNTAX:

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

EX:

```
SQL> CREATE TABLE TEST14(SNO INT PRIMARY KEY);  
SQL> SELECT CONSTRAINT_NAME FROM USER_CONS_COLUMNS  
      WHERE TABLE_NAME='TEST14';
```

CONSTRAINT_NAME

```
-----  
SYS_C005489
```

```
SQL> ALTER TABLE TEST14 RENAME CONSTRAINT SYS_C005489 TO SNO_PK;  
SQL> SELECT CONSTRAINT_NAME FROM USER_CONS_COLUMNS  
      WHERE TABLE_NAME='TEST14';
```

CONSTRAINT_NAME
SNO_PK

HOW DISABLE / ENABLE CONSTRAINT:

- BY DEFAULT, CONSTRAINTS ARE ENABLE MODE. IF WE WANT TO DISABLE CONSTRAINT TEMP. THEN WE USE "DISABLE" KEYWORD. IT MEANS THAT CONSTRAINT IS EXISTING IN DB BUT NOT WORK TILL IT MAKE AS "ENABLE".

- WHENEVER WE WANT TO COPY HUGE AMOUNT OF DATA FROM ONE TABLE TO ANOTHER TABLE THERE WE USE "DISABLE" KEYWORD.

SYNTAX:

```
ALTER TABLE <TN> DISABLE / ENABLE CONSTRAINT <CONSTRAINT KEY NAME>;
```

EX:

```
SQL> CREATE TABLE TEST15(ENAME VARCHAR2(10), SAL NUMBER (10)  
      CHECK(SAL=5000));
```

```
SQL> INSERT INTO TEST15 VALUES('SAI',5000); ----ALLOWED
```

```
SQL> INSERT INTO TEST15 VALUES('JONES',3000); -----NOT ALLOWED
```

ERROR AT LINE 1:

```
ORA-02290: CHECK CONSTRAINT (SCOTT.SYS_C005492) VIOLATED
```

```
SQL> ALTER TABLE TEST15 DISABLE CONSTRAINT SYS_C005492;  
SQL> INSERT INTO TEST15 VALUES('JONES',3000); ----ALLOWED
```

EX:

```
SQL> ALTER TABLE TEST15 ENABLE CONSTRAINT SYS_C005492;  
ERROR AT LINE 1:
```

ORA-02293: CANNOT VALIDATE (SCOTT.SYS_C005492) - CHECK CONSTRAINT VIOLATED

- TO OVERCOME THE ABOVE PROBLEM THEN WE USE "NOVALIDATE" KEYWORD AT THE TIME OF ENABLE CONSTRAINT. ONCE WE USE "NOVALIDATE" KEYWORD THEN CONSTRAINT IS ENABLE WITH "NOVALIDATE" AND ORACLE SERVER WILL NOT CHECK EXISTING DATA IN TABLE BUT CHECKING NEW DATA WHILE INSERTING TIME.

EX:

```
SQL> ALTER TABLE TEST15 ENABLE NOVALIDATE CONSTRAINT SYS_C005492;  
TABLE ALTERED.
```

TESTING:

```
SQL> INSERT INTO TEST15 VALUES('SCOTT',6000); ---NOT ALLOWED  
ERROR AT LINE 1:
```

ORA-02290: CHECK CONSTRAINT (SCOTT.SYS_C005492) VIOLATED

```
SQL> INSERT INTO TEST15 VALUES('SCOTT',5000); -----ALLOWED
```

DEFAULT CONSTRAINT:

- IT A SPECIAL TYPE OF CONSTRAINT WHICH IS USED TO ASSIGN A USER DEFINE DEFAULT VALUE TO A COLUMN.

SYNTAX:

```
<COLUMN NAME> <DATATYPE>[SIZE] DEFAULT <VALUE /  
EXPRESSION>
```

EX:

```
SQL> CREATE TABLE TEST17(SNO INT, SAL NUMBER (10) DEFAULT 5000);  
TABLE CREATED.
```

TESTING:

```
SQL> INSERT INTO TEST17 VALUES (1,8500);  
SQL> INSERT INTO TEST17(SNO)VALUES (2);
```

OUTPUT:

SNO	SAL
1	8500
2	5000

HOW TO ADD DEFAULT VALUE TO AN EXISTING TABLE:

SYNTAX:

```
ALTER TABLE <TN> MODIFY <COLUMN NAME> DEFAULT <VALUE /  
EXPRESSION>;
```

EX:

```
SQL> CREATE TABLE TEST18(EID INT, SAL NUMBER (10));
```

```
SQL> ALTER TABLE TEST18 MODIFY SAL DEFAULT 8000;
```

TESTING:

```
SQL> INSERT INTO TEST18(EID)VALUES (1021);
```

NOTE:

- IF WE WANT TO VIEW DEFAULT VALUE OF A COLUMN THEN WE USE
"USER_TAB_COLUMNS" DATADICTONARY.

EX:

```
SQL> DESC USER_TAB_COLUMNS;
```

```
SQL> SELECT COLUMN_NAME, DATA_DEFAULT FROM USER_TAB_COLUMNS  
WHERE TABLE_NAME='TEST18';
```

COLUMN_NAME	DATA_DEFAULT
SAL	8000

HOW TO REMOVE DEFAULT VALUE OF A COLUMN:

EX:

```
ALTER TABLE TEST18 MODIFY SAL DEFAULT NULL;
```

COLUMN_NAME	DATA_DEFAULT
SAL	NULL

TRANSACTION CONTROL LANGUAGE (TCL)

TRANSACTION:

- A TRANSACTION IS A UNIT OF WORK THAT IS PERFORMED AGAINST DATABASE.

Ex:

- IF WE ARE INSERTING / UPDATING / DELETING DATA TO / FROM A TABLE THEN WE ARE PERFORMING A TRANSACTION ON A TABLE.

- TO MANAGE TRANSACTIONS ON DATABASE TABLES THEN WE PROVIDE THE FOLLOWING COMMAND ARE

- 1) COMMIT**
- 2) ROLLBACK**
- 3) SAVEPOINT**

COMMIT:

- THIS COMMAND IS USED TO MAKE A TRANSACTION IS PERMANENT.THESE ARE TWO TYPES.

i) IMPLICIT COMMIT:

- THESE TRANSACTIONS ARE COMMITTED BY SYSTEM (ORACLE DB) BY DEFAULT.

Ex: DDL COMMANDS

ii) EXPLICIT COMMIT:

- THESE TRANSACTIONS ARE COMMITTED BY USER AS PER REQUIREMENT.

Ex: DML COMMANDS

EX:

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

STEP1:

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

SQL> COMMIT;

STEP2:

```
SQL> UPDATE BRANCH SET BLOC='MUMBAI' WHERE  
BCODE=1021;
```

```
SQL> COMMIT;
```

STEP3:

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

```
SQL> COMMIT;
```

NOTE: THE ABOVE DML OPERATIONS ARE NOT POSSIBLE TO "ROLLBACK" BECAUSE THOSE OPERATIONS ARE COMMITTED BY USER EXPLICITLY.

ROLLBACK:

- THIS COMMAND IS USED TO CANCEL TRANSACTION.BUT ONCE A TRANSACTION IS COMMITTED THEN WE CANNOT "ROLLBACK(CANCEL)".

EX:

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

```
SQL> ROLLBACK;
```

NOTE: THE ABOVE "DELETE" OPERATION IS NOT COMMITTED SO THAT USER HAS A CHANCE TO ROLLBACK THAT OPERATION.

RULE OF TRANSACTION:

- THE RULE OF TRANSACTION TELLS THAT EITHER ALL THE STATEMENTS IN THE TRANSACTION SHOULD BE EXECUTED (ALL ARE COMMITTED) SUCCESSFULLY OR NONE OF THOSE STATEMENTS TO BE EXECUTED. (i.e., ALL ARE CANCELLED)

SAVEPOINT:

- WHENEVER A USER CREATE SAVEPOINT WITH IN THE TRANSACTION THEN INTERNALLY SYSTEM IS ALLOCATING A SPECIAL MEMORY FOR A SAVEPOINT AND STORE A TRANSACTION INFORMATION WHICH WE WANT TO ROLLBACK(CANCEL).

HOW TO CREATE A SAVEPOINT:

SYNTAX:

SQL> SAVEPOINT <POINTER NAME>;

HOW TO ROLLBACK A SAVEPOINT:

SYNTAX:

SQL> ROLLBACK TO <POINTER NAME>;

EX1:

SQL> DELETE FROM BRANCH WHERE BCODE=1021;

SQL> DELETE FROM BRANCH WHERE BCODE=1025;

SQL> SAVEPOINT S1;

Save point created.

SQL> DELETE FROM BRANCH WHERE BCODE=1023;

CASE1:

=====

SQL> ROLLBACK TO S1; -----1023 RECORD ONLY

CASE2:

=====

SQL> ROLLBACK; ----- 1021,1025 ROLLBACK

(OR)

SQL> COMMIT; ----- 1021,1025 COMMITTED

EX2:

SQL> DELETE FROM BRANCH WHERE BCODE=1021;

SQL> SAVEPOINT S1;

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

CASE1:

SQL> ROLLBACK TO S1; ----- 1023,1025 RECORDS ARE ROLLBACK

CASE2:

SQL> ROLLBACK; -----1021 ROLLBACK

(OR)

SQL> COMMIT; -----1021 COMMITTED

NOTE:

**- GENERALLY, ALL DATABASES ARE MAINTAINING "ACID"
PROPERTIES BY DEFAULT TO MAINTAIN
ACCURATE AND CONSISTENCY DATA.**

ATOMICITY:

**THE ENTIRE TRANSACTION TAKES PLACE AT ONCE OR DOES NOT
HAPPEN AT ALL.**

CONSISTENCY:

**THE DATABASE MUST BE CONSISTENT BEFORE AND AFTER THE
TRANSACTION.**

ISOLATION:

**MULTIPLE TRANSACTIONS OCCURE INDEPENDENTLY WITHOUT
INTERFERENCE.**

DURABILITY:

**MEANS ONCE A TRANSACTION HAS BEEN COMMITTED IT WILL
REMAIN SO, EVEN IN THE EVENT OF ERRORS, POWER LOSS etc**

SUBQUERY / NESTED QUERY:

> A QUERY INSIDE ANOTHER QUERY IS CALLED AS SUBQUERY OR NESTED QUERY.

> A SUBQUERY IS HAVING TWO MORE QUERIES THOSE ARE,

- I) INNER / CHILD / SUB QUERY
- II) OUTER / PARENT / MAIN QUERY

SYNTAX:

```
SELECT * FROM <TN> WHERE <CONDITION> (SELECT * FROM .....);
```

> AS PER THE EXECUTION PROCESS OF SUBQUERY IT AGAIN CLASSIFIED INTO TWO CATEGORISED.

1) NON - CORELATED SUBQUERIES:

- IN NON-CO RELATED SUBQUERY FIRST INNER QUERY IS EXECUTED AND RETURN A VALUE BASED ON RETURN VALUE OF INNER QUERY LATER OUTER QUERY WILL EXECUTE AND PRODUCE THE FINAL RESULT.

2) CORELATED SUBQUERIES:

- IN CO RELATED SUBQUERY FIRST OUTER QUERY IS EXECUTED AND RETURN VALUES BASED ON RETURN VALUES OF OUTER QUERY LATER INNER QUERY WILL EXECUTE AND PRODUCE THE FINAL RESULT.

TYPES OF NON - CORELATED SUBQUERIES:

- > SINGLE ROW SUBQUERY
- > MULTIPLE ROW SUBQUERY
- > MULTIPLE COLUMN SUBQUERY
- > INLINE VIEW

SINGLE ROW SUBQUERY:

WHEN A SQ RETURNS A SINGLE VALUE IS CALLED AS SRSQ.

IN SRSQ WE CAN USE OPERATORS ARE =, <, >, <=, >=, !=.

EX1:

WHO ARE GETTING THE FIRST HIGH SALARY FROM EMP TABLE?

SOL:

```
=====
|| SUBQUERY = OUTER + INNER ||
=====
```

STEP1:(INNER QUERY):

SELECT MAX(SAL) FROM EMP;

STEP2:(OUTER QUERY):

SELECT * FROM EMP WHERE SAL= (INNER QUERY);

STEP3: (SUBQUERY= OUTER+INNER):

SELECT * FROM EMP WHERE SAL= (SELECT MAX(SAL) FROM EMP);

EX2:

WHOSE EMPLOYEE JOB IS SAME AS THE JOB OF 'SMITH'?

SOL:

SQL> SELECT * FROM EMP WHERE JOB= (SELECT JOB FROM EMP WHERE ENAME='SMITH');

EX3:

WHOSE SALARY IS MORE THAN MAX.SALARY OF THE JOB IS "SALESMAN"?

SOL:

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

EX4:

WHOSE EMPLOYEE JOB IS SAME AS THE JOB OF "BLAKE" AND WHO ARE EARNING

SALARY MORE THAN "BLAKE" SALARY?

SOL:

SELECT * FROM EMP WHERE JOB= (SELECT JOB FROM EMP WHERE ENAME='BLAKE')

AND SAL> (SELECT SAL FROM EMP WHERE ENAME='BLAKE');

EX5:

DISPLAY SENIOR EMPLOYEE?

SOL:

```
SQL> SELECT * FROM EMP WHERE HIREDATE=(SELECT MIN(HIREDATE) FROM EMP);
```

EX6:

TO FIND SECOND HIGH. SALARY FROM EMP TABLE ?

SOL:

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

MAX(SAL)

3000

EX7:

**WASQ DISPLAY EMPLOYEE DETAILS WHO ARE GETTING
SECOND HIGH. SALARY IN EMP TABLE?**

SOL:

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

EX8:

**DISPLAY EMPLOYEE DETAILS WHO ARE GETTING
3RD HIGH. SALARY IN EMP TABLE?**

SOL:

```
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)));
```

EX9:

TO DISPLAY NO. OF EMPLOYEE OF DEPARTMENT NUMBERS. IN WHICH DEPTNO NO. OF EMPLOYEE IS LESS THAN THE NO. OF EMPLOYEE OF DEPTNO IS 20?

SOL:

```
SQL> SELECT DEPTNO,COUNT(*) FROM EMP GROUP BY DEPTNO  
      HAVING COUNT(*)<(SELECT COUNT(*) FROM EMP WHERE DEPTNO=20);
```

EX10:

SUM OF SALARY OF JOBS. IF SUM OF SALARY OF JOBS ARE MORE THAN SUM OF SALARY OF THE JOB IS 'CLERK'?

SOL:

```
SQL> SELECT JOB,SUM(SAL) FROM EMP GROUP BY JOB  
      HAVING SUM(SAL)>(SELECT SUM(SAL) FROM EMP WHERE JOB='CLERK');
```

SUBQUERY WITH "UPDATE":

EX11:

TO UPDATE EMPLOYEE SALARY WITH MAX.SALARY OF EMP TABLE WHOSE EMPNO IS 7900?

SOL:

```
SQL> UPDATE EMP SET SAL=(SELECT MAX(SAL) FROM EMP) WHERE  
EMPNO=7900;
```

SUBQUERY WITH "DELETE":

EX12:

WASQ TO DELETE EMPLOYEE DETAILS FROM EMP TABLE WHOSE JOB IS SAME AS THE JOB OF 'SCOTT'?

SOL:

```
SQL> DELETE FROM EMP WHERE JOB=(SELECT JOB FROM EMP WHERE  
ENAME='SCOTT');
```

MULTIPLE ROW SUBQUERY:

WHEN A SQ RETURNS MORE THAN ONE VALUE IS CALLED AS MRSQ. IN THIS SUB QUERY WE CAN USE THE FOLLOWING OPERATORS ARE IN, ANY, ALL.

EX1:

**TO DISPLAY EMPLOYEE DETAILS WHOSE EMPLOYEE JOB IS SAME AS
THE JOB OF THE EMPLOYEE "SMITH","CLARK"?**

SOL:

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

EX2:

DISPLAY EMPLOYEE DETAILS WHO ARE GETTING MIN, MAX SALARIES?

SOL:

```
SELECT * FROM EMP WHERE SAL IN(  
SELECT MIN(SAL) FROM EMP  
UNION  
SELECT MAX(SAL) FROM EMP);
```

EX3:

TO DISPLAY THE SENIOR MOST EMPLOYEES FROM EACH DEPTNO WISE?

SOL:

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

EX4:

**TO DISPLAY EMPLOYEES WHO ARE EARNING HIGHEST SALARY FROM EACH JOB
WISE?**

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

WORKING WITH "ANY", "ALL" OPERATORS:

ANY: IS USED TO SATISFIED ANY OF THE VALUES IN THE GIVEN

LIST WITH USER DEFINED CONDITION.

EX: A > ANY (10,20,30)

A = 40 - TRUE

A = 09 - FALSE

A = 25 - TRUE

A < ANY (10,20,30)

A = 40 - FALSE

A = 09 - TRUE

A = 25 - TRUE

ALL: IS USED TO SATISFY ALL OF THE VALUES IN THE GIVEN LIST WITH USER DEFINED CONDITION.

EX: A > ALL (10,20,30)

A = 40 - TRUE

A = 09 - FALSE

A = 25 - FALSE

A < ALL (10,20,30)

A = 40 - FALSE

A = 09 - TRUE

A = 25 - FALSE

EX:

WASQ TO DISPLAY EMPLOYEES WHOSE SALARY IS MORE THAN ANY "SALESMAN" SALARY?

SOL:

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

EX:

WASQ TO DISPLAY EMPLOYEES WHOSE SALARY IS MORE THAN OF ALL "SALESMAN" SALARY?

SOL:

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

EX:

TO DISPLAY EMPLOYEES WHO ARE EARNING HIGHEST SALARY FROM EACH JOB WISE BY USING MULTIPLE ROW SUBQUERY?

SQL> UPDATE EMP SET SAL=1300 WHERE EMPNO=7902;

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

OUTPUT:

JOB	SAL
ANALYST	1300
CLERK	1300
SALESMAN	1600
MANAGER	2975
ANALYST	3000
PRESIDENT	5000

NOTE: IN THE ABOVE EXAMPLE WHEN WE ARE COMPARING GROUP OF VALUES BY USING MULTIPLE ROW SUBQUERY THEN ORACLE RETURNS WRONG RESULT. TO OVERCOME THIS PROBLEM THEN WE USE MULTIPLE COLUMN SUBQUERY.

MULTIPLE COLUMN SUBQUERY:

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

SYNTAX:

**SELECT * FROM <TN> WHERE(<COL1>,<COL2>,.....)
IN (SELECT <COL1>,<COL2>,..... FROM <TN>);**

EX:

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

OUTPUT:

JOB	SAL
CLERK	1300
SALESMAN	1600
MANAGER	2975
ANALYST	3000
PRESIDENT	5000

EX:

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

SOL:

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

PSEUDO COLUMNS:

PSEUDO COLUMN IS JUST LIKE A TABLE COLUMN.

- I) ROWID**
- II) ROWNUM**

ROWID:

WHEN WE INSERT A NEW ROW INTO A TABLE INTERNALLY SYSTEM CREATE A UNIQUE ID ADDRESS /NUMBER FOR EACH ROW WISE AUTOMATICALLY.

THESE ROWID'S ARE STORED IN DATABASE SO THAT THESE ARE PERMANENT ID'S.

EX:

SQL> SELECT ROWID, ENAME FROM EMP;

ROWID	ENAME
-----	-----
AAAPLMAEAAAAADAAA	SMITH

SQL> SELECT ROWID, ENAME, DEPTNO FROM EMP WHERE DEPTNO=10;

ROWID	ENAME	DEPTNO
-----	-----	-----
AAAPLMAEAAAAADAAN	MILLER	10

EX:

SQL> SELECT MIN(ROWID) FROM EMP;

MIN(ROWID)

AAAPLMAEAAAAADAAA

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

```
MAX(ROWID)
```

```
-----  
AAAPLMAAEEAAAADAAN
```

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

WHENEVER WE WANT TO DELETE MULTIPLE DUPLICATE ROWS EXCEPT ONE DUPLICATE ROW FROM A TABLE THEN WE USE "ROWID" PSEUDO COLUMN.

EX:

```
SQL> SELECT * FROM TEST;
```

SNO	NAME
10	A
10	A
10	A
20	B
20	B
30	C
30	C
30	C
40	D
40	D
50	E
50	E
50	E

SOL:

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

OUTPUT:**SQL> SELECT * FROM TEST;**

SNO	NAME
10	A
20	B
30	C
40	D
50	E

ROWNUM:

TO GENERATE ROW NUMBERS TO EACH ROW WISE / GROUP OF ROWS WISE AUTOMATICALLY.THESE ROWNUMBERS ARE NOT SAVED IN DB.SO THAT THESE ARE TEMPORARY NUMBERS.

EX:**SQL> SELECT ROWNUM, ENAME FROM EMP;**

ROWNUM	ENAME
1	SMITH
2	ALLEN

SQL> SELECT ROWNUM, ENAME, DEPTNO FROM EMP WHERE DEPTNO=10;

ROWNUM	ENAME	DEPTNO
1	MILLER	10
2	CLARK	10
3	KING	10

EX:

WAQ TO FETCH THE FIRST ROW EMPLOYEE DETAILS FROM EMP TABLE BY USING ROWNUM?

SOL:

SQL> SELECT * FROM EMP WHERE ROWNUM=1;

EX:

WAQ TO FETCH THE SECOND ROW EMPLOYEE DETAILS FROM EMP TABLE BY USING ROWNUM?

SOL:

SQL> SELECT * FROM EMP WHERE ROWNUM=2;

NO ROWS SELECTED

NOTE: GENERALLY, ROWNUM IS ALWAYS START WITH 1 FROM EVERYSELECTED ROW IN A TABLE.SO TO AVOID THIS PROBLEM THEN WE USE <, <= OPERATORS.

EX:

SELECT * FROM EMP WHERE ROWNUM<=2

MINUS

SELECT * FROM EMP WHERE ROWNUM=1;

EX:

WAQ TO FETCH THE FIRST FIVE RCWS FROM EMP TABLE BY USING ROWNUM?

SOL:

SQL> SELECT * FROM EMP WHERE ROWNUM<=5;

EX:

WAQ TO FETCH THE FIFTH ROW EMPLOYEE DETAILS FROM EMP TABLE BY USING ROWNUM?

SOL:

SELECT * FROM EMP WHERE ROWNUM<=5

MINUS

SELECT * FROM EMP WHERE ROWNUM<=4;

EX:

WAQ TO FETCH FROM 3RD TO 9TH ROW FROM EMP TABLE BY USING ROWNUM?

SOL:

SELECT * FROM EMP WHERE ROWNUM<=9

MINUS

SELECT * FROM EMP WHERE ROWNUM<3;

EX:

WAQ TO FETCH THE LAST TWO ROWS FROM EMP TABLE BY ROWNUM?

SOL:

SELECT * FROM EMP WHERE ROWNUM<=14

MINUS .

SELECT * FROM EMP WHERE ROWNUM<=12;

(OR)

SELECT * FROM EMP

MINUS

SELECT * FROM EMP WHERE ROWNUM<=(SELECT COUNT(*)-2 FROM EMP);

INLINE VIEW:

IT IS SPECIAL TYPE OF SUBQUERY.PROVIDING A SELECT QUERY IN PLACE OF TABLE NAME IN SELECT STATEMENT.

IN INLINE VIEW SUBQUERY,THE RESULT OF INNER QUERY WILL ACT AS A TABLE FOR THE OUTER QUERY.

SYNTAX:

SELECT * FROM <TABLE NAME>; -----SQL SELECT QUERY

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

NOTE:

1. GENERALLY SUBQUERY IS NOT ALLOWED TO USE "ORDER BY" CLAUSE.SO THAT WE USE "INLINE VIEW".

2. GENERALLY COLUMN ALIAS NAMES ARE NOT ALLOWED TO USE IN "WHERE" CLAUSE CONDITION.SO THAT WE USE "INLINE VIEW".

USING COLUMN ALIAS NAMES IN WHERE CLAUSE CONDITION:

EX:

WAQ TO DISPLAY EMPLOYEE WHOSE EMPLOYEE ANNUAL SALARY IS MORE THAN 25000?

SOL:

```
SQL> SELECT * FROM (SELECT ENAME,SAL,SAL*12 ANNUSAL FROM EMP)  
WHERE ANNUSAL>25000;
```

USING "ORDER BY" CLAUSE IN SUBQUERY:

EX:

WAQ TO DISPLAY FIRST FIVE HIGHEST SALARIES OF EMPLOYEE FROM EMP TABLE BY USING ROWNUM ALONG WITH INLINE VIEW?

SOL:

```
SQL> SELECT * FROM (SELECT * FROM EMP ORDER BY SAL DESC) WHERE  
ROWNUM<=5;
```

EX:

WAQ TO DISPLAY 5TH HIGHEST SALARY OF EMPLOYEE FROM EMP

TABLE BY USING ROWNUM ALONG WITH INLINE VIEW?

SOL:

```
SELECT * FROM (SELECT * FROM EMP ORDER BY SAL DESC) WHERE  
ROWNUM<=5
```

MINUS

```
SELECT * FROM (SELECT * FROM EMP ORDER BY SAL DESC) WHERE  
ROWNUM<=4;
```

USING ROWNUM ALIAS NAME:

EX:

WAQ TO DISPLAY 3RD POSITION ROW FROM EMP TABLE BY USING ROWNUM ALIAS NAME ALONG WITH INLNE VIEW?

SOL:

```
SQL> SELECT * FROM (SELECT ROWNUM R,ENAME,JOB,SAL FROM EMP) WHERE  
R=3;
```

(OR)

```
SQL> SELECT * FROM (SELECT ROWNUM R,EMP.* FROM EMP) WHERE R=3;
```

EX:

WAQ TO DISPLAY 1ST,3RD,5TH,7TH,9TH ROWS FROM EMP TABLE BY USING ROWNUM ALIAS NAME ALONG WITH INLINE VIEW?

SOL:

```
SQL> SELECT * FROM (SELECT ROWNUM R,EMP.* FROM EMP) WHERE R IN(1,3,5,7,9);
```

EX:

WAQ TO DISPLAY EVEN POSITION ROWS FROM EMP TABLE BY USING ROWNUM ALIAS NAME ALONG WITH INLINE VIEW?

SOL:

```
SQL> SELECT * FROM (SELECT ROWNUM R,EMP.* FROM EMP) WHERE MOD(R,2)=0;
```

EX:

WAQ TO DISPLAY FIRST ROW AND LAST ROW FROM EMP TABLE BY USING ROWNUM ALIAS NAME ALONG WITH INLINE VIEW?

```
SQL> SELECT * FROM (SELECT ROWNUM R,EMP.* FROM EMP) WHERE R=1 OR R=(SELECT COUNT(*) FROM EMP);
```

ANALYTICAL FUNCTIONS:

- ORACLE SUPPORTING THE FOLLOWING THREE TYPES OF ANALYTICAL FUNCTIONS THOSE ARE,

1. **ROW_NUMBER ()**
2. **RANK ()**
3. **DENSE_RANK ()**

THESE ANALYTICAL FUNCTIONS ARE AUTOMATICALLY GENERATE RANKING NUMBERS TO EACH ROW WISE (OR) GROUP OF ROWS WISE EXCEPT ROW_NUMBER (). THIS FUNCTION WILL GENERATE ROW NUMBERS FOR EACH ROW WISE / GROUP OF ROWS WISE.

EX:

<u>ENAME</u>	<u>SALARY</u>	<u>ROW_NUMBER()</u>	<u>RANK()</u>	<u>DENSE_RANK()</u>
A	85000	1	1	1
B	72000	2	2	2
C	72000	3	2	2
D	68000	4	4	3
E	55000	5	5	4

RANK (), DENSE_RANK () WILL ASSIGN SAME RANK NUMBER TO SAME VALUE BUT RANK () WILL SKIP THE NEXT RANK NUMBER IN THE ORDER WHERE AS DENSE_RANK () WILL NOT SKIP THE NEXT RANK NUMBER IN THE ORDER.

SYNTAX:

**ANALYTICAL FUNCTION NAME () OVER () [PARTITION BY <COLUMN NAME>]
ORDER BY <COLUMN NAME> [ASC / DESC])**

HERE,

PARTITION BY CLAUSE IS OPTIONAL.

ORDER BY CLAUSE IS MANDATORY.

WITHOUT PARTITION BY CLAUSE:

EX:

**SQL> SELECT ENAME,SAL,ROW_NUMBER()OVER(ORDER BY SAL DESC) RANKS
FROM EMP;**

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

**SQL> SELECT ENAME,SAL,DENSE_RANK()OVER(ORDER BY SAL DESC) RANKS
FROM EMP;**

WITH PARTITION BY CLAUSE:

EX:

**SELECT ENAME,SAL,DEPTNO,DENSE_RANK()OVER(PARTITION BY DEPTNO
ORDER BY SAL DESC) RANKS FROM EMP;**

OUTPUT:

ENAME	SAL	DEPTNO	RANKS
KING	5000	10	1
CLARK	2450	10	2
MILLER	1300	10	3

EX1:

**WAQ TO DISPLAY 3RD HIGHEST SALARY EMPLOYEE DETAILS FROM EMP TABLE
IN EACH DEPTNO WISE BY USING DENSE_RANK () ALONG WITH INLINE VIEW?**

SOL:

```
SELECT * FROM (SELECT ENAME,SAL,DEPTNO,DENSE_RANK()OVER(PARTITION  
BY DEPTNO ORDER BY SAL DESC) R FROM EMP) WHERE R=3;
```

EX2:

WAP TO DISPLAY THE 4TH SENIOR MOST EMPLOYEE FROM EACH JOB WISE?

SOL:

```
SELECT * FROM (SELECT ENAME, JOB, HIREDATE, DENSE_RANK () OVER  
(PARTITION BY JOB ORDER BY HIREDATE) R FROM EMP) WHERE R=4;
```

2. CO-RELATED SUBQUERY:

- IN CO-RELATED SUBQUERY FIRST OUTER QUERY IS EXECUTED AND LATER INNER QUERY WILL EXECUTE.

SYNTAX TO FIND "NTH" HIGH / LOW SALARY:

```
SELECT * FROM <TN> <TABLE ALIAS NAME1> WHERE N-1= (SELECT COUNT  
(DISTINCT <COLUMN NAME>) FROM <TN> <TABLE ALIAS NAME2> WHERE  
<TABLE ALIAS NAME2>.<COLUMN NAME> (< / >) <TABLE ALIAS  
NAME1>.<COLUMN NAME>);
```

EX1: TO FIND OUT EMPLOYEE WHO ARE GETTING FIRST HIGHEST SALARY FROM EMPLOYEE TABE?

SOL:

```
SELECT * FROM EMPLOYEE E1 WHERE 0= (SELECT COUNT (DISTINCT SAL)  
FROM EMPLOYEE E2 WHERE E2.SAL>E1.SAL);
```

EX2: TO FIND OUT EMPLOYEE WHO ARE GETTING FOURTH HIGHEST SALARY FROM EMPLOYEE TABLE?

SOL:

```
SQL> SELECT * FROM EMPLOYEE E1 WHERE=(SELECT COUNT(DISTINCT SAL)  
FROM EMPLOYEE E2 WHERE E2.SAL>E1.SAL);
```

SYNTAX TO DISPLAY "TOP N" HIGH / LOW SALARIES:

```
SELECT * FROM <TN> <TABLE ALIAS NAME1> WHERE N>(SELECT  
COUNT(DISTINCT <COLUMN NAME>) FROM <TN> <TABLE ALIAS NAME2>  
WHERE <TABLE ALIAS NAME2>.<COLUMN NAME> (< / >) <TABLE ALIAS  
NAME1>.<COLUMN NAME>);
```

EX1:

WAQ TO DISPLAY TOP 3 HIGHEST SALARIES EMPLOYEE DETAILS FROM EMPLOYEE TABLE?

SOL:

```
SQL> SELECT * FROM EMPLOYEE E1 WHERE 3>(SELECT COUNT(DISTINCT SAL) FROM EMPLOYEE E2 WHERE E2.SAL>E1.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 ONLY. THIS OPERATOR IS USED TO CHECK WHETHER ROW / ROWS EXISTS IN THE TABLE OR NOT.

- IT RETRUNS EITHER TRUE (OR) FALSE. IF SUBQUERY RETRUNS AT LEAST ONE ROW THEN RETRUNS TRUE OR ELSE IF SUBQUERY NOT RETRUNS ANY ROW THEN RETURN FALSE.

SYNTAX:

```
WHERE EXISTS (<SELECT STATEMENT>)
```

EX1:

WAQ TO DISPLAY DEPARTMENT DETAILS IN WHICH DEPARTMENT EMPLOYEE ARE WORKING?

SOL:

```
SQL> SELECT * FROM DEPT D WHERE EXISTS (SELECT DEPTNO FROM EMP E WHERE E. DEPTNO=D.DEPTNO);
```

EX2:

WAQ TO DISPLAY DEPARTMENT DETAILS IN WHICH DEPARTMENT EMPLOYEE ARE NOT WORKING?

SOL:

```
SQL> SELECT * FROM DEPT D WHERE NOT EXISTS (SELECT DEPTNO FROM EMP E WHERE E. DEPTNO=D.DEPTNO);
```

SCALAR SUBQUERY:

- SUBQUERIES IN SELECT CLAUSE IS CALLED AS SCALAR SUBQUERY. EVERY SUBQUERY OUTPUT WILL ACT AS A COLUMN.

SYNTAX:

```
-----  
SELECT (SUBQUERY1), (SUBQUERY2), .....FROM <TABLE NAME> [ WHERE <COND>];
```

EX:

```
SQL> SELECT (SELECT COUNT (*) FROM EMP) AS EMPTOTAL, (SELECT COUNT (*)
(*) FROM DEPT) AS DEPTTOTAL FROM DUAL;
```

EMPTOTAL	DEPTTOTAL
-----	-----
14	4

EX:

```
SQL> SELECT (SELECT SUM(SAL) FROM EMP WHERE DEPTNO=10) AS "10",
      (SELECT SUM(SAL) FROM EMP WHERE DEPTNO=20) AS "20",
      (SELECT SUM(SAL) FROM EMP WHERE DEPTNO=30) AS "30"
      FROM DUAL;
```

10	20	30
-----	-----	-----
8750	10875	9400

DB Security

Authentication: Authentication is a process of verifying the credentials (username & password) of a user to login into the system.

Authorization: Authorization is process of verifying whether the user has permissions to perform any operation on the database.

Data Control Language: DCL commands are used to enforce database security in multiple users' database environment. These are two types....

- **GRANT**
- **REVOKE**

GRANT: Grant command is used for giving a privilege or permission for a user to perform operations on the database.

Syntax: GRANT <Privilege Name> on <object name> To {User} ;

Privilege Name: Used to granted permission to the users for some rights are **ALL** and **SELECT**.

Object Name: It is the name of database objects like Table, Views and Stored Procedure etc....

User: Used for to whom an access rights is being granted.

REVOKE: Revoke command removes user access rights / privileges to the database OR taking back the permission that is given to a user.

Syntax: Revoke <privilege name> on <object name > from {user};

WORKING WITH DCL COMMANDS:

STEP1: CONNECT WITH SYSTEM DBA:

SQL> CONN SYSTEM / MANAGER;
CONNECTED.

STEP2: CREATE A NEW USER IN ORACLE DB:

SYNTAX:

SQL> CREATE USER <USER NAME> IDENTIFIED BY <PASSWORD>;

EX:

SQL> CREATE USER A IDENTIFIED BY A;

STEP3: CONNECT WITH USER "A":

SQL> CONN A/A;

ERROR:

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

Warning: You are no longer connected to ORACLE.

NOTE: USER WAS CREATED BUT THIS USER IS DUMMY USER BECAUSE IS NOT HAVING CONNECT PERMISSION.SO PERMISSION MUST BE GIVEN TO USER "A".

STEP4: GRANTING CONNECT PERMISSIONS TO USER "A":

SQL> CONN SYSTEM / MANAGER;

CONNECTED.

SQL> GRANT CONNECT TO A;

CONNECT = TO CONNECT TO ORACLE DB.

STEP5: NOW CONNECT WITH USER "A":

SQL> CONN A/A;

CONNECTED.

STEP6: HOW TO CHANGING USER PASSWORD:

SQL> PASSWORD

Changing password for A

Old password: A

New password: 123

Retype new password: 123

Password changed

SQL> CONN A/123;

Connected.

NOTE:

- 1. PASSWORD CAN BE CHANGED BOTH USER & DBA.**
- 2. PASSWORD CAN BE CHANGED BUT USERNAME CANNOT BE CHANGED.**
- 3. USER NAME IS NOT CASE SENSITIVE BUT PASSWORD IS CASE SENSITIVE.**

STEP7: NOW USER "A" CAN CREATE ITS OWN TABLES:

EX:

SQL> CREATE TABLE TEST1(SNO INT, NAME VARCHAR2(10));

Error: INSUFFICIENT PRIVILEGE.

STEP8: GRANTING CREATE TABLE PERMISSION FROM DBA:

SQL> CONN SYSTEM / MANAGER;

CONNECTED.

SQL> GRANT CREATE TABLE TO A;

GRANTED.

SQL> CONN A/123

EX:

SQL> CREATE TABLE TEST1(SNO INT, NAME VARCHAR2(10));

TABLE CREATED

STEP9: NOW USER "A" CAN INSERT ROWS INT A TABLE:

SQL> INSERT INTO TEST1 VALUES (1021,'SAI');

ERROR: INSUFFICIENT PRIVILEGE ON TABLE SPACE.

STEP10: GRANTING TABLE SPACE PERMISSION FROM DBA:

SQL> GRANT UNLIMITED TABLE SPACE TO A;

GRANTED.

SQL> CONN A/123

SQL> INSERT INTO TEST1 VALUES (1021,'SAI');

SQL> INSERT INTO TEST1 VALUES (1022,'JONES');

SQL> COMMIT;

SQL> UPDATE TEST1 SET NAME='MILLER' WHERE SNO=1022;

SQL> COMMIT;

SQL> DELETE FROM TEST1 WHERE SNO=1022;

SQL> COMMIT;

SQL> SELECT * FROM TEST1;

PRIVILEGES: PRIVILEGE IS RIGHT / PERMISSION GIVEN TO THE USERS.ALL DATABASES ARE HAVING TWO TYPES OF PRIVILEGES.

i) SYSTEM PRIVILEGES

ii) OBJECT PRIVILEGES

i) SYSTEM PRIVILEGES:

> SYSTEM PRIVILEGES ARE GIVEN BY DBA.SUCH AS CREATE SYNONYM, CREATE VIEW, CREATE MATERIALIZED VIEW, CREATE INDEXetc.

SYNTAX:

SQL> GRANT <SYSTEM PRIVILEGE> TO <USER1>;

EX:

SQL> CONN SYSTEM/MANAGER;

Connected.

SQL> GRANT CREATE SYNONYM, CREATE VIEW TO A;

Grant succeeded.

SQL> CONN A/123;

Connected.

SQL> CREATE SYNONYM SYN1 FOR TEST1;

Synonym created.

SQL> CREATE VIEW V1 AS SELECT * FROM TEST1;

View created.

NOTE:IF WE WANT TO VIEW SYSTEM PRIVILEGES RELATED TO USER IN ORACLE DB, THEN WE FOLLOW THE FOLLOWING DATA DICTIONARY IS " SESSION_PRIVS "SYNTAX:

SQL> SELECT * FROM SESSION_PRIVS;

ii) OBJECT PRIVILEGES: OBJECT PRIVILEGES ARE USED TO USERS TO ALLOWED TO PERFORM SOME OPERATIONS ON OBJECT.THESE PRIVILEGES ARE GIVEN BY EITHER DBA (OR) DB DEVELOPER. ORACLE HAVING THE FOLLOWING OBJECT PRIVILEGES ARE INSERT, UPDATE, DELETE, SELECT.

> THESE FOUR OBJECT PRIVILEGES ARE REPRESENTED BY USING "ALL" KEYWORD.

SYNTAX:

GRANT <OBJECT PRIVILEGES> ON <OBJECT NAME> TO USER1;

EX:

SQL> CONN A/123;

SQL> SELECT * FROM DEPT;

(OR)

SQL> SELECT * FROM SYSTEM.DEPT;

ERROR at line 1:

ORA-00942: table or view does not exist

NOTE:USER "A" CANNOT SELECT / ACCESS DATA FROM DEPT TABLE BECAUSE USER "A" IS NOT HAVING PERMISSION OF ACCESSING DATA FROM DEPT.SO THAT WE WANT TO TAKE GRANT SELECT PERMISSION FROM DBA.

EX:

SQL> CONN SYSTEM/MANAGER

Connected.

SQL> GRANT SELECT ON DEPT TO A;

Grant succeeded.

```
SQL> CONN A/123;
Connected.

SQL> SELECT * FROM SYSTEM.DEPT; -----ALLOWED

SQL> INSERT INTO SYSTEM.DEPT VALUES (50,'SAP','INDIA'); ---NOT
ALLOW

SQL> UPDATE SYSTEM.DEPT SET LOC='HYD' WHERE DEPTNO=50; ---
NOT ALLOW

SQL> DELETE FROM SYSTEM.DEPT WHERE DEPTNO=50; ---NOT
ALLOW

NOTE: USER "A" CANNOT PERFORM DML OPERATIONS ON DEPT
TABLE BECAUSE USER "A" DID NOT HAVE PERMISSION FROM DBA.

EX:

SQL> CONN SYSTEM/MANAGER

SQL> GRANT INSERT, UPDATE, DELETE ON DEPT TO A;

SQL> CONN A/123;

SQL> INSERT INTO SYSTEM.DEPT VALUES (50,'SAP','INDIA'); ---
ALLOW

SQL> UPDATE SYSTEM.DEPT SET LOC='HYD' WHERE DEPTNO=50; ---
ALLOW

SQL> DELETE FROM SYSTEM.DEPT WHERE DEPTNO=50; --- ALLOW

NOTE:TO VIEW THE INFORMATION ABOUT PRIVILEGE AND ALSO
GRANTER THEN WE USE FOLLOWING DATADICTONARY "
USER_TAB_PRIVS_MADE".

EX:

SQL> CONN SYSTEM/MANAGER;

SQL> DESC USER_TAB_PRIVS_MADE;

SQL> SELECT GRANTEE, TABLE_NAME, GRANTOR, PRIVILEGE FROM
USER_TAB_PRIVS_MADE;
```

<u>GRANTEE</u>	<u>TABLE NAME</u>	<u>GRANTOR</u>	<u>PRIVILEGE</u>
A	DEPT	SYSTEM	DELETE

NOTE: TO VIEW THE INFORMATION ABOUT PRIVILEGE AND ALSO WHO RECEIVED PERMISSION(GRANTEE) THEN WE USE FOLLOWING DATA DICTIONARY " USER_TAB_PRIVS_RECV".

EX:

```
SQL> CONN A/123;
SQL> DESC USER_TAB_PRIVS_RECV;
SQL> SELECT GRANTOR, PRIVILEGE, TABLE_NAME FROM
USER_TAB_PRIVS_RECV;
```

<u>GRANTOR</u>	<u>PRIVILEGE</u>	<u>TABLE NAME</u>
SYSTEM	UPDATE	DEPT

EX:

```
SQL> CONN SYSTEM/MANAGER
SQL> REVOKE ALL ON DEPT FROM A;
> DBA(SYSTEM) CANCELLED ALL PERMISSIONS OF USER " A ".
```

WITH GRANT OPTION:

WHEN A USER RECEIVING PERMISSIONS "WITH GRANT OPTION" THEN THAT USER ALSO ALLOW TO GIVE OBJECT PRIVILEGE TO ANOTHER USER. EX:

```
SQL> CONN SYSTEM/MANAGER
SQL> GRANT SELECT ON DEPT TO U1;
```

SQL> CONN U1/U1;

SQL> GRANT SELECT ON SYSTEM.DEPT TO U2;

ERROR:

INSUFFICIENT PRIVILEGES TO GRANT TO U2.

> TO OVERCOME THE ABOVE PROBLEM THEN USE "WITH GRANT OPTION " BY SYSTEM.

EX:

SQL> CONN SYSTEM/MANAGER

SQL> GRANT SELECT ON DEPT TO U1 WITH GRANT OPTION;

SQL> CONN U1/U1;

SQL> GRANT SELECT ON SYSTEM.DEPT TO U2;

Grant succeeded.

SQL> CONN U2/U2;

SQL> SELECT * FROM SYSTEM.DEPT; -----ALLOWED

CREATE SESSION;BY USING CREATE SESSION SYSTEM PRIVILEGE

ONLY USER ARE ALLOWED TO CONNECT TO ORACLE DB OTHERWISE

ORACLE SERVER RETURNS AN ERROR.

EX:

SQL> CONN SYSTEM/MANAGER

SQL> CREATE USER U3 IDENTIFIED BY U3;

User created.

SQL> CONN U3/U3;

ERROR:

ORA-01045: user U3 lacks CREATE SESSION privilege; logon denied

> TO OVERCOME THE ABOVE "CREATE SESSION" PRIVILEGE PROBLEM THEN WE FOLLOW THE FOLLOWING SOLUTION IS,

SQL> CONN SYSTEM/MANAGER

SQL> GRANT CREATE SESSION TO U3;

SQL> CONN U3/U3;

CONNECTED.

SQL> CREATE TABLE TEST11(SNO INT, NAME VARCHAR2(10));

ERROR at line 1:

ORA-01031: insufficient privileges

SQL> CONN SYSTEM/MANAGER

SQL> GRANT CREATE TABLE TO U3;

SQL> GRANT UNLIMITED TABLESPACE TO U3;

SQL> CONN U3/U3;

SQL> CREATE TABLE TEST11(SNO INT, NAME VARCHAR2(10));

TABLE CREATED.

SQL> INSERT INTO TEST11 VALUES(1021,'SAI');

SQL> UPDATE TEST11 SET NAME='SAI KUMAR' WHERE SNO=1021;

SQL> DELETE FROM TEST11 WHERE SNO=1021;

SQL> SELECT * FROM TEST11;

NOTE:TO VIEW ALL USERS DETAILES IN ORACLE THEN WE FOLLOW THE FOLLOWING DATADICTONARY IS "ALL_USERS".

EX:

SQL> DESC ALL_USERS;

SQL> SELECT USERNAME FROM ALL_USERS;

SYNTAX TO DROP A USER:

SQL> DROP USER <USER NAME> CASCADE;

EX:

SQL> DROP USER A CASCADE;

ROLE: ROLE IS COLLECTION OF SYSTEM / OBJECT PRIVILEGES AND CREATED BY DBA.

WHY WE NEED TO CREATE ROLE:

IN REALTIME ENVIRONMENT NO. OF USERS WORKING ON SAME PROJECT IN THIS SOME GROUP OF USERS REQUIRES COMMON SET OF SYSTEM PRIVILEGES OR OBJECT PRIVILEGES AT THIS TIME DBA CREATING ROLE AND ASSIGNING THAT ROLE TO THE NO. OF USERS.

STEPS TO CREATE A ROLE:

STEP1: CREATE A ROLE:

SYNTAX:

CREATE ROLE <ROLE NAME>;

STEP2: ASSIGN SYSTEM / OBJECT PRIVILEGES TO A ROLE:

SYNTAX:

GRANT SYSTEM PRIVILEGES / OBJECT PRIVILEGES TO <ROLE NAME>;

STEP3: ASSIGN ROLE TO THE NO. OF USERS:

SYNTAX:

GRANT ROLENAMES TO USER1, USER2, USER3,;

EX:

SQL> CONN SYSTEM/MANAGER

SQL> CREATE ROLE R1;

Role created.

SQL> GRANT CREATE SYNONYM TO R1;

Grant succeeded.

SQL> GRANT R1 TO U1, U2;

Grant succeeded.

NOTE1:TO VIEW SYSTEM PRIVILEGES RELATED TO ROLE THEN WE ARE USING THE FOLLOWING DATADICTONARY IS "ROLE_SYS_PRIVS".

EX:

SQL> DESC ROLE_SYS_PRIVS;

SQL> SELECT ROLE, PRIVILEGE FROM ROLE_SYS_PRIVS WHERE ROLE='R1';

ROLE	PRIVILEGE
-----	-----
R1	CREATE SYNONYM

NOTE2:TO VIEW OBJECT PRIVILEGES RELATED TO ROLE THEN WE USE THE FOLLOWING DATADICTONARY IS " ROLE_TAB_PRIVS "

EX:

SQL> DESC ROLE_TAB_PRIVS;

SQL> SELECT ROLE, PRIVILEGE, TABLE_NAME FROM ROLE_TAB_PRIVS WHERE ROLE='R1';

ROLE	PRIVILEGE	TABLE_NAME
R1	SELECT	DEPT

NOTE3: TO VIEW ROLES GRANTED TO A USER IN ORACLE DB THEN USE THE FOLLOWING DATA DICTIONARY IS "USER_ROLE_PRIVS".

EX:

SQL> CONN U1/U1;

SQL> DESC USER_ROLE_PRIVS;

**SQL> SELECT USERNAME, GRANTED_ROLE FROM
USER_ROLE_PRIVS;**

USERNAME	GRANTED_ROLE
U1	R1

U1 R1

NOTE: "WITH GRANT OPTION " DOESN'T WORK ON ROLE.

EX:

SQL> CONN SYSTEM/MANAGER

SQL> GRANT INSERT ON DEPT TO R1 WITH GRANT OPTION;

ERROR at line 1:

ORA-01926: cannot GRANT to a role WITH GRANT OPTION

SYNTAX TO DROP A ROLE:

SQL> DROP ROLE <ROLE NAME>;

EX:

SQL> DROP ROLE R1;

GRANTING PERMISSIONS TO DIFFERENT LEVELS:

SYNTAX:

GRANT <SYSTEM PRIVILEGES> TO <USER> / <ROLE> / <PUBLIC>;

&

GRANT <OBJECT PRIVILEGES> ON <OBJECT NAME> TO <USER> / <ROLE> / <PUBLIC>;

REVOKING PERMISSIONS TO DIFFERENT LEVELS:

SYNTAX:

REVOKE <SYSTEM PRIVILEGES> FROM <USER> / <ROLE> / <PUBLIC>;

&

REVOKE <OBJECT PRIVILEGES> ON <OBJECT NAME> FROM <USER> / <ROLE> / <PUBLIC>;

SYNONYMS:

IT A DB OBJECT TO CREATE PERMANANT ALIAS NAMES FOR DB OBJECTS LIKE TABLE, VIEW, PROCEDUREETC.

SYNONYM IS NOTHING BUT ALTERNATIVE NAME FOR DB OBJECTS LIKE TABLE, VIEW, PROCEDURE....

SYNONYMS ARE CREATED TO REDUCE LENGTHY TABLE NAME.

TYPES OF SYNONYMS:

1. PRIVATE SYNONYM (DEFAULT)

2. PUBLIC SYNONYM

1. PRIVATE SYNONYMS:

> THESE SYNONYMS ARE CREATED BY USERS WHICH HAVE PERMISSION.

SYNTAX:

CREATE SYNONYM <SYNONYM NAME> FOR [USERNAME]. <DB OBJECT NAME>;

EX:

SQL> CONN SCOTT/TIGER;

SQL> CREATE USER U1 IDENTIFIED BY U1;

SQL> GRANT CONNECT, CREATE TABLE, UNLIMITED TABLESPACE TO U1;

SQL> CONN U1/U1;

SQL> CREATE TABLE STUDENTDETAILS (STID INT, SNAME VARCHAR2(10));

SQL> INSERT INTO STUDENTDETAILS VALUES (1021,'SAI');

SQL> INSERT INTO STUDENTDETAILS VALUES (1022,'JONES');

SQL> COMMIT;

SQL> CREATE SYNONYM SYN1 FOR STUDENTDETAILS;

ERROR AT LINE 1:

ORA-01031: INSUFFICIENT PRIVILEGES

```
SQL> CONN SCOTT/TIGER;  
SQL> GRANT CREATE SYNONYM TO U1;  
GRANT SUCCEEDED.
```

```
SQL> CONN U1/U1;  
SQL> CREATE SYNONYM SYN1 FOR STUDENTDETAILS;  
SYNONYM CREATED.
```

NOTE: ONCE WE CREATED SYNONYM INSTEAD OF USING TABLE NAME WE CAN USE SYNONYM NAME FOR ACCESSING DATA / TO PERFORM DB OPERATIONS ON TABLE.

TESTING:

```
SQL> SELECT * FROM SYN1;  
SQL> INSERT INTO SYN1 VALUES (1023,'MILLER');  
SQL> UPDATE SYN1 SET SNAME='SCOTT' WHERE STID=1022;  
SQL> DELETE FROM SYN1 WHERE STID=1023;
```

2. PUBLIC SYNONYMS:

THESE SYNONYMS ARE CREATED BY DBA. WE SHOULD HAVE "CREATE PUBLIC SYNONYM" PRIVILEGE. AND IT CAN ACCESSING BY ALL USERS. FOR HIDING THE INFORMATION ABOUT USERNAME, OBJECT NAME(TABLE).

SYNTAX:

```
CREATE PUBLIC SYNONYM <SYNONYM NAME> FOR [USER NAME].  
<DB OBJECT NAME>;
```

EX:

```
SQL> CONN SCOTT/TIGER;  
SQL> CREATE PUBLIC SYNONYM PUB_SYN FOR DEPT;
```

```
SQL> GRANT ALL ON PUB_SYN TO U1, U2, U3;
```

(OR)

```
SQL> GRANT ALL ON PUB_SYN TO PUBLIC;
```

```
SQL> CONN U1/U1;
```

```
SQL> SELECT * FROM PUB_SYN;---ALLOWED
```

```
SQL> CONN U2/U2;
```

```
SQL> SELECT * FROM PUB_SYN;---ALLOWED
```

NOTE: ONCE WE CREATED PUBLIC SYNONYM THEN ANY USER CAN ACCESS THAT PUBLIC SYNONYM WITHOUT "USERNAME".

NOTE: TO VIEW ALL SYNONYMS INFORMATION IN ORACLE DB THEN WE USE "USER_SYNONYMS" DATADICTONARY.

EX:

```
SQL> DESC USER_SYNONYMS;
```

```
SQL> SELECT SYNONYM_NAME, TABLE_NAME FROM  
USER_SYNONYMS;
```

NOTE: TO VIEW ALL PRIVATE AND PUBLIC SYNONYMS OF A PARTICULAR USER IN ORACLE DATABASE THEN WE "ALL_SYNONYMS" DATADICTONARY.

EX:

```
SQL> DESC ALL_SYNONYMS;
```

```
SQL> SELECT SYNONYM_NAME, TABLE_NAME FROM  
ALL_SYNONYMS WHERE TABLE_NAME='DEPT';
```

SYNTAX TO DROP SYNONYMS:

SQL> DROP SYNONYM <SYNONYM NAME>;

EX:

SQL> DROP SYNONYM SYN_DEPT;

**IN THIS CASE WE DROPPING PRIVATE SYNONYMS ONLY
AND DROPPING BY USER.**

SQL> DROP PUBLIC SYNONYM <SYNONYM NAME>;

EX:

SQL> DROP PUBLIC SYNONYM PUB_SYN;

**IN THIS CASE WE DROPPING PUBLIC SYNONYMS ONLY AND
DROPPING BY DBA.**

VIEWS:

VIEW IS DB OBJECT IS CALLED SUBSET OF A TABLE. VIEW IS ALSO CALLED AS VIRTUAL TABLE BECAUSE IT DOESN'T STORE DATA AND IT DOESN'T OCCUPY ANY MEMORY.

VIEW IS CREATING BY USING "SELECT QUERY" FOR GETTING THE REQ.INFORMATION FROM TABLE (BASE TABLE).

TYPES OF VIEWS:

A USER CAN CREATE THE FOLLOWING TWO TYPES OF VIEWS ON BASE TABLES THOSE ARE,

1. SIMPLE VIEWS

2. COMPLEX VIEWS

1. SIMPLE VIEWS:

WHEN WE CREATE A VIEW TO ACCESS REQUIRED DATA FROM A SINGLE BASE TABLE IS CALLED AS SIMPLE VIEWS.

THROUGH A SIMPLE VIEW WE CAN PERFORM ALL DML (INSERT, UPDATE, DELETE) OPERATIONS ON BASE TABLE.

SYNTAX:

CREATE VIEW <VIEW NAME> AS SELECT * FROM <TN> [WHERE <CONDITION>];

EX1:

```
SQL> CREATE VIEW SV1 AS SELECT * FROM DEPT;  
SQL> SELECT * FROM SV1;
```

DML OPERATIONS THROUGH A SIMPLE VIEW:

```
SQL> INSERT INTO SV1 VALUES (50,'DBA','HYD');  
SQL> UPDATE SV1 SET LOC='INDIA' WHERE DEPTNO=50;  
SQL> DELETE FROM SV1 WHERE DEPTNO=50;
```

NOTE: WHENEVER WE PERFORM DML OPERATIONS ON VIEW INTERNALLY THE VIEW WILL PERFORM THOSE OPERATIONS ON BASE TABLE. HERE VIEW WILL ACT AS AN INTERFACE BETWEEN USER AND BASE TABLE.

USER <-----> <VIEW> <-----> BASE TABLE

EX2:

**SQL> CREATE VIEW SV2 AS SELECT EMPNO, ENAME, JOB, SAL
FROM EMP;**

TESTING:

**SQL> INSERT INTO SV2 VALUES (1122,'SAI','HR',8000); ---ALLOW
SQL> INSERT INTO SV2 VALUES (1122,'WARNER','SR.HR',9500); -
---NOT ALLOW (EMPNO COLUMN IS PRIMARY KEY COLUMN IN
EMP TABLE)**

WITH CHECK OPTION:

**IT IS A CONSTRAINT WHICH IS USED TO RESTRICT ROWS ON
BASE TABLE THROUGH
A VIEW WHILE PERFORMING DML OPERATIONS.**

EX:

**SQL> CREATE VIEW SV3 AS SELECT * FROM TEST1 WHERE
SAL=18000 WITH CHECK OPTION;**

TESTING:

**SQL> INSERT INTO SV3 VALUES (1025,'SCOTT',12000); ---NOT
ALLOW**

**SQL> INSERT INTO SV3 VALUES (1025,'SCOTT',58000); ---NOT
ALLOW**

**SQL> INSERT INTO SV3 VALUES (1025,'SCOTT',18000); ---
ALLOWED**

WITH READ ONLY:

IF WE CREATED A VIEW "WITH READ ONLY" CLAUSE THEN WE RESTRICT DML OPERATIONS. WE ALLOW "SELECT" AND "DESC" COMMANDS.

EX:

```
SQL> CREATE VIEW SV4 AS SELECT * FROM DEPT WITH READ ONLY;
```

NOTE: NOW WE CANNOT PERFORM DML OPERATIONS THROUGH A VIEW ON BASE TABLE.

2.COMPLEX VIEWS: A VIEW IS CALLED AS COMPLEX VIEW,

I) WHEN WE CREATE ON MULTIPLE BASE TABLES.

II) WHEN WE CREATE A VIEW WITH AGGREGATIVE FUNCTIONS, GROUP BY, HAVING CLAUSES, SET OPERATORS, SUB-QUERY, DISTINCT KEY WORD.

COMPLEX VIEW ARE NOT ALWAYS SUPPORTS DML OPERATIONS.

EX1:

```
SQL> CREATE VIEW CV1 AS SELECT * FROM STUDENT S INNER  
JOIN COURSE C  
ON S.CID=C.CID;
```

ERROR AT LINE 1:

ORA-00957: DUPLICATE COLUMN NAME

NOTE: WHEN WE CREATE A VIEW ON BASE TABLES THEN WE SHOULD NOT ALLOW DUPLICATE COLUMN NAMES. TO AVOID THIS PROBLEM THEN USE "USING" CLAUSE.

```
SQL> CREATE VIEW CV1 AS SELECT * FROM STUDENT S INNER  
JOIN COURSE C USING(CID);
```

NOW WE CREATED A COMPLEX VIEW ON MULTIPLE TABLES. BUT NOT ALLOW DML OPERATIONS.

EX2:

```
SQL> CREATE VIEW CV2 AS  
      SELECT * FROM EMP_HYD  
      UNION  
      SELECT * FROM EMP_CHENNAI;
```

> THE ABOVE COMPLEX VIEW CV2 IS NOT ALLOW DML OPERATIONS.

EX3:

```
SQL> CREATE VIEW CV3 AS SELECT DEPTNO, SUM(SAL) FROM  
EMP GROUP BY DEPTNO;
```

ERROR AT LINE 1:

ORA-00998: MUST NAME THIS EXPRESSION WITH A COLUMN ALIAS

NOTE: WHEN WE CREATE A VIEW WITH FUNCTION THEN WE MUST CREATE ALIAS NAME

FOR THOSE FUNCTIONS OTHERWISE ORACLE RETURNS AN ERROR.

EX:

```
SQL> CREATE VIEW CV3 AS SELECT DEPTNO, SUM(SAL) AS  
SUMSAL FROM EMP  
      GROUP BY DEPTNO;
```

> THE ABOVE COMPLEX VIEW CV3 NOT ALLOWED DML OPERATIONS.

EX4:

```
SQL> CREATE VIEW CV4 AS SELECT EMPNO, ENAME, SAL, D.  
DEPTNO, DNAME, LOC FROM EMP E INNER JOIN DEPT D ON E.  
DEPTNO=D.DEPTNO;
```

TESTING:

```
SQL> UPDATE CV4 SET SAL=500 WHERE EMPNO=7788; ---  
ALLOWED  
SQL> DELETE FROM CV4 WHERE EMPNO=7782; ----ALOOWED  
SQL> INSERT INTO CV4 VALUES (1122,'SAI',6000,10,'SAP','HYD');  
---NOT ALLOW
```

NOTE: GENERALLY COMPLEX VIEW ARE NOT ALLOWED TO PERFORM DML OPERATIONS BUT WE PERFORM UPDATE, DELETE OPERATIONS ON KEY PRESERVED TABLE (I.E PRIMARY KEY) SO THAT COMPLEX VIEWS ARE SUPPORTING DML OPERATION PARTIALLY.

FORCE VIEWS:

GENERALLY, VIEWS ARE CREATED BASED ON TABLES, BUT FORCE VIEWS ARE CREATE WITHOUT TABLES.

SYNTAX:

```
CREATE FORCE VIEW <VIEW NAME> AS SELECT * FROM <TN>;
```

EX:

```
SQL> CREATE FORCE VIEW FV1 AS SELECT * FROM TEST;  
WARNING: VIEW CREATED WITH COMPILATION ERRORS.
```

TESTING:

```
SQL> SELECT * FROM FV1;
```

ERROR AT LINE 1:

ORA-04063: VIEW "SCOTT.FV1" HAS ERRORS

SQL> DESC FV1;

ERROR:

ORA-24372: INVALID OBJECT FOR DESCRIBE

> TO ACTIVATE A FORCE VIEW THEN WE SHOULD CREATE A TABLE WITH THE NAME

AS "TEST".

EX:

SQL> CREATE TABLE TEST (SNO INT, NAME VARCHAR2(10));

TABLE CREATED.

TESTING:

SQL> SELECT * FROM FV1; ----ACTIVATED

SQL> DESC FV1; ----ACTIVATED

NOTE: TO VIEW ALL VIEWS DETAILS IN ORACLE DB THEN WE USE THE FOLLOWING DATA DICTIONARY IS "USER_VIEWS".

EX:

SQL> DESC USER_VIEWS;

SQL> SELECT VIEW_NAME FROM USER_VIEWS;

SYNTAX TO DROP A VIEW:

SQL> DROP VIEW <VIEW NAME>;

EX:

SQL> DROP VIEW SV1;

SQL> DROP VIEW CV1;

SQL> DROP VIEW FV1;

ADVANTAGES OF VIEWS:

- 1. IT IS PROVIDING SECURITY. IT MEANS THAT TO EACH USER CAN BE GIVEN PERMISSION TO ACCESS SPECIFIC COLUMNS & SPECIFIC ROWS FROM A TABLE.**
- 2. IF DATA IS ACCESSED AND ENTERED THROUGH A VIEW, THE DB SERVER WILL CHECK DATA TO ENSURE THAT IT MEETS SPECIFIED INTEGRITY CONSTRAINTS RULES OR NOT.**
- 3. QUERY SIMPLIFY IT MEANS THAT TO REDUCE COMPLEX QUERY.**

DIFFERENCES BETWEEN SYNONYM AND VIEW:

<u>SYNONYM</u>	<u>VIEW</u>
1. IT IS A MIRROR OF TABLE.	1. IT IS A SUBSET OF TABLE.
2. CREATED ON A SINGLE TABLE.	2. CREATED ON MULTIPLE TABLES.
3. CREATE ON ENTIRE TABLE.	3. CREATED ON SPECIFIC ROWS AND SPECIFIC COLUMNS OF TABLE.
4. NOT SUPPORTS DATA ABSTRACTION.	4. SUPPORTING DATA ABSTRACTION MECHANISM. (HIDE DATA)

MATERIALIZED VIEWS:

- ORACLE 8I INTRODUCED MATERIALIZED VIEWS. GENERALLY, VIEWS DOESN'T STORE ANY DATA WHERE AS MATERIALIZED VIEWS ARE STORING DATA.
- THESE VIEWS ARE USED IN DATA WAREHOUSING AND HANDLING BY DBA. MATERIALIZED VIEWS ALSO CREATED FROM BASE TABLES.

DIFFERENCES BETWEEN VIEW AND MATERIALIZED VIEW:

VIEW	MATERIALIZED VIEW
VIEW DOES NOT STORE ANY DATA.	MATERIALIZED VIEW STORE DATA.
WHEN WE DROPPING BASE TABLE THEN VIEW CANNOT BE ACCESSIBLE	WHEN WE DROPPING BASE TABLE THEN MATERIALIZED VIEW CAN BE ACCESSIBLE.
IT IS DEPENDENT OBJECT	IT IS INDEPENDENT OBJECT
WE CAN PERFORM DML OPERATIONS ON VIEW	WE CANNOT PERFORM DML OPERATIONS ON MATERIALIZED VIEW

SYNTAX:

**CREATE MATERIALIZED VIEW <VIEW NAME> AS SELECT * FROM
<TABLE NAME>;**

EX:

SQL> CREATE TABLE TEST1 (SNO INT, NAME VARCHAR2 (20));

**SQL> CREATE VIEW V1 AS SELECT * FROM TEST1;
VIEW CREATED.**

**SQL> CREATE MATERIALIZED VIEW MV1 AS SELECT * FROM TEST1;
MATERIALIZED VIEW CREATED.**

TESTING:

SQL> INSERT INTO TEST1 VALUES (101,'SMITH');

- ✓ HERE, BASE TABLE (TEST1) AND VIEW (V1) TABLE DATA IS UPDATED BUT MATERIALIZED VIEW (MV1) TABLE DATA IS NOT UPDATED. IF WE WANT TO UPDATE DATA IN MATERIALIZED VIEW THEN WE REFRESH MATERIALIZED VIEW BY USING THE FOLLOWING SYNTAX,

SYNTAX:

EXEC DBMS_MVIEW.REFRESH ('MATERIALIZED VIEW NAME');

EX: EXEC DBMS_MVIEW.REFRESH ('MV1');

ON DEMAND / ON COMMIT:

IN ORACLE WE ARE REFRESHING MATERIALIZED VIEW IN TWO WAYS THOSE ARE,

I) ON DEMAND:

IT IS A DEFAULT REFRESHING METHOD. IN THIS METHOD WE ARE REFRESHING MATERIALIZED VIEW BY USING "DBMS_MVIEW" PROCEDURE.

SYNTAX:

EXECUTE DBMS_MVIEW.REFRESH ('MVVIEW NAME');

EX:

EXECUTE DBMS_MVIEW.REFRESH ('MV1');

II) ON COMMIT:

WE CAN REFRESHING A MATERIALIZED VIEW WITHOUT USING "DBMS_MVIEW" BUT USING "ON COMMIT" METHOD.

SYNTAX:

CREATE MATERIALIZED VIEW <VIEW NAME>

REFRESH ON COMMIT

AS SELECT * FROM <TN>;

EX:

```
CREATE TABLE TEST2 (EID INT, SAL NUMBER (10));
```

EX:

```
CREATE MATERIALIZED VIEW MV2 REFRESH ON COMMIT
```

```
AS SELECT * FROM TEST2;
```

ERROR:

TABLE 'TEST2' DOES NOT CONTAIN A PRIMARY KEY CONSTRAINT.

EX:

```
CREATE TABLE TEST2 (EID INT PRIMARY KEY, SAL NUMBER (10));
```

EX:

```
CREATE MATERIALIZED VIEW MV2 REFRESH ON COMMIT
```

```
AS SELECT * FROM TEST2;
```

MATERIALIZED VIEW CREATED.

**NOTE: WHEN WE CREATE MATERIALIZED VIEW ALONG WITH
REFRESH ON COMMIT METHOD ON BASE TABLE THEN BASE TABLE
SHOULD HAVE PRIMARY KEY CONSTRAINT OTHERWISE ORACLE
RETURNS AN ERROR.**

TESTING:

```
SQL> INSERT INTO TEST2 VALUES (101,25000);
```

```
SQL> COMMIT;
```

```
SQL> SELECT * FROM MV2;
```

**NOTE: IF WE WANT TO VIEW MATERIALIZED VIEWS
THEN WE ARE USING THE FOLLOWING DATA DICTIONARY IS
"USER_MVIEWS".**

EX:

```
SQL> DESC USER_MVIEWS;
```

```
SQL> SELECT MVVIEW_NAME FROM USER_MVIEWS;
```

SYNTAX TO DROP MATERIALIZED VIEW:

SQL> DROP MATERIALIZED VIEW <MVVIEW NAME>;

SQL> DROP MATERIALIZED VIEW MV1;

MATERIALIZED VIEW DROPPED.

PARTITION TABLE:

GENERALLY, PARTITIONS ARE CREATED ON VERY LARGE-SCALE DATABASE TABLES FOR DIVIDING INTO MULTIPLE SMALL PARTS AND EACH PART IS CALLED AS "PARTITION".

- BY SPLITTING A LARGE TABLE INTO SMALLER PARTS THEN DATA CAN ACCESS VERY FAST BECAUSE THERE IS LESS DATA TO SCAN INSTEAD OF LARGE DATA OF A TABLE.

TYPES OF PARTITIONS:

- 1) RANGE PARTITION**
- 2) LIST PARTITION**
- 3) HASH PARTITION**

- IF WE WANT TO ACCESS A PARTICULAR PARTITION THEN WE FOLLOW THE FOLLOWING,

SYNTAX:

```
SQL> SELECT * FROM <TN> PARTITION (<PARTITION NAME>);
```

1) RANGE PARTITION:

- IN THIS METHOD WE ARE CREATING PARTITIONS TABLE BASED ON A PARTICULAR RANGE VALUE.

SYNTAX:

```
CREATE TABLE <TN> (<COLUMN NAME1> <DATATYPES>[SIZE],  
.....) PARTITION BY RANGE (<KEY COLUMN  
NAME>) (PARTITION <PARTITION NAME1> VALUES LESS  
THAN(VALUE), PARTITION <PARTITION NAME2> VALUES LESS  
THAN(VALUE), .....);
```

EX:

```
CREATE TABLE TEST1(EID INT, ENAME VARCHAR2(10), SAL  
NUMBER (10)) PARTITION BY RANGE(SAL) (PARTITION P1  
VALUES LESS THAN (1000),PARTITION P2 VALUES LESS THAN  
(2000), PARTITION P3 VALUES LESS THAN (3000));
```

TESTING:

SQL> INSERT INTO TEST1 VALUES(1,'SAI',2500);

SQL> INSERT INTO TEST1 VALUES(2,'JONES',500);

.....;
.....;

CALLING A PARTICULAR PARTITION:

SQL> SELECT * FROM TEST1 PARTITION(P1);

EID	ENAME	SAL
2	JONES	500

2) LIST PARTITION:

- IN THIS METHOD WE ARE CREATING PARTITIONS BASED ON LIST OF VALUES.

SYNTAX:

**CREATE TABLE <TN> (<COLUMN NAME1> <DATATYPE>[SIZE],
.....)**

**PARTITION BY LIST (<KEY COLUMN NAME>) (PARTITION
<PARTITION NAME1> VALUES (VALUE1, VALUE2,),
PARTITION <PARTITION NAME2> VALUES (VALUE1, VALUE2,
.....), , PARTITION OTHERS VALUES(DEFAULT));**

EX:

CREATE TABLE TEST2(SNO INT, CNAME VARCHAR2(10))

**PARTITION BY LIST(CNAME) (PARTITION P1
VALUES('ORACLE','MYSQL'),**

**PARTITION P2 VALUES('JAVA','PHP'), PARTITION OTHERS
VALUES(DEFAULT));**

TESTING:

```
SQL> INSERT INTO TEST2 VALUES(1,'ORACLE');
SQL> INSERT INTO TEST2 VALUES(2,'C');
```

.....

CALLING A PARTICULAR PARTITION:

```
SQL> SELECT * FROM TEST2 PARTITION(P1);
```

SNO	CNAME
-----	-----
1	ORACLE

3) HASH PARTITION:

- IN THIS METHOD PARTITIONS ARE CREATED BY THE SYSTEM BY DEFAULT.

SYNTAX:

```
CREATE TABLE <TN> (<COLUMN NAME1> <DATATYPE>[SIZE],
.....) PARTITION BY HASH (<KEY COLUMN NAME>) PARTITIONS <NUMBER>;
```

EX:

```
SQL> CREATE TABLE TEST3(SNO INT, SAL NUMBER (10))
PARTITION BY HASH(SAL) PARTITIONS 5;
```

NOTE: IF WE WANT TO VIEW ALL PARTITIONS INFORMATION IN ORACLE DATABASE THEN WE USE "USER_TAB_PARTITIONS" DATA DICTIONARY.

EX:

```
SQL> DESC USER_TAB_PARTITIONS;
SQL> SELECT PARTITION_NAME FROM USER_TAB_PARTITICNS
WHERE TABLE_NAME='TEST3';
```

ADDING A NEW PARTITION:

SYNTAX:

```
ALTER TABLE <TN> ADD PARTITION <PARTITION NAME> VALUES  
LESS THAN(VALUE);
```

EX:

```
SQL> ALTER TABLE TEST1 ADD PARTITION P4 VALUES LESS THAN  
(4000);
```

DROPPING A PARTITION:

SYNTAX:

```
ALTER TABLE <TN> DROP PARTITION <PARTITION NAME>;
```

EX:

```
SQL> ALTER TABLE TEST1 DROP PARTITION P1;
```

**NOTE: IF WE WANT TO KNOW WHETHER TABLE IS PARTITIONED
OR NOT THEN WE USE "USER_TABLES" DATA DICTIONARY.**

EX:

```
SQL> DESC USER_TABLES;
```

```
SQL> SELECT PARTITIONED FROM USER_TABLES WHERE  
TABLE_NAME='EMP';
```

SEQUENCE:

- SEQUENCE IS A DB OBJECT WHICH IS USED TO GENERATE SEQUENCE NUMBERS ON A PARTICULAR COLUMN AUTOMATICALLY.

SYNTAX:

```
CREATE SEQUENCE <SEQUENCE NAME>
[ START WITH N]
[ MINVALUE N]
[ INCREMENT BY N]
[ MAXVALUE N]
[ NO CYCLE / CYCLE]
[ NO CACHE / CACHE N];
```

PARAMETERS OF SEQUENCE OBJECT:

START WITH N:

- IT REPRESENT THE STARTING SEQUENCE NUMBER. HERE "N" IS REPRESENT WITN NUMBER.

MINVALUE N:

- IT SPECIFY THE MINIMUM VALUE OF THE SEQUENCE. HERE "N" IS REPRESENT WITN NUMBER.

INCREMENT BY N:

- IT SPECIFY THE INCREMENTAL VALUE IN BETWEEN SEQUENCE NUMBERS. HERE "N" IS REPRESENT WITN NUMBER.

MAXVALUE N:

- IT SPECIFY THE MAXIMUM VALUE OF THE SEQUENCE. HERE "N" IS REPRESENT WITN NUMBER.

NO CYCLE:

- IT IS DEFAULT PARAMETER.IF WE CREATED SEQUENCE WITH "NO CYCLE" THEN SEQUENCE STARTS FROM START WITH VALUE AND GENERATE VALUES UPTO MAX VALUE.AFTER REACHING MAX VALUE THEN SEQUENCE IS STOP.

CYCLE:

- IF WE CREATED A SEQUENCE WITH "CYCLE" THEN SEQUENCE STARTS FROM START WITH VALUE AND GENERATE VALUES UPTO MAXVALUE.AFTER REACHING MAX VALUE THEN SEQUENCE WILL STARTS WITH MINVALUE.

NO CACHE:

- IT IS DEFAULT PARAMETER.WHEN WE CREATED A SEQUENCE WITH "NO CACHE"

PARAMETER THEN THE SET OF SEQUENCE VALUES ARE STORING INTO DATABASE

MEMORY.EVERY TIME WE WANT ACCESS SEQUENCE NUMBERS THEN ORACLE SERVER WILL GO TO DATABASE MEMORY AND RETURN TO USER.SO THAT IT WILL DEGRADE THE PERFORMANCE OF AN APPLICATION.

CACHE N:

- WHEN WE CREATED A SEQUENCE WITH "CACHE" PARAMETER THEN SYSTEM IS ALLOCATING TEMP. MEMORY(CACHE) AND IN THIS MEMORY WE WILL STORE THE SET SEQUENCE NUMBERS.WHENEVER USER WANT TO ACCESS SEQUENCE NUMBERS THEN ORACLE SERVER WILL GO TO CACHE MEMORY AND RETURN TO USER.

- ACCESSING DATA FROM CACHE IS MUCH FASTER THAN ACCESSING DATA FROM DATABASE.IT WILL INCREASE THE PERFORMANCE OF AN APPLICATION.HERE "N" IS REPRESENT THE SIZE OF CACHE FILE.MINIMUM SIZE OF CACHE IS 2KB AND MAXIMUM SIZE OF CACHE IS 20KB.

NOTE:

- TO WORK WITH SEQUENCE OBJECT WE SHOULD USE THE FOLLOWING TWO PSEUDO COLUMNS ARE "NEXTVAL" AND "CURRVAL".

NEXTVAL:

- IT IS USED TO GENERATE SEQUENCE NUMBERS ON A PARTICULAR COLUMN.

SYNTAX:

SELECT <SEQUENCE NAME>. <NEXTVAL> FROM DUAL;

CURRVAL:

- IT IS USED TO SHOW THE CURRENT VALUE OF THE SEQUENCE.

SYNTAX:

SELECT <SEQUENCE NAME>. <CURRVAL> FROM DUAL;

EX1:

STEP1:

SQL> CREATE SEQUENCE SQ1

START WITH 1

MINVALUE 1

INCREMENT BY 1

MAXVALUE 3;

SEQUENCE CREATED.

STEP2:

SQL> CREATE TABLE TEST1(SNO INT, NAME VARCHAR2(10));

TABLE CREATED.

TESTING:

=====

SQL> INSERT INTO TEST1 VALUES (SQ1.NEXTVAL,'&NAME');

ENTER VALUE FOR NAME: A

/

ENTER VALUE FOR NAME: B

/

ENTER VALUE FOR NAME: C

/

ENTER VALUE FOR NAME: D

ERROR AT LINE 1:

**ORA-08004: SEQUENCE SQ1.NEXTVAL EXCEEDS MAXVALUE AND
CANNOT BE INSTANTIATED.**

ALTERING A SEQUENCE:

SYNTAX:

ALTER SEQUENCE <SEQUENCE NAME> <PARAMETER NAME> N;

EX:

SQL> ALTER SEQUENCE SQ1 MAXVALUE 5;

SEQUENCE ALTERED.

TESTING:

SQL> INSERT INTO TEST1 VALUES (SQ1.NEXTVAL,'&NAME');

ENTER VALUE FOR NAME: D

/

ENTER VALUE FOR NAME: E

OUTPUT:

SQL> SELECT * FROM TEST1;

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

NOTE: WE CAN ALTER ALL PARAMETERS EXCEPT "START WITH" PARAMETER.

EX2:

SQL> CREATE SEQUENCE SQ2

START WITH 1

MINVALUE 1

INCREMENT BY 1

MAXVALUE 3

CYCLE

CACHE 2;

SEQUENCE CREATED.

SQL> CREATE TABLE TEST2(SNO INT, NAME VARCHAR2(10));

TABLE CREATED.

TESTING:

SQL> INSERT INTO TEST2 VALUES (SQ2.NEXTVAL,'&NAME');

ENTER VALUE FOR NAME: A

/

ENTER VALUE FOR NAME: B

/

ENTER VALUE FOR NAME: C

/

.....
.....

OUTPUT:

SQL> SELECT * FROM TEST2;

SNO	NAME
1	A
2	B
3	C
1	D
2	E
3	F

EX3:

SQL> CREATE SEQUENCE SQ3

START WITH 3

MINVALUE 1

INCREMENT BY 1

MAXVALUE 5

CYCLE

CACHE 2;

SEQUENCE CREATED.

SQL> CREATE TABLE TEST3(SNO INT, NAME VARCHAR2(10));

TABLE CREATED.

TESTING:

SQL> INSERT INTO TEST3 VALUES (SQ3.NEXTVAL,'&NAME');

ENTER VALUE FOR NAME: A

/

.....
.....
OUTPUT:

SQL> SELECT * FROM TEST3;

SNO	NAME
3	A
4	B
5	C
1	M
2	N
3	O
4	P
5	Q

**NOTE: IF WE WANT TO VIEW ALL SEQUENCES IN ORACLE DATABASE THEN WE USE
"USER_SEQUENCES" DATA DICTIONARY.**

EX:

SQL> DESC USER_SEQUENCES;

SQL> SELECT SEQUENCE_NAME FROM USER_SEQUENCES;

SYNTAX TO DROP A SEQUENCE:

=====

SQL> DROP SEQUENCE <SEQUENCE NAME>;

EX:

SQL> DROP SEQUENCE SQ1;

LOCKS: IT IS A MECHANISM WHICH IS USED TO PREVENT UNAUTHORIZED ACCESS FOR OUR RESOURCE. ALL DATABASE SYSTEMS ARE HAVING TWO TYPES OF LOCKS. THOSE ARE,

- i) ROW LEVEL LOCKS
- ii) TABLE LEVEL LOCKS

ROW LEVEL LOCKS: IN ROW LEVEL LOCKING WE ARE LOCKING A ROW / SET OF ROWS FROM THE TABLE. IN ALL DATABASES WHENEVER WE ARE USING COMMIT / ROLLBACK COMMAND THEN ONLY LOCKS ARE RELEASED.

EX. ON ROW LEVEL LOCKING ON A SINGLE ROW:

USER - 1:

SQL> CONN SYSTEM / MANAGER;

SQL> UPDATE SCOTT.EMP SET SAL=2000 WHERE EMPNO=7788;

1ROW UPDATED.

SQL> COMMIT; [FOR RELEASING LOCKS]

USER - 2:

SQL> CONN SCOTT/TIGER;

SQL> UPDATE EMP SET SAL=3000 WHERE EMPNO=7788;

[WE CANNOT PERFORM UPDATE OPERATION BCZ THIS ROW IS LOCKED BY THE USER SYSTEM]

EX. ON ROW LEVEL LOCKING ON SET OF ROWS:

WHEN WE ARE LOCKING SET OF ROWS FROM TABLE THEN USE "FOR UPDATE" CLAUSE IN SELECT QUERY.

USER - 1:

SQL> CONN SYSTEM / MANAGER;

SQL> SELECT * FROM SCOTT.EMP WHERE DEPTNO=10 FOR UPDATE;

SQL> COMMIT; [FOR RELEASING LOCKS]

USER - 2:

```
SQL> CONN SCOTT/TIGER;  
SQL> UPDATE EMP SET SAL=3500 WHERE DEPTNO=10;  
[WE CANNOT PERFORM UPDATE]
```

DEAD LOCK: IN ORACLE DEAD LOCKS OCCURS WHENEVER TWO /
MORE THAN TWO SESSIONS WAITING FOR DATA IF THOSE
SESSIONS ARE ALREADY LOCKED TO EACH OTHER.

THESE DEAD LOCKS ALSO RELEASED WHEN WE ARE USING COMMIT
/ ROLLBACK COMMAND.

EX:

USER - 1:

```
SQL> CONN SYSTEM / MANAGER;  
SQL> UPDATE SCOTT.EMP SET SAL=3000 WHERE EMPNO=7369;  
1ROW UPDATED.  
SQL> UPDATE SCOTT.EMP SET SAL=5000 WHERE EMPNO=7788;  
ERROR: DEADLOCK DEFECTED WHILE WAITING FOR RESOURCE.  
SQL> COMMIT; [FOR LOCKS RELEASE]
```

USER - 2:

```
SQL> CONN SCOTT / TIGER;  
SQL> UPDATE EMP SET SAL=4000 WHERE EMPNO=7788;  
1ROW UPDATED.  
SQL> UPDATE EMP SET SAL=6000 WHERE EMPNO=7369;
```

TABLE LEVEL LOCKING: IN THIS LEVEL WE ARE LOCKING A
TABLE (ALL ROWS). ORACLE HAVING TWO TYPES OF TABLE LEVEL
LOCKING

- i) SHARE LOCK
- ii) EXCLUSIVE LOCK

SHARE LOCK: SHARED LOCK EXISTS WHEN TWO TRANSACTIONS (USERS) ARE GRANTED READ ACCESS. ONE TRANSACTION GETS SHARED LOCK ON DATA AND WHEN THE SECOND TRANSACTION REQUESTS THE SAME DATA IT IS ALSO GIVEN A SHARED LOCK. BOTH TRANSACTIONS ARE READ-ONLY MODE. HERE AT A TIME NO. OF USERS ARE LOCKS THE RESOURCES.

UPDATING DATA NOT ALLOWED UNTIL THE SHARED LOCK IS RELEASED BY USING COMMIT/ROLLBACK.

SYNTAX:

LOCK TABLE <TN> IN SHARE MODE;

EX:

USER - 1:

SQL> CONN SYSTEM / MANAGER;

SQL> LOCK TABLE EMP IN SHARE MODE;

SQL> COMMIT; [FOR RELEASING LOCKS]

USER - 2:

SQL> CONN SCOTT /TIGER;

SQL> SELECT * FROM EMP;

SQL> LOCK TABLE EMP IN SHARE MODE;

EXCLUSIVE LOCK: EXCLUSIVE LOCK WHEN A STATEMENT MODIFIES DATA. ITS TRANSACTION HOLDS AS EXCLUSIVE LOCK ON DATA THAT PREVENTS OTHER TRANSACTION FROM ACCESSING THE DATA AND ALSO HERE AT A TIME ONLY ONE USER LOCK THE RESOURCE.

SYNTAX:

LOCK TABLE <TN> IN EXCLUSIVE MODE;

EX:

USER - 1:

SQL> CONN SYSTEM / MANAGER;

SQL> LOCK TABLE EMP IN EXCLUSIVE MODE;

SQL> COMMIT; [FOR RELEASING LOCKS]

USER - 2:

SQL> CONN SCOTT/TIGER;

SQL> SELECT * FROM EMP;

SQL> LOCK TABLE EMP IN EXCLUSIVE MODE;

(OR)

SQL>LOCK TABLE EMP IN SHARE MODE;

[CANNOT PERFORM ANY LOCKS].

INDEXES:

- INDEX IS AN DATABASE OBJECT WHICH IS USED TO RETRIEVE DATA FROM A TABLE FASTLY.
 - A DATABASE INDEX WILL WORK AS A BOOK INDEX PAGE IN TEXT BOOK. IN TEXT BOOK BY USING INDEX PAGE WE CAN RETRIEVE A PARTICULAR TOPIC FROM A TEXT BOOK VERY FASTLY SAME AS BY USING DATABASE INDEX OBJECT WE CAN RETRIEVE A PARTICULAR ROW FROM A TABLE VAERY FASTLY.
 - BY USING INDEXES, WE CAN SAVE TIME AND IMPROVE THE PERFORMANCE OF DATABASE. THESE INDEXES ARE CREATED BY DBA.
 - INDEX OBJECT CAN BE CREATED ON A PARTICULAR COLUMN (OR) COLUMNS OF A TABLE AND THESE COLUMNS ARE CALLED AS "INDEX KEY COLUMNS".
 - ALL DATABASES ARE SUPPORTING THE FOLLOWING TWO TYPES OF SEARCHING MECHANISMS THOSE ARE,
1. TABLE SCAN(DEFAULT)
 2. INDEX SCAN

1. TABLE SCAN:

- IT IS A DEFAULT SCANNING MECHANISM FOR RETRIEVING DATA FROM TABLE. IN THIS MECHANISM ORACLE SERVER IS SCANNING ENTIRE TABLE (TOP - BOTTOM)
- EX:

SQL> SELECT * FROM EMP WHERE SAL=3000;

SOL:

SAL

800

1600

1250

2975

1250

2850

2450

3000 (IN THIS TABLE SCAN WE ARE COMPARING WHERE CONDITION 14 TIMES)

5000

1500

1100

950

3000

1300

2) INDEX SCAN:

- IN INDEX SCAN MECHANISM ORACLE SERVER SCANNING ONLY INDEXED COLUMN FROM A TABLE. IN THIS MECHANISM WE AGAIN FOLLOW THE FOLLOWING TWO METHODS,

I) AUTOMATICALLY / IMPLICITLY:

- WHENEVER WE ARE CREATING A TABLE ALONG WITH "PRIMARY KEY" (OR) "UNIQUE" KEY CONSTRAINT THEN INTERNALLY SYSTEM IS CREATING AN INDEX OBJECT ON THAT PARTICULAR COLUMN AUTOMATICALLY.

EX:

```
SQL> CREATE TABLE TEST1(EID INT PRIMARY KEY, ENAME  
VARCHAR2(10));
```

```
SQL> CREATE TABLE TEST2(SNO INT UNIQUE, NAME  
VARCHAR2(10));
```

NOTE:

- IF WE WANT TO VIEW INDEX NAME ALONG WITH COLUMN NAME OF A PARTICULAR TABLE THEN WE USE "USER_IND_COLUMNS" DATA DICTIONARY.

EX:

```
SQL> DESC USER_IND_COLUMNS;
```

```
SQL> SELECT COLUMN_NAME, INDEX_NAME FROM  
USER_IND_COLUMNS WHERE TABLE_NAME='TEST1';
```

COLUMN_NAME	INDEX_NAME
EID	SYS_C005501

**SQL> SELECT COLUMN_NAME, INDEX_NAME FROM
USER_IND_COLUMNS WHERE TABLE_NAME='TEST2';**

COLUMN_NAME	INDEX_NAME
SNO	SYS_C005502

II) MANUALLY / EXPLICITLY:

- WHEN USER WANT TO CREATE AN INDEX OBJECT ON A PARTICULAR COLUMN/(S) THEN WE FOLLOW THE FOLLOWING SYNTAXS,

TYPES OF INDEXES:

1. B - TREE INDEX (DEFAULT INDEX)

- SIMPLE INDEX**
- COMPOSITE INDEX**
- UNIQUE INDEX**
- FUNCTIONAL BASED INDEX**

2. BITMAP INDEX

SIMPLE INDEX:

- WHEN WE CREATED AN INDEX ON A SINGLE COLUMN THEN WE CALLED AS SIMPLE INDEX.

SYNTAX:

CREATE INDEX <INDEX NAME> ON <TN> (<COLUMN NAME>);

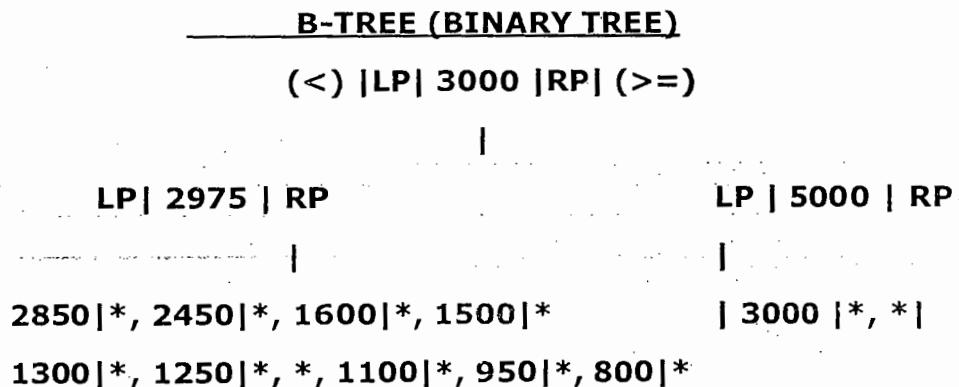
EX:

```
SQL> CREATE INDEX SIND ON EMP(SAL);  
INDEX CREATED.
```

EX:

```
SQL> SELECT * FROM EMP WHERE SAL=3000;
```

SOL:



NOTE: IN INDEX SCAN WE ARE COMPARING 3 TIMES. WHICH IS MUCH FASTER THAN TABLE SCAN (14 TIMES COMPARING). HERE " * " IS REPRESENT ROWID.

COMPOSITE INDEX:

- WHEN WE CREATED AN INDEX ON MULTIPLE COLUMNS THEN WE CALLED AS COMPOSITE INDEX.

SYNTAX:

```
CREATE INDEX <INDEX NAME> ON <TN> (<COLUMN NAME1>, <COLUMN NAME2>, .....);
```

EX:

```
SQL> CREATE INDEX CIND ON EMP (DEPTNO, JOB);  
INDEX CREATED.
```

NOTE: ORACLE SERVER USES ABOVE INDEX WHEN "SELECT" QUERY WITH WHERE CLAUSE IS BASED ON LEADING COLUMN OF INDEX,I.E (DEPTNO).

EX:

SQL> SELECT * FROM EMP WHERE DEPTNO=10;(INDEX SCAN)

**SQL> SELECT * FROM EMP WHERE DEPTNO=10 AND
JOB='CLERK';(INDEX SCAN)**

SQL> SELECT * FROM EMP WHERE JOB='CLERK';(TABLE SCAN)

UNIQUE INDEX:

- WHEN WE CREATE AN INDEX BASED ON "UNIQUE CONSTRAINT" COLUMN IS CALLED UNIQUE INDEX. UNIQUE INDEX DOES NOT ALLOW DUPLICATE VALUES.

SYNTAX:

CREATE UNIQUE INDEX <INDEX NAME> ON <TN> (<COLUMN NAME>);

EX:

SQL> CREATE UNIQUE INDEX UIND ON DEPT(DNAME);

INDEX CREATED.

TESTING:

SQL> INSERT INTO DEPT VALUES (50,'SALES','HYD')

ERROR AT LINE 1:

ORA-00001: UNIQUE CONSTRAINT (SCOTT.UIND) VIOLATED.

NOTE: PRIMARY KEY COLUMNS AND UNIQUE COLUMNS ARE AUTOMATICALLY INDEXED BY ORACLE.

FUNCTIONAL BASED INDEX:

- WHEN WE CREATE AN INDEX BASED ON FUNCTION THEN WE CALLED AS FUNCTIONAL BASED INDEX.

SYNTAX:

CREATE INDEX <INDEX NAME> ON <TN>(<FUNCTION NAME>(COLUMN NAME));

EX:

SQL> CREATE INDEX IND4 ON EMP(UPPER(ENAME));

INDEX CREATED.

**SQL> SELECT * FROM EMP WHERE
UPPER(ENAME)='SCOTT';(INDEX SCAN)**

2. BITMAP INDEX:

- BITMAP INDEX IS CREATED ON DISTINCT VALUES OF A PARTICULAR COLUMN.GENERALLY BITMAP INDEXES ARE CREATED ON LOW CARDINALITY OF COLUMNS.

- WHEN WE CREATE BITMAP INDEX INTERNALLY ORACLE SERVER IS PREPARING BITMAP INDEXED TABLE WITH BIT NUMBERS ARE 1 AND 0. HERE 1 IS REPRESENT CONDITION IS TRUE WHERE AS 0 IS REPRESENT CONDITION IS FALSE.

CARDINALITY:

- IT REFERES TO THE UNIQUENESS OF DATA VALUES CONTAINED IN PARTICULAR COLUMN OF TABLE.

HOW TO FIND CARDINALITY OF A COLUMN:

**CARDINALITY OF COLUMN = NO. OF DISTINCT VALUES OF A
COLUMN**

NO. OF ROWS IN A TABLE

EX:

CARDINALITY OF EMPNO = 14

14

CARDINALITY OF EMPNO IS "1" ----(CREATING BTREE INDEX)

EX:

CARDINALITY OF JOB = 5

14

CARDINALITY OF JOB = 0.35 ----- (CREATING BIT MAP INDEX)

SYNTAX:

CREATE BITMAP INDEX <INDEX NAME> ON <TN>(<COLUMN NAME>);

EX:

CREATE BITMAP INDEX BITIND ON EMP(JOB);

EX:

SELECT * FROM EMP WHERE JOB='MANAGER';

BITMAP INDEXED TABLE

JOB	1	2	3	4	5	6	7	8	9	10	11	12	13	14
CLERK	1	0	0	0	0	0	0	0	0	0	1	1	0	1
SALESMAN	0	1	1	0	1	0	0	0	0	1	0	0	0	0
MANAGER	0	0	0	1	0	1	1	0	0	0	0	0	0	0

ANALYST 0 0 0 0 0 0 0 1 0 0 0 0 1 0

PRESIDENT 0 0 0 0 0 0 0 0 1 0 0 0 0 0

NOTE: HERE "1" IS REPRESENTED WITH ROWID OF A PARTICULAR ROW IN A TABLE.

NOTE:

- IF WE WANT TO VIEW INDEX NAME ALONG WITH INDEX TYPE THEN WE USE "USER_INDEXES" DATADICTONARY.

EX:

SQL> DESC USER_INDEXES;

SQL> SELECT INDEX_NAME, INDEX_TYPE FROM USER_INDEXES
WHERE TABLE_NAME='EMP';

INDEX_NAME	INDEX_TYPE
SIND	NORMAL(B-TREE)
BITIND	BITMAP
FIND	FUNCTION-BASED NORMAL(B-TREE)
UIND	NORMAL(B-TREE)
CIND	NORMAL(B-TREE)

HOW TO DROP AN INDEX:

SQL> DROP INDEX <INDEX NAME>;

EX:

SQL> DROP INDEX SIND;

SQL> DROP INDEX BITIND;

CLUSTER:

- > CLUSTER IS A DB OBJECT WHICH CONTAIN GROUP OF TABLES TOGETHER AND ALSO IT SHARES SAME DATABLOCK.
- > GENERALLY, CLUSTER ARE USED TO IMPROVE PERFORMANCE OF THE JOINS AND ALSO CLUSTERS ARE CREATED BY DBA ONLY.
- > CLUSTER TABLE MUST HAVE A COMMON COLUMN NAME.THIS COMMON COLUMN IS ALSO CALLED AS CLUSTER KEY.GENERALLY CLUSTER ARE CREATED AT THE TIME OF TABLE CREATION.

STEPS TO CREATE CLUSTER IN ORACLE:

STEP1: CREATE CLUSTER:

SYNTAX:

```
CREATE CLUSTER <CLUSTER NAME> (<COMMON COLUMN NAME>  
<DT>[SIZE]);
```

EX:

```
SQL> CREATE CLUSTER EMP_DEPT (DEPTNO INT);
```

CLUSTER CREATED.

STEP2: CREATE INDEX ON CLUSTER:

SYNTAX:

```
CREATE INDEX <INDEX NAME> ON CLUSTER <CLUSTER NAME>;
```

EX:

```
SQL> CREATE INDEX ABC ON CLUSTER EMP_DEPT;
```

INDEX CREATED.

STEP3: CREATE CLUSTER TABLES:

SYNTAX:

```
CREATE TABLE  
<TN>(<COL1><DT>[SIZE],<COL2><DT>[SIZE].....)  
CLUSTER <CLUSTER NAME> (COMMON COLUMN NAME);
```

EX:

```
SQL> CREATE TABLE EMP1(EMPNO INT, ENAME VARCHAR2(20),  
DEPTNO INT) CLUSTER EMP_DEPT(DEPTNO);
```

TABLE CREATED.

```
SQL> CREATE TABLE DEPT1(DEPTNO INT, DNAME VARCHAR2(20),  
LOC VARCHAR2(20)) CLUSTER EMP_DEPT(DEPTNO);
```

TABLE CREATED.

```
SQL> DESC EMP1;
```

NAME	NULL?	TYPE
EMPNO		NUMBER (38)
ENAME		VARCHAR2(20)
DEPTNO		NUMBER (38)

```
SQL> DESC DEPT1;
```

NAME	NULL?	TYPE
DEPTNO		NUMBER (38)
DNAME		VARCHAR2(20)
LOC		VARCHAR2(20)

```
SQL> INSERT INTO EMP1 VALUES(1,'A',10);
```

```
SQL> INSERT INTO EMP1 VALUES(2,'B',20);
```

```
SQL> SELECT * FROM EMP1;
```

EMPNO	ENAME	DEPTNO
1	A	10
2	B	20

```
SQL> INSERT INTO DEPT1 VALUES (10,'X','HYD');
SQL> INSERT INTO DEPT1 VALUES (20,'Z','UP');
SQL> SELECT * FROM DEPT1;
```

DEPTNO	DNAME	LOC
10	X	HYD
20	Z	UP

NOTE: THESE TWO TABLES HAVING COMMON COLUMN(DEPTNO) AND HAVING IN THE SAME MEMORY SO THAT THEIR ROWID'S ARE SAME.

```
SQL> SELECT ROWID FROM EMP1;
```

ROWID
AAA0ZUAAEAAAAHEAAA
AAA0ZUAAEAAAAHFAAA

```
SQL> SELECT ROWID FROM DEPT1;
```

ROWID
AAA0ZUAAEAAAAHEAAA
AAA0ZUAAEAAAAHFAAA

NOTE: TO VIEW ALL CLUSTER OBJECTS IN ORACLE THEN WE FOLLOW THE FOLLOWING DATADICTONARY IS "USER_CLUSTERD".

```
SQL> DESC USER_CLUSTERS;
SQL> SELECT CLUSTER_NAME FROM USER_CLUSTERS;
```

CLUSTER_NAME

EMP_DEPT

NOTE: TO VIEW CLUSTERED TABLES IN ORACLE THEN WE USE DATADICTONARY IS "USER_TABLES".

SQL> DESC USER_TABLES;

SQL> SELECT TABLE_NAME FROM USER_TABLES WHERE CLUSTER_NAME='EMP_DEPT';

TABLE_NAME

DEPT1

EMP1

DROPPING CLUSTER WITH TABLES:

SYNTAX:

SQL> DROP CLUSTER <CLUSTER NAME>;

SQL> DROP CLUSTER EMP_DEPT;

ERROR AT LINE 1:

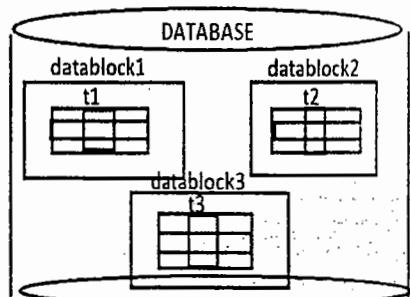
ORA-00951: CLUSTER NOT EMPTY

**> TO OVERCOME THE ABOVE ERROR, WE SHOULD USE
" INCLUDING TABLES " CLAUSE TO DROP CLUSTER ALONG WITH
TABLES.**

SQL> DROP CLUSTER EMP_DEPT INCLUDING TABLES;

CLUSTER DROPPED.

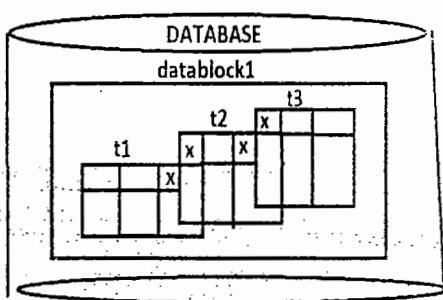
NON - CLUSTER TABLES (DEFAULT):



`select * from t1,t2,t3 where t1.cc=t2.cc and t2.cc=t3.cc;`

to degrade joins performance

CLUSTER TABLES:



`select * from t1,t2,t3 where t1.cc=t2.cc and t2.cc=t3.cc;`

to improve joins performance

USER - DEFINE DATATYPES:

USER DEFINE DATATYPES ARE INTRODUCED IN ORACLE 8.0 VERSION. WHEN PRE-DEFINE DATATYPES ARE NOT REACHING TO OUR REQUIREMENTS THEN WE CREATE OUR OWN DATATYPES ARE CALLED AS USER DEFINE DATATYPES.

THE ADVANTAGE OF USER DEFINE DATATYPES ARE REUSABILITY THAT MEANS WE CAN CREATE DATATYPE AND REUSE IN MULTIPLE TABLES. ORACLE SUPPORTS THE FOLLOWING THREE TYPES OF USERS DEFINE DATATYPES.

- 1. OBJECT TYPE (OR) COMPOSITE TYPE**
- 2. VARRAY**
- 3. NESTED TABLE.**

1. OBJECT TYPE (OR) COMPOSITE TYPE:

IT ALLOWS GROUP OF VALUES /ELEMENTS OF DIFFERENT DATATYPES.

SYNTAX:

```
CREATE TYPE <TYPE NAME> AS OBJECT(<COL1>
DATATYPE[SIZE], <COL2> DATATYPE[SIZE], .....);
```

/

EX:

```
CREATE TYPE COURSE_TYPE AS OBJECT (CID NUMBER (4), CNAME
VARCHAR2(10), FEE NUMBER (10));
```

/

TESTING:

EX:

```
CREATE TABLE STUDENTS (SID NUMBER (4), SNAME
VARCHAR2(10), COURSE COURSE_TYPE);
```

TABLE CREATED.

```
SQL> INSERT INTO STUDENTS VALUES (101,'SAI', COURSE_TYPE
(1021,'ORACLE',1200));
```

```
SQL> INSERT INTO STUDENTS VALUES (102,'WARD',
COURSE_TYPE (1022,'C',500));
```

TO SELECT:

**SQL> SELECT S.SID, S. SNAME, S.COURSE.CID, S. COURSE.CNAME,
S.COURSE.FEE FROM STUDENTS S;**

(OR)

**SQL> SELECT S.SID, S. SNAME, S.COURSE.CID CID, S.
COURSE.CNAME CNAME, S.COURSE.FEE FEE FROM STUDENTS S;**

TO UPDATE:

**SQL> UPDATE STUDENTS S SET S.COURSE.FEE=2000 WHERE
S.SID=101;**

TO DELETE:

SQL> DELETE FROM STUDENTS S WHERE S.COURSE.CID=1022;

2. VARRAY:

**IT ALLOWS GROUP OF VALUES /ELEMENTS OF SAME
DATATYPES.VARRAY SHOULD DECLARE WITH SIZE.**

SYNTAX:

**CREATE TYPE <TYPE NAME> IS VARRAY(SIZE) OF
DATATYPE[SIZE];**

/

EX:

CREATE TYPE MBNO_ARRAY1 IS VARRAY (3) OF NUMBER (10);

/

TESTING:

**SQL> CREATE TABLE EMPLOYEE (EMPNO NUMBER (4), MBNO
MBNO_ARRAY1);**

TABLE CREATED.

```
SQL> INSERT INTO EMPLOYEE VALUES (1021,  
MBNO_ARRAY1(9703542749,8502045789));  
SQL> INSERT INTO EMPLOYEE VALUES (1022,  
MBNO_ARRAY1(9632587412,8523691478,7412356896));
```

3.NESTED TABLE:

- > A TABLE WITHIN ANOTHER TABLE IS CALLED AS NESTED TABLE.
- > NESTED TABLE ALSO ALLOW GROUP OF VALUES /ELEMENTS OF DIFF. DATATYPES.
- > NESTED TABLE IS NOT DECLARE WITH SIZE.

STEPS TO CREATE NESTED TABLE:

STEP1: CREATE AN OBJECT TYPE:

SYNTAX:

```
CREATE TYPE <TYPE NAME> AS OBJECT(<COL1>  
DATATYPE[SIZE], <COL2> DATATYPE[SIZE], .....);  
/
```

STEP2: CREATE NESTED TABLE TYPE:

SYNTAX:

```
CREATE TYPE <TYPE NAME> AS TABLE OF <OBJECT TYPE NAME>;  
/
```

STEP3: CREATE A TABLE:

SYNTAX:

```
CREATE TABLE <TN>(<COL1> <DATATYPE>[SIZE], ....., <COL N>  
<NESTED TABLE TYPE NAME>)  
NESTED TABLE <COL N NAME> STORE AS <ANY NAME>;
```

EX:

**STEP1: CREATE TYPE ADDR_TYPE AS OBJECT (HNO NUMBER (4),
STREET VARCHAR2(10), CITY VARCHAR2(10));**

/

STEP2:

CREATE TYPE ADDR_ARRAY AS TABLE OF ADDR_TYPE;

/

STEP3:

**CREATE TABLE CUSTOMER (CID NUMBER (4), CNAME
VARCHAR2(10), CADDRESS ADDR_ARRAY) NESTED TABLE
CADDRESS STORE AS CUST_ADDR;**

TESTING:

**SQL> INSERT INTO CUSTOMER VALUES (1,'SAI', ADDR_ARRAY
(ADDR_TYPE (1122,'GANDHI','HYD')));**

**SQL> INSERT INTO CUSTOMER VALUES (2,'WARD', ADDR_ARRAY
(ADDR_TYPE (1123,'ASHOK','CHE'), ADDR_TYPE
(1124,'VASATI','MUM')));**

**NOTE: WE CAN ALSO SELECT, UPDATE, DELETE, INSERT DATA
WITHIN NESTED TABLE BY USING THE FOLLOWING SYNTAX,**

SYNTAX:

**SELECT / UPDATE / DELETE / INSERT (SELECT <NESTED TABLE
TYPE COLUMN NAME> FROM <TN>);**

EX:

**SQL> SELECT * FROM TABLE (SELECT CADDRESS FROM CUSTOMER
WHERE CID=1);**

**SQL> UPDATE TABLE (SELECT CADDRESS FROM CUSTOMER
WHERE CID=2) SET HNO=1024 WHERE HNO=1124;**

```
SQL> DELETE FROM TABLE (SELECT CADDRESS FROM CUSTOMER  
WHERE CID=2) WHERE CITY='MUM';
```

```
SQL> INSERT INTO TABLE (SELECT CADDRESS FROM CUSTOMER  
WHERE CID=1) VALUES (1124,'YUVIN','HYD');
```

**NOTE: IN ORACLE WE WANT TO VIEW USER TYPES THEN FOLLOW
THE FOLLOWING DATADICTONARY IS "USER_TYPES".**

EX:

```
SQL> DESC USER_TYPES;
```

```
SQL> SELECT TYPE_NAME FROM USER_TYPES;
```

SYNTAX TO DROP TYPE:

```
SQL> DROP TYPE <TYPE NAME> FORCE;
```

EX:

```
DROP TYPE MBNO_ARRAY1 FORCE;
```

NORMALIZATION: Normalization is a technique of organizing the data into multiple tables. Normalization process automatically eliminates data redundancy (repetition) and also avoiding Insertion, Update and Deletion problems.

Problems without Normalization: If a table is not properly normalized and have data redundancy then it will not only occupy extra memory space but will also make it difficult to handle insert, delete and update operations in student table.

STUDENT DETAILS

Roll no	Name	Branch	Hod	Office Number
101	SAI	CSE	Mr. X	040-53337
102	ALLEN	CSE	Mr. X	040-53337
103	JAMES	CSE	Mr. X	040-53337
104	MILLER	CSE	Mr. X	040-53337

In the table above, we have data of 4 Computer Sci. students. As we can see, data for the fields BRANCH, HOD and OFFICE NUMBER is repeated for the students who are in the same branch in the college, this is Data Redundancy.

Insertion problem:

If we have to insert data of 100 students of same branch, then the branch information will be repeated for all those 100 students. These scenarios are nothing but Insertion problem. Reason for data redundancy is two different related data stored in the same table.

Student data + Branch data

Updation problem:

If we want to change HOD name then system admin has to update all students records with new HOD name. and if by mistake we miss any record, it will lead to data inconsistency. This is Updation problem.

Ex: Mr. X leaves and Mr. Y join as a new HOD for CSE. Then the table will be like below,

STUDENT DETAILS

Roll no	Name	Branch	Hod	Office Number
101	SAI	CSE	Mr. Y	040-53337
102	ALLEN	CSE	Mr. Y	040-53337
103	JAMES	CSE	Mr. Y	040-53337
104	MILLER	CSE	Mr. Y	040-53337
105	WARNER	CSE	Mr. Y	040-53337

Deletion problem:

In our Student Details table, two different information's are kept together, Student information and Branch information. Hence, at the end of the academic year, if student records are deleted, we will also lose the branch information. This is called as Deletion problem.

HOW NORMALIZATION WILL SOLVE ALL THESE PROBLEMS:

STUDENT DETAILS

Roll no	Name	Branch	Hod	Office Number
101	SAI	CSE	Mr. Y	040-53337
102	ALLEN	CSE	Mr. Y	040-53337
103	JAMES	CSE	Mr. Y	040-53337
104	MILLER	CSE	Mr. Y	040-53337
105	WARNER	CSE	Mr. Y	040-53337

NOTE: NOW WE NEED TO DECOMPOSING A STUDENT TABLE INTO TWO TABLES LIKE BELOW,

STUDENT DETAILS

Roll no	Name	Branch (FK)
101	SAI	CSE
102	ALLEN	CSE
103	JAMES	CSE
104	MILLER	CSE
105	WARNER	CSE

BRANCH DETAILS

Branch (PK)	Hod	Office Number
CSE	Mr. Y	040-53337

NOTE: BY THE ABOVE EXAMPLE WE AVOID INSERTION, DELETION AND UPDATION PROBLEMS.

Types of Normal Forms: Normalization can be achieved in multiple ways:

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. Fourth Normal Form
6. Fifth Normal form

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).

Ex: column1 column2

A X, Y

B W, X

C Y

D Z

2. A COLUMN SHOULD CONTAIN VALUES THAT ARE SAME DATATYPE.

EX: NAME DOB

SAI 01-JAN-92

JONES 24-APR-84

18-DEC-85 MILLER

3. All the columns in a table should have unique names.

EX: NAME NAME DOB

SAI SAI 16-OCT-93

4. The order in which data is stored, does not matter.

EX: ROLLNO FIRST_NAME LAST_NAME

1 SAI KUMAR

2 JONES ROY

4 MILLER JOY

3 JAMES WARTON

EX: **STUDENT TABLE**

Roll no **Name** **Subject**

101 **SAI** **JAVA, ORACLE**

102 **JONES** **PYTHON**

103 **ALLEN** **C, C++**

The above table already satisfies 3 rules out of the 4 rules, as all our column names are unique, we have stored data in the order we wanted to and we have not inter-mixed different type of data in columns.

But out of the 3 different students in our table, 2 have opted for more than 1 subject. And we have stored the subject names in a single column. But as per the 1st Normal form each column must contain atomic value.

To avoid this problem, we have to break the values into atomic values. Here is our updated table and it now satisfies the First Normal Form.

< COMPOSITE PRIMARY KEY>

Roll no	Name	Subject
101	SAI	ORACLE
101	SAI	JAVA
102	JONES	PYTHON
103	ALLEN	C
103	ALLEN	C++

NOTE: By doing so, although a few values are getting repeated but values for the SUBJECT column are now atomic for each record/row.

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: A B C D

Here, "A" IS A KEY COLUMN → "B", "C", "D" ARE NON-KEY COLUMNS.

EX:

(PK)

STUDENT TABLE

STUDENT_ID	Name	Branch	ADDRESS
101	SAI	CSE	HYD
102	SAI	IT	MUM
103	JAMES	CSE	CHENNAI
104	MILLER	CSE	HYD

NOTE: A PRIMARY KEY COLUMN (STID) CAN BE USED TO FETCH DATA ANY COLUMN IN THE TABLE.

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: A B C D

Here, "A AND B" IS A KEY COLUMNS → "C", "D" ARE NON-KEY COLUMNS. THEN "D" DEPENDS ON "B" BUT NOT "A" COLUMN.

EX: Let's create another table for Subject, which will have SUBJECT_ID and SUBJECT_NAME fields and SUBJECT_ID will be the primary key.

<PRIMARY KEY>	SUBJECT TABLE
SUBJECT_ID	SUBJECT_NAME
1	ORACLE
2	JAVA
3	PYTHON

Now we have a student table with student information and another table Subject for storing subject information.

Let's create another table Score, to store the marks obtained by students in the respective subjects.

We will also be saving name of the teacher who teaches that subject along with marks.

(COMPOSITE PRIMARY KEY) SCORE TABLE

STUDENT_ID SUBJECT_ID MARKS TEACHER

101	1	70	ORACLE Teacher
101	2	75	JAVA Teacher
102	1	80	OACLE Teacher
103	3	68	PYTHON Teacher

In the score table we are saving the **STUDENT_ID** to know which student's marks are these and **SUBJECT_ID** to know for which subject the marks are for.

Together **STUDENT_ID + SUBJECT_ID** forms composite primary key for this table, which can be the Primary key.

NOTE:

1. IN ABOVE SCORE TABLE, "TEACHER COLUMN" IS ONLY DEPENDS ON SUBJECT_ID BUT NOT ON STUDENT_ID IS CALLED AS "PARTIAL DEPENDENCY".
2. IF THERE IS NO COMPOSITE PRIMARY KEY ON A TABLE THEN THERE IS NO PARTIAL DEPENDENCY.

HOW TO REMOVE PARTIAL DEPENDENCY: there are many different solutions to remove partial dependency. so our objective is to remove "teacher" column from score table and add to subject table. hence, the subject table will become

SUBJECT TABLE

SUBJECT_ID	SUBJECT_NAME	TEACHER
1	ORACLE	ORACLE Teacher
2	JAVA	JAVA Teacher
3	PYTHON	PYTHON Teacher

And our Score table is now in the second normal form, with no partial dependency.

<COMPOSITE PRIMARY KEY>

STUDENT_ID	SUBJECT_ID	Marks
101	1	70
101	2	75
102	1	80
103	3	68

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 NON-KEY COLUMN, THEN IT IS CALLED AS TRANSITIVE DEPENDENCY.

(Composite Primary key)

EX: A B C D

Here, "A AND B" ARE KEY COLUMNS → "C," "D" ARE NON-KEY COLUMNS. THEN "D" DEPENDS ON "C" BUT NOT "A & B" COLUMNS.

NOTE: In the Score table, we need to store some more information, which is the exam name and total marks, so let's add 2 more columns to the Score table.

<COMPOSITE PRIMARY KEY> SCORE TABLE

STUDENT_ID SUBJECT_ID MARKS EXAM_NAME TOTAL_MARKS

with exam name and total marks added to our score table, it saves more data now. primary key for our score table is a composite key, which means it's made up of two attributes or columns → student +subject

our new column exam-name depends on both student and subject. for example, a mechanical engineering student will have workshop exam but a computer science student won't. and for some subjects you have practical exams and for some you don't. so, we can say that exam name is dependent on both student id and subject id.

well, the column total marks depend on exam name as with exam type the total score changes. for example, practical is less marks while theory exams are having more marks.

but exam name is just another column in the score table. it is not a primary key and total marks depends on it.

this is transitive dependency. when a non-prime attribute depends on other non-prime attributes rather than depending upon the prime attributes or primary key.

How to remove Transitive Dependency: again, the solution is very simple. take out the column's exam name and total marks from score table and put them in an exam table and use the exam_id wherever required.

Score Table: In 3rd Normal Form

STUDENT_ID	SUBJECT_ID	MARKS	EXAM_ID(FK)
------------	------------	-------	-------------

Exam table

EXAM_ID(PK)	EXAM_NAME	TOTAL_MARKS
1	Workshop	200
2	Mains	70
3	Practical's	30

SUPER KEY & CANDIDATE KEY:

SUPER KEY: A COLUMN (OR) COMBINATION 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 IS NOT A CANDIDATE KEY.

IN DB DESIGN ONLY DB DESIGNER USES SUPER KEY AND CANDIDATE KEY. THAT MEAN FIRST DESIGNERS SELECT SUPER KEYS AND THEN ONLY THEY ARE SELETING CANDIDATE KEYS FROM THOSE SUPER KEYS.

EX: **STUDENT TABLE**

	STUDENT_ID	NAME	BRANCH	MAILID	REG_NUMBER
101	SAI	CSE	sai@gamil.com	CS-10021	
102	JONES	CSE	joy@gmail.com	CS-10022	
103	ALLEN	IT	all@ymail.com	IT-20021	
104	SAI	EEE	mi@hotmail.com	EE-30021	

EX. OF SUPER KEYS:

stid | stid + mailid |
mailid | mailid + reg_number | stid + mailid + reg_number
reg_number | reg_number + stid |

EX. ON CANDIDATE KEYS:

stid

mailid

reg_number

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 $A \rightarrow B$, A should be a super key.

EX:

(COMPOSITE PRIMARY KEY) College Enrollment Table

STUDENT_ID	SUBJECT(B)	PROFESSOR(A)
101	Java	P. Java
101	C++	P. Cpp
102	Java	P. Java2
103	Oracle	P. Oracle
104	Java	P. Java

in the table above, student id, subject form primary key, which means subject column, is a prime attribute. but there is one more dependency, professor → subject. and while subject is a prime attribute, professor is a non-prime attribute, which is not allowed by bcnf.

How to satisfy BCNF?

to make this relation (table) satisfy bcnf, we will decompose this table into two tables, student table and professor table.

below we have the structure for both the tables.

Student Table

STUDENT_ID	PROFESSOR_ID
101	1
101	2

Professor Table
(COMPOSITE PRIMARY KEY)

PROFESSOR_ID	professor	Subject
1	P. Java	Java
2	P. Cpp	C++

And now, this relation satisfies Boyce-Codd Normal Form.

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.

NOTE: Generally, when a table having more than one independent multi valued attributes then the table having more duplicate data for reducing this duplicate data then DB DESIGNERS use 4NF process otherwise no need (it is optional).

Ex: COLLEGE ENROLLMENT TABLE (5NF)

STUDENT_ID	COURSE	HOBBY
1	ORACLE	Cricket
1	JAVA	Reading
1	C#	Hockey

in the table above, there is no relationship between the columns course and hobby. they are independent of each other.so there is multi-value dependency, which leads to un-necessary repetition of data.

identify independent multi valued attributes and those attributes move into separate tables these tables are called as 4nf tables. these tables do not contain more than one independent multi valued attribute (column).

Hobbies Table (4NF)

STUDENT_ID	Hobby
1	Cricket
1	Reading
1	Hockey

Course Opted Table (4NF)

STUDENT_ID	Course
1	ORACLE
1	JAVA
1	C#

Fifth Normal Form (5NF):

If a table having multi valued attributes and also that table cannot decomposed into multiple tables is called as fifth normal form.

Generally, in 4NF resource table some attributes are not logically related where as in 5NF resource table all attributes are related to one to another.

Fifth normal form is also called as project joined normal form because if possible decomposing table into number of tables and also whenever we are joining those tables then the result records must be available in resource table.

PL/SOL

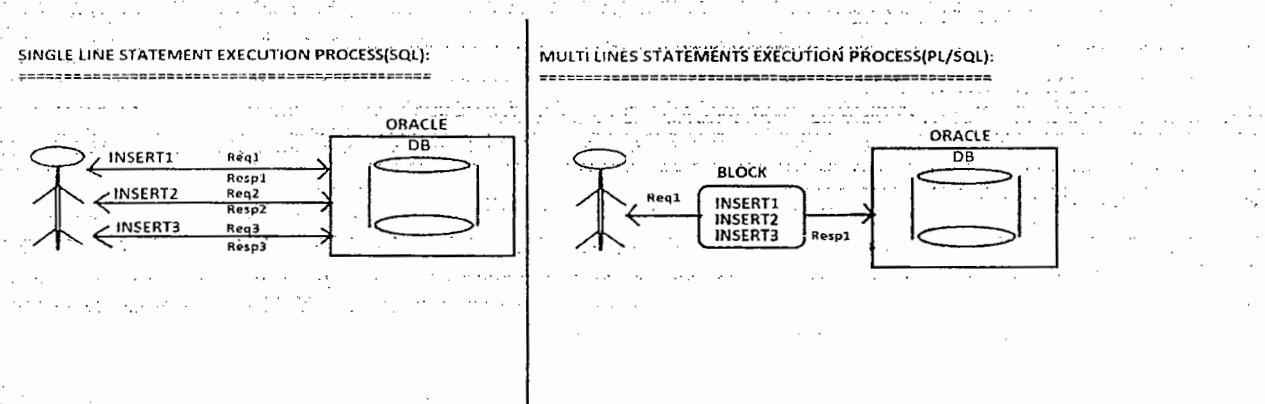
INTRODUCTION TO PL/SQL:

PL/SQL STANDS FOR PROCEDURAL LANGUAGE WHICH IS AN EXTENSION OF SQL.PL/SQL WAS INTRODUCED IN ORACLE 6.0 VERSION.

SQL IS A NON-PROCEDURAL LANGUAGE WHEREAS PL/SQL IS A PROCEDURAL LANGUAGE.

SQL SUPPORTS A SINGLE LINE STATEMENT (QUERY) EXECUTION PROCESS WHEREAS PL/SQL SUPPORTS MULTI LINES STATEMENTS (PROGRAM) EXECUTION PROCESS.

IN SQL EVERY QUERY STATEMENT IS COMPILED AND EXECUTING INDIVIDUALLY.SO THAT NO. OF COMPILEMENTS ARE INCREASED AND REDUCE PERFORMANCE OF DATABASE.



IN PL/SQL ALL SQL QUERIES ARE GROUPED INTO A SINGLE BLOCK AND WHICH WILL COMPILE AND EXECUTE ONLY ONE TIME.SO THAT IT WILL REDUCE NO. OF COMPILEMENTS AND IMPROVE PERFORMANCE OF DATABASE.

FEATURES OF PL/SOL:

1. TO IMPROVES PERFORMANCE.
2. SUPPORTING CONDITIONAL & LOOPING STATEMENTS.
3. SUPPORTING REUSABILITY.
4. PROVIDING SECURITY BECAUSE ALL PROGRAMS ARE SAVED IN DATABASE AND AUTHORIZED USER CAN ONLY ACCESS THE PROGRAMS.

5. SUPPORTING PORTABILITY I.E PL/SQL PROGRAMS CAN BE MOVED FROM ONE

PLATFORM TO ANOTHER PLATFORM WITHOUT ANY CHANGES.

6. SUPPORTING EXCEPTION HANDLING MECHANISM.

7. SUPPORTING MODULAR PROGRAMMING I.E IN A PL/SQL A BIG PROGRAM CAN BE DIVIDED INTO SMALL MODULES WHICH ARE CALLED AS STORED PROCEDURE AND

STORED FUNCTIONS.

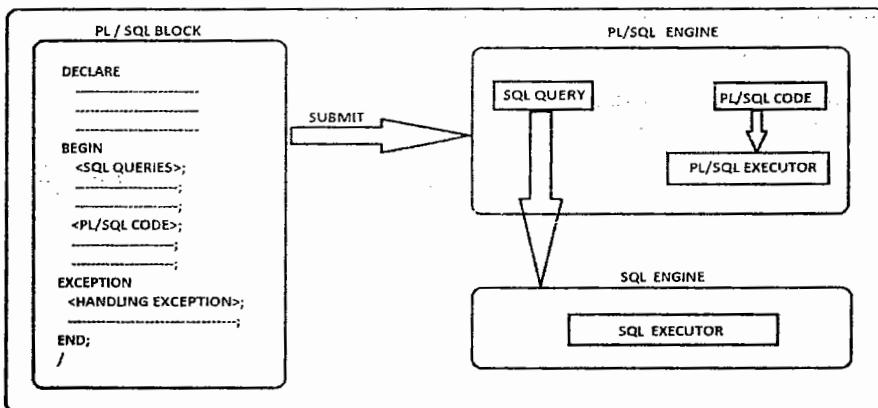
PL/SQL ARCHITECTURE:

PL/SQL IS BLOCK STRUCTURE PROGRAMMING LANGUAGE WHICH IS HAVING THE FOLLOWING TWO ENGINES THOSE ARE

1. SQL ENGINE

2. PL/SQL ENGINE

PL/SQL ARCHITECTURE :



WHENEVER WE ARE SUBMITTING A PL/SQL BLOCK INTO ORACLE SERVER THEN ALL SQL STATEMENTS(QUERIES) ARE SEPERATED AND EXECUTING BY SQLQUERY EXECUTOR WITH IN SQL ENGINE.WHERE AS ALL PL/SQL STATEMENTS(CODE) ARE SEPERATED AND EXECUTING BY PL/SQL CODE EXECUTOR WITH IN PL/SQL ENGINE.

WHAT IS BLOCK:

A BLOCK IS A SET OF STATEMENTS WHICH ARE COMPILE & EXECUTED BY ORACLE AS A SINGLE UNIT. PL/SQL SUPPORTING THE FOLLOWING TWO TYPES OF BLOCKS THOSE ARE,

1. ANONYMOUS BLOCK

2. SUB BLOCK

DIFF. B/W ANONYMOUS & SUB BLOCK:

<u>ANONYMOUS BLOCK</u>	<u>SUB BLOCK</u>
1. UNNAMED BLOCK	1. NAMED BLOCK
2. THIS BLOCK CODE IS NOT SAVED IN DB.	2. THIS BLOCK CODE IS SAVED IN DB AUTOMATICALLY.
3. IT CANNOT REUSABLE.	3. IT CAN BE REUSABLE.
4. EVERY TIME COMPILED OF CODE.	4. PRE - COMPILED CODE (FIRST TIME COMPILED ONLY)
5. ARE USING IN "DB TESTING".	5. ARE USING IN APPLICATION DEVELOPMENT LIKE "JAVA", .NET" & "DB APPLICATIONS".

ANONYMOUS BLOCKS:

THESE ARE UNNAMED BLOCKS IN PL/SQL WHICH CONTAINS THREE MORE BLOCKS THOSE ARE,

- I) DECLARATION BLOCK
- II) EXECUTION BLOCK
- III) EXCEPTION BLOCK

I) DECLARATION BLOCK:

- > THIS BLOCK STARTS WITH " DECLARE " STATEMENT.
- > DECLARING VARIABLES, CURSORS, USER DEFINE EXCEPTIONS.
- > IT IS OPTIONAL BLOCK.

II) EXECUTION BLOCK:

- > THIS BLOCK STARTS WITH " BEGIN " STATEMENT & ENDS WITH "END" STATEMENT.
- > IMPLEMENTING SQL STATEMENTS(SQL) & LOGICAL CODE OF A PROGRAM (PL/SQL).
- > IT IS MANDATORY BLOCK.

III) EXCEPTION BLOCK:

- > THIS BLOCK STARTS WITH "EXCEPTION" STATEMENT.
- > HANDLING EXCEPTIONS.
- > IT IS An OPTIONAL BLOCK.

STRUCTURE OF PL/SQL BLOCK:

```
DECLARE
    < VARIABLES, CURSOR, UD EXCEPTIONS>;
BEGIN
    < WRITING SQL STATEMENTS>;
    < PL/SQL LOGICAL CODE>;
EXCEPTION
    < HANDLING EXCEPTIONS>;
END;
/
```

VARIABLES IN PL/SQL:

STEP1: DECLARING VARIABLES:

SYNTAX:

```
DECLARE
    <VARIABLE NAME> <DT>[SIZE];
```

EX:

```
DECLARE
    A NUMBER (10) (OR) A INT;
    B VARCHAR2(10);
```

STEP2: ASSIGNING / STORING A VALUE INTO VARIABLE:

SYNTAX:

```
<VARIABLE NAME>:= <VALUE>;
```

EX:

A: = 1021;

B: = 'SAI';

HERE,

: = - ASSIGNMENT OPERATOR IN PL/SQL

= - COMPARISION OPERATOS IN PL/SQL

STEP3: PRINTING VARIABLES VALUES:

SYNTAX:

**DBMS_OUTPUT.PUT_LINE (<VARIABLE NAME> (OR) '<UD
MESSAGE>');**

EX:

DBMS_OUTPUT.PUT_LINE(A);

DBMS_OUTPUT.PUT_LINE(B);

DBMS_OUTPUT.PUT_LINE ('WELCOME TO PL/SQL');

EX1:

TO PRINT "WELCOME TO PL/SQL" STATEMENT.

SOL:

SQL> BEGIN

DBMS_OUTPUT.PUT_LINE ('WELCOME TO PL/SQL');

END;

/

PL/SQL PROCEDURE SUCCESSFULLY COMPLETED.

NOTE:

THE ABOVE PROGRAM WILL NOT DISPLAY THE OUTPUT OF A PL/SQL PROGRAM. IF ORACLE SERVER WANT TO DISPLAY OUTPUT OF A PL/SQL PROGRAM THEN WE USE THE FOLLOWING SYNTAX,

SYNTAX:

SET SERVEROUTPUT OFF / ON;

HERE,

**OFF: IT IS DEFAULT. OUTPUT IS NOT DISPLAY
ON: OUTPUT IS DISPLAY**

**SQL> SET SERVEROUTPUT ON;
SQL> /
WELCOME TO PL/SQL**

EX2:

TO PRINT VARIABLES VALUES?

SOL:

**SQL> DECLARE
X NUMBER (10);
Y NUMBER (10);
BEGIN
X: =100;
Y: =200;
DBMS_OUTPUT.PUT_LINE ('VARIABLES VALUES ARE:'||X||','||Y||);
END;
/
VARIABLES VALUES ARE:100,200**

EX3:

TO PRINT SUM OF TWO NUMBERS AT RUNTIME?

SOL:

DECLARE

```
X NUMBER (2);
Y NUMBER (2);
Z NUMBER (10);
```

BEGIN

```
X: =&X;
Y: =&Y;
Z: =X+Y;
DBMS_OUTPUT.PUT_LINE(Z);
```

END;

/

OUTPUT:

ENTER VALUE FOR X: 10

OLD 6: X: =&X;

NEW 6: X: =10;

ENTER VALUE FOR Y: 20

OLD 7: Y: =&Y;

NEW 7: Y: =20;

30

VERIFY:

ON = DISPLAY OLD, NEW BIND VARIABLE STATEMENTS

**OFF = DOESNOT DISPLAY OLD, NEW BIND VARIABLES
STATEEMTNDS**

SYNTAX:

SET VERIFY ON / OFF

EX:

SQL> SET VERIFY OFF;

SQL> /

ENTER VALUE FOR X: 10

ENTER VALUE FOR Y: 20

30

SELECT..... INTO STATEMENT:

**STORING A TABLE COLUMNS VALUES INTO VARIABELS.
RETURNS A SINGLE ROW (OR) A SINGLE VALUE.CAN USE IN
EXECUTION BLOCK.**

SYNTAX:

**SELECT <COLUMN NAME1>, <COLUMN NAME2>,INTO
<VARIABLE NAME1>, <VARIABLE NAME2>..... FROM <TN>
[WHERE <CONDITION>];**

EX1:

WA PL/SQL PRG. TO DISPLAY ENAME, SALARY DETAILS FROM EMP TABLE AS PER THE GIVEN EMPNO BY USING SELECTINTO STATEMENT?

SOL:

DECLARE

V_ENAME VARCHAR2(10);

V_SAL NUMBER (10);

BEGIN

**SELECT ENAME, SAL INTO V_ENAME, V_SAL FROM EMP WHERE
EMPNO=&EMPNO;**

DBMS_OUTPUT.PUT_LINE(V_ENAME||','||V_SAL);

END;

/

OUTPUT:

ENTER VALUE FOR EMPNO: 7788

SCOTT,3000

EX:

**WA PL/SQL PRG. TO FETCH MAX.SALARY OF EMP TABLE BY USING
"SELECT INTO" STATEMENT?**

SOL:

DECLARE

V_MAXSAL NUMBER (10);

BEGIN

SELECT MAX(SAL) INTO V_MAXSAL FROM EMP;

DBMS_OUTPUT.PUT_LINE(V_MAXSAL);

END;

/

OUTPUT:

5000

VARIABLES ATTRIBUTES (OR) ANCHOR NOTATIONS:

VARIABLES ATTRIBUTES ARE USED IN PLACE OF DATATYPES AT VARIABLE DECLARATION.

WHENEVER WE ARE USING VARIABLES ATTRIBUTES INTERNALLY ORACLE SERVER IS ALLOCATE SOME MEMORY FOR THESE VARIABLES ATTRIBUTES FOR STORING THE CORRESPONDING VARIABLE COLUMN DATATYPE WHICH WAS ASSIGNED AT THE TIME OF TABLE CREATION.

VARIABLES ATTRIBUTES ARE ALSO CALLED AS "ANCHOR NOTATIONS".

THE ADVANTAGE OF VARIABLES ATTRIBUTES ARE WHENEVER WE WANT TO CHANGE

A PARTICULAR COLUMN DATATYPE IN A TABLE THEN THE CORRESPONDING COLUMN VARIABLE DATATYPE ALSO CHANGED IN VARIABLE ATTRIBUTE MEMORY AUTOMATICALLY.

PL/SQL SUPPORTS THE FOLLOWING TWO TYPE VARIABLES ATTRIBUTES ARE,

- 1. COLUMN LEVEL ATTRIBUTES**
- 2. ROW LEVEL ATTRIBUTES**

1. COLUMN LEVEL ATTRIBUTES:

IN THIS LEVEL WE ARE DEFINING VARIABLES ATTRIBUTES FOR INDIVIDUAL COLUMNS. IT IS REPRESENTING WITH "%TYPE" STATEMENT.

SYNTAX:

<VARIABLE NAME> <TN>. <COLUMN NAME>%TYPE;

EX:

```
V_ENAME  EMP.ENAME%TYPE;  
V_SAL    EMP.SAL%TYPE;
```

PROGRAM1:

```
DECLARE
V_ENAME EMP.ENAME%TYPE;
V_SAL EMP.SAL%TYPE;
BEGIN
SELECT ENAME, SAL INTO V_ENAME, V_SAL FROM EMP WHERE
EMPNO=&EMPNO;
DBMS_OUTPUT.PUT_LINE(V_ENAME||' '||V_SAL);
END;
```

OUTPUT:

```
ENTER VALUE FOR EMPNO: 7788
SCOTT,3000
```

2. ROW LEVEL ATTRIBUTES:

**IN THIS LEVEL WE ARE DECLARING A SINGLE VARIABLE
WILL REPRESENT ALL DIFFERENT DATATYPES OF COLUMNS IN A
TABLE.IT REPRESENT WITH "%ROWTYPE ".**

SYNTAX:

```
<VARIABLE NAME> <TABLE NAME>%ROWTYPE;
```

```
EX: I EMP%ROWTYPE;
```

PROGRAM2:

```
DECLARE
I EMP%ROWTYPE;
BEGIN
SELECT ENAME, SAL INTO I.ENAME, I.SAL FROM EMP WHERE
EMPNO=&EMPNO;
DBMS_OUTPUT.PUT_LINE (I.ENAME||','||I.SAL);
END;
/
```

(OR)

```
DECLARE
I EMP%ROWTYPE;
BEGIN
SELECT * INTO I FROM EMP WHERE EMPNO=&EMPNO;
DBMS_OUTPUT.PUT_LINE (I.ENAME||','||I.SAL||','||I.DEPTNO);
END;
/
```

CONTROL STRUCTURES:

- USED TO CONTROL FLOW OF THE PROGRAM.
- THERE ARE THREE TYPES OF CONTROL STRUCTURES.

I. CONDITIONAL CONTROL STRUCTURES

II. BRANCHING CONTROL STRUCTURES

III. ITERATION CONTROL STRUCTURES

I. CONDITIONAL CONTROL STRUCTURES:

I. SIMPLE IF: IT CONTAINS ONLY TRUE BLOCK.

SYNTAX:

```
IF <CONDITION> THEN  
    <EXEC-STATEMENTS>; -- TRUE BLOCK
```

```
END IF;
```

II. IF. ELSE: IT CONTAINS BOTH TRUE BLOCK & FALSE BLOCK.

SYNTAX:

```
IF <CONDITION> THEN  
    <EXEC-STATEMENTS>; -- TRUE BLOCK  
ELSE  
    <EXEC-STATEMENTS>; -- FALSE BLOCK  
END IF;
```

III. NESTED IF:

-> IF WITHIN THE IF IS CALLED AS NESTED IF.

SYNTAX:

```
IF <CONDITION> THEN  
    IF <CONDITION> THEN  
        <EXEC-STATEMENT>;  
    ELSE  
        <EXEC-STATEMENTS>;  
    END IF;  
ELSE
```

```
IF <CONDITION> THEN
    <EXEC-STATEMENT>;
ELSE
    <EXEC-STATEMENTS>;
END IF;
END IF;
```

IV. IF..ELSE LADER:

SYNTAX:

```
IF <CONDITION> THEN
    <EXEC-STATEMENTS>;
ELSIF <CONDITION> THEN
    <EXEC-STATEMENTS>;
ELSIF <CONDITION> THEN
    <EXEC-STATEMENTS>;
.....
ELSE
    <EXEC-STATEMENTS>;
END IF;
```

II. BRANCHING CONTROL STURCTURES:

I. CASE:

SYNTAX:

```
CASE <VARIABLE/EXPRESSION>
WHEN <COND> THEN
    <EXEC-STATEMENTS>;
WHEN <COND> THEN
    <EXEC-STATEMENTS>;
WHEN <COND> THEN
    <EXEC-STATEMENTS>;
```

```
ELSE  
    <EXEC-STATEMENT>;  
END CASE;
```

ITERATION CONTROL STATEMENTS:

I. SIMPLE LOOP:

-> IT IS AN INFINITE LOOP. IF WE WANT BREAK A SIMPLE LOOP THEN WE SHOULD USE "EXIT" STATEMENT.

SYNTAX:

```
LOOP  
    <EXEC-STATEMENTS>;  
END LOOP;
```

II. WHILE LOOP:

SYNTAX:

```
WHILE <CONDITION>  
LOOP  
    <EXEC-STATEMENTS>;  
    <INCRE/DECRE>;  
END LOOP;
```

III. FOR LOOP:

-> BY DEFAULT, IT IS INCREMENTED BY 1.

SYNTAX:

```
FOR <INDEX_VARIABLE> IN <START_VALUE>..<END_VALUE>  
LOOP  
    <EXEC-STATEMENTS>;  
END LOOP;
```

CURSORS:

CURSOR IS A TEMP.MEMORY / A PRIVATE SQL AREA(PSA) / A WORK SPACE. CURSOR ARE TWO TYPES.THOSE ARE:

- I) EXPLICIT CURSOR (USER DEFINE CURSOR)
- II) IMPLICIT CURSOR (SYSTEM DEFINE CURSOR)

I) EXPLICIT CURSOR:

THESE CURSOR ARE CREATING BY USER FOR HOLDING MULTIPLE ROWS BUT WE CAN ACCESS ONLY ONE ROW AT TIME. (ONE BY ONE / ROW BY ROW MANNER).

IF WE WANT TO CREATE AN EXPLICIT CURSOR, WE NEED FOLLOW THE FOLLOWING FOUR STEPS.THOSE ARE

- 1) DECLARING A CURSOR
- 2) OPEN A CURSOR
- 3) FETCH ROWS FROM A CURSOR
- 4) CLOSE A CURSOR

STEPS TO CREATE EXPLICIT CURSOR:

1)DECLARING A CURSOR: IN THIS PROCESS WE DEFINE A CURSOR.

SYNTAX:

DECLARE CURSOR <CURSORNAME> IS < SELECT STATEMENT>;

2)OPENING A CURSOR: WHEN WE OPEN A CURSOR, IT WILL INTERNALLY EXECUTE THE SELECT STATEMENT THAT IS ASSOCIATED WITH THE CURSOR DECLARATION AND LOAD THE DATA INTO CURSOR.

SYNTAX:

OPEN < CURSORNAME>;

3)FETCHING DATA FROM THE CURSOR: IN THIS PROCESS WE ACCESS ROW BY ROW FROM CURSOR.

SYNTAX:

FETCH <CURSORNAME> INTO <VARIABLES>;

4)CLOSING A CURSOR: IN THIS PROCESS, IT RELEASES THE CURRENT RESULT SET OF THE CURSOR LEAVING THE DATASTRUCTURE AVAILABLE FOR REOPENING.

SYNTAX:

CLOSE <CURSORNAME>;

ATTRIBUTES OF EXPLICIT CURSORS:

IT SHOWS STATUS OF THE CURSOR AND IT RETURNS BOOLEAN VALUE.

SYNTAX:

<CURSOR_NAME>%<ATTRIBUTE>;

A. %ISOPEN:

IT RETURNS TRUE, WHEN THE CURSOR OPENS SUCCESSFULLY.

B. %FOUND:

IT RETURNS TRUE, WHEN THE CURSOR CONTAINS DATA.

C. %NOTFOUND:

IT RETURNS TRUE, WHEN THE CURSOR DOESN'T FIND ANY DATA.

D.%ROWCOUNT:

IT RETURNS NO. OF FETCH STATEMENTS EXECUTED.RETURN TYPE IS NUMBER.

EX1:

WA CURSOR PROGRAM TO FETCH A SINGLE ROW FROM EMP TABLE?

SOL:

```
DECLARE CURSOR C1 IS SELECT ENAME, SAL FROM EMP;
V_ENAME VARCHAR2(10);
V_SAL NUMBER (10);
BEGIN
OPEN C1;
FETCH C1 INTO V_ENAME, V_SAL;
DBMS_OUTPUT.PUT_LINE(V_ENAME||','||V_SAL);
CLOSE C1;
END;
/
```

OUTPUT:

SMITH,800

EX2: TO FETCH MULTIPLE ROWS FROM EMP TABLE BY USING LOOPING STATEMENTS?

I) BY USING SIMPLE LOOP:

- IT IS AN INFINITE LOOP.SO THAT WE NEED BREAK A LOOP THEN WE ARE USING "EXIT" STATEMENT.

SOL:

```
DECLARE CURSOR C1 IS SELECT ENAME, SAL FROM EMP;
V_ENAME VARCHAR2(10);
V_SAL NUMBER (10);
BEGIN
OPEN C1;
LOOP
FETCH C1 INTO V_ENAME, V_SAL;
```

```
EXIT WHEN C1%NOTFOUND;  
DBMS_OUTPUT.PUT_LINE(V_ENAME||','||V_SAL);  
END LOOP;  
CLOSE C1;  
END;
```

```
/
```

OUTPUT:

SMITH,800

ALLEN,1600

WARD,1250

II) BY USING WHILE LOOP:

```
DECLARE CURSOR C1 IS SELECT ENAME, SAL FROM EMP;  
V_ENAME VARCHAR2(10);  
V_SAL NUMBER (10);  
BEGIN  
OPEN C1;  
FETCH C1 INTO V_ENAME, V_SAL; ---LOOP START FROM 1ST ROW  
WHILE(C1%FOUND)  
LOOP  
DBMS_OUTPUT.PUT_LINE(V_ENAME||','||V_SAL);  
FETCH C1 INTO V_ENAME, V_SAL; ---LOOP CONTINUE UPTO LAST  
ROW  
END LOOP;  
CLOSE C1;  
END;  
/
```

III) BY USING FOR LOOP:

```
DECLARE CURSOR C1 IS SELECT ENAME, SAL FROM EMP;  
BEGIN  
FOR I IN C1  
LOOP  
DBMS_OUTPUT.PUT_LINE (I. ENAME||','||I.SAL);  
END LOOP;  
END;  
/
```

NOTE: WHENEVER WE ARE USING "FOR LOOP" STATEMENT IN CURSOR FOR FETCHING ROWS FROM A CURSOR MEMORY THEN THERE NO NEED TO OPEN CURSOR, FETCH ROW FROM CURSOR AND CLOSE CURSOR BY EXPLICITLY BECAUSE INTERNALLY ORACLE SERVER WILL OPEN, FETCH AND CLOSE CURSOR BY IMPLICITLY.

HERE, FOR LOOP EXECUTE NO. OF TIMES DEPENDS ON NO. OF ROWS IN CURSOR(C1). EVERY TIME FOR LOOP IS EXECUTE AND FETCH A ROW FROM C1 AND

ASSIGNED / STORED IN LOOP VARIABLE (I) AND LATER I LOOP VARIABLE VALUES ARE PRINTED.

EX3:

WA CURSOR PROGRAM TO FETCH TOP FIVE HIGHEST SALARIES EMPLOYEE ROWS FROM EMP TABLE?

SOL:

```
DECLARE CURSOR C1 IS SELECT ENAME, SAL FROM EMP ORDER BY  
SAL DESC;  
V_ENAME VARCHAR2(10);  
V_SAL NUMBER (10);  
BEGIN  
OPEN C1;  
LOOP  
FETCH C1 INTO V_ENAME, V_SAL;  
EXIT WHEN C1%ROWCOUNT>5;
```

```
DBMS_OUTPUT.PUT_LINE(V_ENAME||','||V_SAL);
END LOOP;
CLOSE C1;
END;
/
OUTPUT:
KING,5000
FORD,3000
SCOTT,3000
JONES,2975
BLAKE,2850
```

EX4:

WA CURSOR PROGRAM TO FETCH EVEN POSITION ROWS FROM EMP TABLE?

SOL:

```
DECLARE CURSOR C1 IS SELECT EMPNO, ENAME FROM EMP;
V_EMPNO NUMBER(10);
V_ENAME VARCHAR2(10);
BEGIN
OPEN C1;
LOOP
FETCH C1 INTO V_EMPNO, V_ENAME;
EXIT WHEN C1%NOTFOUND;
IF MOD(C1%ROWCOUNT,2) =0 THEN
DBMS_OUTPUT.PUT_LINE(V_EMPNO||','||V_ENAME);
END IF;
END LOOP;
CLOSE C1;
END;
/
```

OUTPUT:

7499, ALLEN

7566, JONES

7698, BLAKE

7788, SCOTT

7844, TURNER

EX5: WA CURSOR PROGRAM TO FETCH 9TH POSITION ROW FROM EMP TABLE?

SOL:

```
DECLARE CURSOR C1 IS SELECT ENAME FROM EMP;
V_ENAME VARCHAR2(10);
BEGIN
OPEN C1;
LOOP
FETCH C1 INTO V_ENAME;
EXIT WHEN C1%NOTFOUND;
IF C1%ROWCOUNT=9 THEN
DBMS_OUTPUT.PUT_LINE(V_ENAME);
END IF;
END LOOP;
CLOSE C1;
END;
/
```

OUTPUT:

KING

PARAMETERIZED CURSORS:

- WHENEVER WE ARE PASSING PARAMETERS TO THE CURSOR AT THE TIME DECLARATION IS CALLED AS PARAMETERIZED CURSOR. THESE PARAMETERIZED CURSOR WANT TO DECLARE THEN WE FOLLOW THE FOLLOWING TWO STEPS ARE

STEP1: DECLARE PARAMETERIZED CURSOR:

SYNTAX:

DECLARE CURSOR <CURSOR NAME> (<PARAMETER NAME> <DATATYPE>,) IS SELECT * FROM <TN> WHERE <CONDITION>;

STEP2: OPEN PARAMETERIZED CURSOR:

SYNTAX:

OPEN <CURSOR NAME> (<PARAMETER NAME>./<VALUE>);

EX1:

WA CURSOR PROGRAM TO ACCEPT DEPTNO AS A PARAMETER AND DISPLAY THE NO. OF EMPLOYEE WORKING IN THE GIVEN DEPTNO FROM EMP TABLE?

SOL:

```
DECLARE CURSOR C1(P_DEPTNO NUMBER) IS SELECT ENAME,
DEPTNO FROM EMP
WHERE DEPTNO=P_DEPTNO;
V_ENAME VARCHAR2(10);
V_DEPTNO NUMBER (10);
BEGIN
OPEN C1(&P_DEPTNO);
LOOP
FETCH C1 INTO V_ENAME, V_DEPTNO;
EXIT WHEN C1%NOTFOUND;
DBMS_OUTPUT.PUT_LINE(V_ENAME||','||V_DEPTNO);
```

```
END LOOP;  
CLOSE C1;  
END;  
/
```

OUTPUT:

```
ENTER VALUE FOR P_DEPTNO: 10  
MILLER,10  
CLARK,10  
KING,10
```

**EX2: WA CURSOR PROGRAM TO ACCEPT EMPNO AS A PARAMETER
AND CHECK THAT EMPLOYEE IS EXISTS OR NOT EXISTS IN EMP
TABLE?**

SOL:

```
DECLARE CURSOR C1(P_EMPNO NUMBER) IS SELECT ENAME FROM  
EMP  
WHERE EMPNO=P_EMPNO;  
V_ENAME VARCHAR2(10);  
BEGIN  
OPEN C1(&P_EMPNO);  
FETCH C1 INTO V_ENAME;  
IF C1%FOUND THEN  
DBMS_OUTPUT.PUT_LINE ('EMPLOYEE EXISTS, NAME IS: -  
'||V_ENAME);  
ELSE  
DBMS_OUTPUT.PUT_LINE ('EMPLOYEE NOT EXISTS');  
END IF;  
CLOSE C1;  
END;  
/
```

OUTPUT:

ENTER VALUE FOR P_EMPNO: 7788

EMPLOYEE EXISTS, NAME IS: -SCOTT

IMPLICIT CURSOR:

- THESE CURSORS ARE DECLARING BY ORACLE SERVER BY DEFAULT. ORACLE DECLARE THESE CURSORS AFTER EXECUTION OF DML COMMAND (INSERT / UPDATE / DELETE).

- IMPLICIT CURSOR TELLING US THE STATUS OF LAST DML COMMAND WHETHER SUCCESSFULL OR NOT.

ATTRIBUTES OF IMPLICIT CURSORS:

1. %ISOPEN:

- IT RETURNS TRUE THEN CURSOR SUCCESSFUL OPEN
OTHERWISE RETURNS FALSE.

2. %NOTFOUND:

- IT RETURNS TRUE THEN LAST DML COMMAND IS FAIL
OTHERWISE RETURNS FALSE.

3. %FOUND:

- IT RETRUNS TRUE THEN LAST DML COMMAND IS
SUCCESSFULLY EXECUTED

OTHERWISE RETURNS FALSE.

4. %ROWCOUNT:

- IT RETURNS NO. OF ROWS AFFECTED BY LAST DML
COMMAND.

EX:

```
DECLARE
  V_EMPNO NUMBER (10);
BEGIN
  V_EMPNO:=&V_EMPNO;
  DELETE FROM EMP WHERE EMPNO=V_EMPNO;
  IF SQL%FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('RECORD IS DELETED');
  ELSE
    DBMS_OUTPUT.PUT_LINE ('RECORD IS NOT EXISTS');
  END IF;
END;
/
```

OUTPUT:

ENTER VALUE FOR V_EMPNO: 7788

RECORD IS DELETED

REF. CURSORS:

- WHEN WE ASSIGN "SELECT STATEMENT" AT THE TIME OF OPENING CURSOR IS CALLED AS "REF.CURSOR".
- REF.CURSORS ARE TWO TYPES.THOSE ARE,

1. WEAK REF.CURSOR
2. STRONG REF.CURSOR

1. WEAK REF.CURSOR:

WHEN WE DECLARE REF.CURSOR WITHOUT RETURN TYPES IS CALLED AS WEAK REF.CURSOR.

SYNTAX:

<CURSOR VARIABLE NAME> SYS_REF_CURSOR; -----> (IT IS PRE-DEFINE TYPE)

EX:

```
DECLARE
C1 SYS_REFCURSOR;
I EMP%ROWTYPE;
BEGIN
OPEN C1 FOR SELECT * FROM EMP WHERE DEPTNO=10;
LOOP
FETCH C1 INTO I;
EXIT WHEN C1%NOTFOUND;
DBMS_OUTPUT.PUT_LINE (I.EMPNO||','||I.ENAME||','||I.SAL||','||I.DEPTNO);
END LOOP;
CLOSE C1;
END;
/
```

2)STRONG REF.CURSOR:

- WHEN WE DECLARE A REF. CURSOR ALONG WITH RETURN TYPE IS CALLED AS STRONG REF.CURSOR.

CREATE A USER DEFINE STRONG REF.CURSOR DATATYPE:

SYNTAX:

TYPE <TYPE NAME> IS REF CURSOR RETURN <TYPE>;---->(IT IS USER-DEFINE TYPE)

EX:

```
DECLARE
TYPE UD_REF_CURSOR IS REF CURSOR RETURN EMP%ROWTYPE;
C1 UD_REF_CURSOR;
I EMP%ROWTYPE;
BEGIN
OPEN C1 FOR SELECT * FROM EMP WHERE DEPTNO=10;
LOOP
```

```
FETCH C1 INTO I;
EXIT WHEN C1%NOTFOUND;
DBMS_OUTPUT.PUT_LINE (I.
EMPNO||','||I.ENAME||','||I.SAL||','||I.DEPTNO);
END LOOP;
CLOSE C1;
END;
/
```

EX. OF WEAK CURSOR ON MULTIPLE TABLES:

```
DECLARE
C1 SYS_REFCURSOR;
I EMP%ROWTYPE;
J DEPT%ROWTYPE;
V_DEPTNO NUMBER (10):=&V_DEPTNO;
BEGIN
IF V_DEPTNO = 10 THEN
OPEN C1 FOR SELECT * FROM EMP WHERE DEPTNO=10;
LOOP
FETCH C1 INTO I;
EXIT WHEN C1%NOTFOUND;
DBMS_OUTPUT.PUT_LINE (I.
EMPNO||','||I.ENAME||','||I.SAL||','||I.DEPTNO);
END LOOP;
ELSIF V_DEPTNO = 20 THEN
OPEN C1 FOR SELECT * FROM DEPT WHERE DEPTNO=20;
LOOP
FETCH C1 INTO J;
EXIT WHEN C1%NOTFOUND;
DBMS_OUTPUT.PUT_LINE (J. DEPTNO||','||J.DNAME||','||J.LOC);
END LOOP;
```

```
CLOSE C1;  
END IF;  
END;  
/
```

DIFFERENCES B/W WEAK AND STRONG REF.CURSOR:

WEAK REF CURSOR

STRONG REF CURSOR

1. IT IS NOT DECLARE WITH "RETURN" TYPE.

**1. DECLARING WITH
"RETURN" TYPE.**

2. PRE-DEFINE TYPE IS AVAILABLE

**2. PRE-DEFINE TYPE IS
NOT AVAILABLE THATS
WHY WE ARE
CREATING "USED
DEFINE TYPE".**

3. IT CAN ACCESS ROWS OF ANY

**3. IT CAN ACCESS
ROWS OF A SPECIFIC
TABLE ONLY.**

EX. OF STRONG CURSOR ON MULTIPLE TABLES:(NOT SUPPORTING)

DECLARE

TYPE UD_REF_CURSOR IS REF CURSOR RETURN EMP%ROWTYPE;

C1 UD_REF_CURSOR;

I EMP%ROWTYPE;

J DEPT%ROWTYPE;

V_DEPTNO NUMBER (10): =&V_DEPTNO;

BEGIN

IF V_DEPTNO = 10 THEN

OPEN C1 FOR SELECT * FROM EMP WHERE DEPTNO=10;

LOOP

```
FETCH C1 INTO I;
EXIT WHEN C1%NOTFOUND;
DBMS_OUTPUT.PUT_LINE (I.
EMPNO||','||I.ENAME||','||I.SAL||','||I.DEPTNO);
END LOOP;
ELSIF V_DEPTNO = 20 THEN
OPEN C1 FOR SELECT * FROM DEPT WHERE DEPTNO=20;
LOOP
FETCH C1 INTO J;
EXIT WHEN C1%NOTFOUND;
DBMS_OUTPUT.PUT_LINE (J. DEPTNO||','||J.DNAME||','||J.LOC);
END LOOP;
CLOSE C1;
END IF;
END;
/
```

EXCEPTION HANDLING IN PL/SQL

EXCEPTION: RUNTIME ERRORS ARE CALLED AN EXCEPTION. IF AT ANY TIME AN ERROR OCCURS IN THE PL/SQL BLOCK AT THAT TIME PL/SQL BLOCK EXECUTION IS STOPPED AND ORACLE RETURNS AN ERROR MESSAGE.

TO CONTINUE THE PROGRAM EXECUTION AND TO DISPLAY USER FRIENDLY MESSAGE EXCEPTION NEEDS TO BE HANDLE EXCEPTION INCLUDE EXCEPTION BLOCK IN PL/SQL.

EXCEPTIONS ARE CLASSIFIED INTO TWO TYPES. THOSE ARE

- 1) SYSTEM/PRE-DEFINED EXCEPTION
- 2) USER DEFINED EXCEPTION

SYNTAX:

DECLARE

<VARIABLES, CURSOR, USER DEFINE EXCEPTION>;

BEGIN

<STATEMENTS.....>;

EXCEPTION

WHEN <EXCEPTION NAME> THEN

<ERROR STATEMENTS.....>;

END;

1) SYSTEM/PRE-DEFINED EXCEPTION:

THESE ARE DEFINED BY ORACLE BY DEFAULT. WHENEVER RUNTIME ERROR IS OCCURRED IN PL/SQL THEN WE USE AN APPROPRIATE PRE-DEFINED EXCEPTION IN THE PROGRAM.

SOME PRE-DEFINED EXCEPTIONS:

- i. NO_DATA_FOUND
- ii. TOO_MANY_ROWS
- iii. ZERO_DIVIDE
- iv. INVALID_CURSOR
- v. CURSOR_ALREADY_OPEN.....ETC

NO DATA FOUND: WHENEVER PL/SQL BLOCK CARRY THE SELECT....INTO CLAUSE AND ALSO IF REQUIRED DATA NOT AVAILABLE IN A TABLE THEN ORACLE SERVER RETURNS AN EXCEPTION.

EX: ORA-1403: NO DATA FOUND

TO HANDLE THIS EXCEPTION ORACLE PROVIDED "NO_DATA_FOUND" EXCEPTION.

EX:

```
DECLARE TENAME VARCHAR2(20); TSAL NUMBER (10);
```

```
BEGIN
```

```
SELECT ENAME, SAL INTO TENAME, TSAL FROM EMPLOYEE WHERE  
EID=&EID;
```

```
DBMS_OUTPUT.PUT_LINE(TENAME||','||TSAL);
```

```
EXCEPTION
```

```
WHEN NO_DATA_FOUND THEN
```

```
DBMS_OUTPUT.PUT_LINE ('RECORD IS NOT FOUND');
```

```
END;
```

```
/
```

TOO MANY ROWS: WHEN SELECT.... INTO CLAUSE TRY TO RETURN MORE THAN ONE VALUE OR ONE ROW THEN ORACLE SERVER RETURNS AN ERROR.

EX: ORA-1422: EXACT FETCH RETURNS MORE THAN REQUESTED NUMBER OF ROWS.

TO HANDLE FOR THIS ERROR ORACLE, PROVIDE "TOO_MANY_ROWS" EXCEPTION.

EX:

```
DECLARE TSAL NUMBER (10);
```

```
BEGIN
```

```
SELECT SAL INTO TSAL FROM EMPLOYEE;
```

```
DBMS_OUTPUT.PUT_LINE(TSAL);
```

EXCEPTION

```
WHEN TOO_MANY_ROWS THEN  
DBMS_OUTPUT.PUT_LINE ('FETCHING MORE THAN ONE');  
END;
```

ZERO_DIVIDE: - IN ORACLE WHEN WE ARE TRIED TO PERFORM DIVISION WITH ZERO THEN ORACLE RETURN AN ERROR.

ORA-1476: DIVISOR IS EQUAL TO ZERO.

TO HANDLE FOR THIS ERROR ORACLE, PROVIDE "ZERO_DIVIDE" EXCEPTION

EX:

```
DECLARE X NUMBER (10); Y NUMBER (10); Z NUMBER (10);
```

BEGIN

```
X:=&X;
```

```
Y:=&Y;
```

```
Z:=X/Y;
```

```
DBMS_OUTPUT.PUT_LINE ('RESULT: -'||Z);
```

EXCEPTION

WHEN ZERO_DIVIDE THEN

```
DBMS_OUTPUT.PUT_LINE ('SECOND NUMBER SHOULD NOT BE  
ZERO');
```

END;

```
/
```

INVALID_CURSOR: WHEN WE ARE NOT OPENING THE CURSOR BUT WE ARE TRY TO PERFORM OPERATIONS ON CURSOR THEN ORACLE RETURNS AN ERROR.

EX: ORA-1001: INVALID_CURSOR

TO HANDLE THIS ERROR ORACLE, PROVIDE "INVALID_CURSOR" EXCEPTION.

EX:

DECLARE

CURSOR C1 IS SELECT * FROM EMPLOYEE;

TEID NUMBER (10); TENAME VARCHAR2(20); TSAL NUMBER (10);
TAGE NUMBER (10);

BEGIN

FETCH C1 INTO TEID, TENAME, TSAL, TAGE;

DBMS_OUTPUT.PUT_LINE (TEID||' '||TENAME||' '||TSAL||'
'||TAGE);

CLOSE C1;

EXCEPTION

WHEN INVALID_CURSOR THEN

DBMS_OUTPUT.PUT_LINE ('FIRST YOU MUST OPEN THE CURSOR');

END;

/

CURSOR_ALREADY_OPEN: BEFORE REOPENING THE CURSOR, WE MUST CLOSE THE CURSOR PROPERLY OTHERWISE ORACLE RETURNS AN ERROR I.E.

EX: ORA-6511: CURSOR_ALREADY_OPEN

TO HANDLE THIS ERROR ORACLE, PROVIDE 'CURSOR_ALREADY_OPEN' EXCEPTION.

EX:

```
DECLARE
CURSOR C1 IS SELECT * FROM EMPLOYEE;
TEID NUMBER (10); TENAME VARCHAR2(20); TSAL NUMBER (10);
TAGE NUMBER (10);

BEGIN
OPEN C1;
LOOP
FETCH C1 INTO TEID, TENAME, TSAL, TAGE;
EXIT WHEN C1%NOTFOUND;
DBMS_OUTPUT.PUT_LINE (TEID||'|'||TENAME||'|'||TSAL||'
'||TAGE);
END LOOP;
OPEN C1;
EXCEPTION
WHEN CURSOR_ALREADY_OPEN THEN
DBMS_OUTPUT.PUT_LINE ('WE MUST CLOSE THE CURSOR BEFORE
REOPEN');
END;
```

SQLCODE & SQLERRM: PL/SQL PROVIDES FOLLOWING BUILT-IN PROPERTIES WHICH ARE USED IN ERROR HANDLING.

SQLCODE RETURNS ERROR CODE.

SQLERRM RETURNS ERROR MESSAGE.

EX:

```
DECLARE
```

```
X NUMBER (10);
Y NUMBER (20);
Z NUMBER (10);
```

```
BEGIN  
X:=&X;  
Y:=&Y;  
Z:=X/Y;  
DBMS_OUTPUT.PUT_LINE(Z);  
EXCEPTION  
WHEN OTHERS THEN  
DBMS_OUTPUT.PUT_LINE(SQLCODE);  
DBMS_OUTPUT.PUT_LINE(SQLERRM);  
END;
```

OUTPUT:

ENTER VALUE FOR X: 10

ENTER VALUE FOR Y: 2

5

ENTER VALUE FOR X: 10

ENTER VALUE FOR Y: 0

-1476-----ERROR CODE

ORA-01476: DIVISOR IS EQUAL TO ZERO-----ERROR MESSAGE

USER DEFINE EXCEPTION:

- WHEN WE CREATE OUR OWN EXCEPTION NAME AND RAISE EXPLICITLY WHENEVER ISREQUIRED.THESE TYPE OF EXCEPTIONS ARE CALLED AS USER DEFINE EXCEPTIONS.

- GENERALLY, IF WE WANT TO RETURN MESSAGE AS PER CLIENT BUSSINESS RULES THEN WE MUST USE USER DEFINE EXCEPTIONS.

- TO CREATE A USER, DEFINE EXCEPTION NAME THEN WE FOLLOW THE FOLLOWING THREE STEPS ARE,

STEP1: DECLARE USER DEFINE EXCEPTION NAME:

SYNTAX:

<UD EXCEPTION NAME> EXCEPTION;

EX:

EX EXCEPTION;

STEP2: RAISE UD EXCEPTION:

SYNTAX:

RAISE <UD EXCEPTION NAME>;

EX:

RAISE EX;

STEP3: HANDLING UD EXCEPTION:

SYNTAX:

WHEN <UD EXCEPTION NAME> THEN

<STATEMENTS>;

END;

/

EX:

```
WHEN EX THEN  
DBMS_OUTPUT.PUT_LINE ('UD MESSAGE');  
END;  
/
```

EX:

```
DECLARE  
X INT;  
Y INT;  
Z INT;  
EX EXCEPTION; -----(1)  
BEGIN  
X:=&X;  
Y:=&Y;  
IF Y=0 THEN  
RAISE EX; -----(2)  
ELSE  
Z:=X/Y;  
DBMS_OUTPUT.PUT_LINE(Z);  
END IF;  
EXCEPTION  
WHEN EX THEN-----(3)  
DBMS_OUTPUT.PUT_LINE ('SECOND NUMBER NOT BE ZERO');  
END;  
/
```

RAISE APPLICATION_ERROR (NUMBER, MESSAGE):

- IT IS A PRE-DEFINE METHOD WHICH IS USED TO DISPLAY A USER DEFINE EXCEPTION INFORMATION IN FORM OF ORACLE FORMAT.

- RAISE STATEMENT IS USED TO RAISE EXCEPTION AND ALSO HANDLING EXCEPTION WHERE AS RIASE_APPLICATION_ERROR () STATEMENT IS USED TO RAISE EXCEPTION BUT NOT HANDLING EXCEPTION.

- THIS METHOD IS HAVING TWO ARGUMENTS ARE NUMBER AND MESSAGE.

HERE,

NUMBER - NUMBER SHOULD BE -20001 TO -20999

MESSAGE - USER DEFINE EXCEPTION MESSAGE.

EX:

DECLARE

X INT;

Y INT;

Z INT;

EX EXCEPTION;

BEGIN

X:=&X;

Y:=&Y;

IF Y=0 THEN

RAISE EX;

ELSE

Z:=X/Y;

DBMS_OUTPUT.PUT_LINE(Z);

END IF;

```
EXCEPTION  
WHEN EX THEN  
RAISE_APPLICATION_ERROR(-20457,'SECOND NUMBER NOT BE  
ZERO');  
END;  
/  
ENTER VALUE FOR X: 10  
ENTER VALUE FOR Y: 0  
ERROR AT LINE 1:  
ORA-20457: SECOND NUMBER NOT BE ZERO  
ORA-06512: AT LINE 17
```

PRAGMA EXCEPTION_INIT (UNNAMED EXCEPTION):

- IN ORACLE IF WE WANT TO HANDLE OTHER THAN ORACLE PRE-DEFINE EXCEPTION NAME ERRORS THEN WE MUST USE "UNNAMED EXCEPTION" METHOD. IN THIS METHOD WE MUST CREATE A USER DEFINE EXCEPTION AND ASSOCIATE THIS EXCEPTION NAME ALONG WITH SOME ERROR NUMBER BY USING "PRAGMA EXCEPTION_INIT" METHOD. THIS METHOD IS HAVING TWO ARGUMENTS ARE,

SYNTAX:

**PRAGMA EXCEPTION_INIT (<USER DEFINE EXCEPTION NAME>,
ERROR NUMBER)**

EX:

DECLARE

X EXCEPTION;

PRAGMA EXCEPTION_INIT (X, -2291);

BEGIN

```
INSERT INTO EMP (EMPNO, ENAME, DEPTNO) VALUES  
(1122,'SAI',50);  
EXCEPTION  
WHEN X THEN  
DBMS_OUTPUT.PUT_LINE ('NOT ALLOWED INTO EMP TABLE  
BECAUSE PARENT KEY IS NOT FOUND');  
END;  
/  
NOTE: IN THE ABOVE PL/SQL PROGRAM TO HANDLE -2291 ERROR  
THEN USE THE EXCEPTION NAME IS "X".
```

EXCEPTION PROPAGATION:

- EXCEPTION BLOCK HANDLES EXCEPTION WHICH WAS RAISED
IN BODY (EXECUTION BLOCK) BUT CANNOT HANDLE EXCEPTION
WHICH WILL RAISE IN DECLARATION BLOCK.

EX:

DECLARE

```
X VARCHAR2(3):='PQRS';  
BEGIN  
DBMS_OUTPUT.PUT_LINE(X);  
EXCEPTION  
WHEN VALUE_ERROR THEN  
DBMS_OUTPUT.PUT_LINE('INVALID STRING LENGTH');  
END;  
/
```

ERROR AT LINE 1:

**ORA-06502: PL/SQL: NUMERIC OR VALUE ERROR: CHARACTER
STRING BUFFER TOO SMALL.**

- TO OVERCOME THE ABOVE PROBLEM, WE NEED TO PREPARE
NESTED PL/SQL BLOCK TO HANDLE EXCEPTION WHICH WAS RAISED

IN DECLARATION BLOCK THIS IS CALLED AS EXCEPTION PROPAGATION.

SOL:

BEGIN

DECLARE

X VARCHAR2(3):='PQRS';

BEGIN

DBMS_OUTPUT.PUT_LINE(X);

EXCEPTION

WHEN VALUE_ERROR THEN

DBMS_OUTPUT.PUT_LINE('INVALID STRING LENGTH');

END;

EXCEPTION

WHEN VALUE_ERROR THEN

DBMS_OUTPUT.PUT_LINE('STRING LENGTH IS GREATER THAN THE SIZE OF VARIABLE X');

END;

/

OUTPUT:

STRING LENGTH IS GREATER THAN THE SIZE OF VARIABLE X.

NOTE:

- IN PL/SQL EXCEPTIONS ARE OCCURRED IN EXECUTION BLOCK, DECLARATION BLOCK.WHENEVER EXCEPTIONS ARE OCCURRED IN EXECUTION BLOCK THOSE EXCEPTIONS ARE HANDLED IN INNER BLOCK WHERE AS WHEN EXCEPTIONS ARE OCCURED IN DECLARATION BLOCK THOSE EXCEPTIONS ARE

HANDLED IN OUTER BLOCK ONLY.THIS MECHANISM IS CALLED AS "EXCEPTION PROPAGATION".

SUB BLOCKS:

A SUB BLOCK IS A NAMED BLOCK OF CODE THAT IS DIRECTLY SAVED ON THE DB SERVER AND IT CAN BE EXECUTED WHEN AND WHERE IT IS REQUIRED. WE HAVE FOUR TYPES OF SUB BLOCKS IN ORACLE.

1. STORED PROCEDURES

2. STORED FUNCTIONS

3. PACKAGES

4. TRIGGERS

STORED PROCEDURES:

A STORED PROCEDURE IS A DATABASE OBJECT WHICH CONTAINS PRECOMPILED QUERIES. STORED PROCEDURES ARE A BLOCK OF CODE DESIGNED TO PERFORM A TASK WHENEVER WE CALLED AND MAY BE OR MAY NOT BE RETURN A VALUE.

WHY WE NEED STORED PROCEDURE:

WHENEVER WE WANT TO EXECUTE A SQL QUERY FROM AN APPLICATION THE SQL QUERY WILL BE FIRST PARSED (I.E. COMPILED) FOR EXECUTION WHERE THE PROCESS OF PARSING IS TIME CONSUMING BECAUSE PARSING OCCURS EACH AND EVERY TIME, WE EXECUTE THE QUERY OR STATEMENT.

TO OVERCOME THE ABOVE PROBLEM, WE WRITE SQL STATEMENTS OR QUERY UNDER STORED PROCEDURE AND EXECUTE, BECAUSE A STORED PROCEDURE IS A PRE-COMPILED BLOCK OF CODE WITHOUT PARSING THE STATEMENTS GETS EXECUTED WHENEVER THE PROCEDURES ARE CALLED WHICH CAN INCREASE THE PERFORMANCE OF AN APPLICATION.

ADVANTAGES OF STORED PROCEDURE:

- AS THERE IS NO UNNECESSARY COMPILEATION OF QUERIES, THIS WILL REDUCE BURDEN ON DATABASE.
- APPLICATION PERFORMANCE WILL BE IMPROVED
- USER WILL GET QUICK RESPONSE
- CODE REUSABILITY & SECURITY.

PROCEDURE SYNTAX:

```
CREATE OR REPLACE PROCEDURE <PROCEDURE_NAME>
[ PARAMETER NAME [MODE TYPE] DATATYPE,....]
IS
<VARIABLE DECLARATION>;
BEGIN
<EXEC STATEMENTS>;
[ EXCEPTION BLOCK
<EXEC-STATEMENTS>]
END;
```

TO EXECUTE THE PROCEDURE:

SYNTAX1:

```
EXECUTE / EXEC <PROCEDURE_NAME>;
```

SYNTAX2:(ANONYMOUS BLOCK)

```
BEGIN
<PROCEDURE_NAME>;
END;
```

EXAMPLES ON PROCEDURE WITHOUT PARAMATERS:

EX1:

```
CREATE OR REPLACE PROCEDURE MY_PROC
IS
BEGIN
DBMS_OUTPUT.PUT_LINE ('WELCOME TO PROCEDURES....');
END MY_PROC;
```

TO EXECUTE THE PROCEDURE:

SYNTAX1:

EX: EXEC MY_PROC;

SYNTAX2:

EX: BEGIN

MY_PROC;

END;

EX2: WRITE A PROCEDURE TO DISPLAY SUM OF TWO NUMBERS.

CREATE OR REPLACE PROCEDURE ADD_PROC

IS

A NUMBER := 10;

B NUMBER := 20;

BEGIN

DBMS_OUTPUT.PUT_LINE ('SUM OF TWO NUMBERS ='||(A+B));

END ADD_PROC;

EXAMPLES ON PROCEDURES WITH PARAMETERS:

EX3:

CREATE OR REPLACE PROCEDURE ADD_PROC (A NUMBER, B NUMBER)

IS

BEGIN

DBMS_OUTPUT.PUT_LINE ('SUM OF TWO NUMBERS ='||(A+B));

END ADD_PROC;

TO EXECUTE ABOVE PROCEDURE:

EXEC ADD_PROC (10,60);

EXEC ADD_PROC (&A, &B);

EX4: WRITE A PROCEDURE TO ACCEPT EMPLOYEE NUMBER AND DISPLAY CORRESPONDING EMPLOYEE NET SALARY.

```
CREATE OR REPLACE PROCEDURE EMP_PROC (TEMPNO
EMP.EMPNO%TYPE)

IS

TSAL EMP.SAL%TYPE;
TCOMM EMP.COMM%TYPE;
NETSAL NUMBER;
COMM_NULL EXCEPTION;

BEGIN

SELECT SAL, COMM INTO TSAL, TCOMM FROM EMP WHERE
EMPNO=TEMPNO;

IF TCOMM IS NULL THEN

RAISE COMM_NULL;

END IF;

NETSAL:=TSAL+TCOMM;

DBMS_OUTPUT.PUT_LINE ('GIVEN EMPLOYEE NET SALARY =
'||NETSAL);

EXCEPTION

WHEN COMM_NULL THEN

RAISE_APPLICATION_ERROR (-20001,'GIVEN EMPLOYEE IS NOT
GETTING COMMISSION. ');

WHEN NO_DATA_FOUND THEN

RAISE_APPLICATION_ERROR (-20002, 'SUCH EMPLOYEE
NUMBER IS NOT EXIST. ');

END EMP_PROC;
```

PROCEDURES RETURN VALUES THROUGH PARAMETER MODES:

- THERE ARE THREE TYPES OF PARAMETERS MODES.

IN -> IT ACCEPTS INPUT INTO STORED PROCEDURE(DEFAULT)

OUT -> IT RETURNS OUTPUT THROUGH STORED PROCEDURE

IN OUT -> BOTH ACCEPTING AND ALSO RETURN.

EX. ON "IN" PARAMETERS:

EX5:

**CREATE OR REPLACE PROCEDURE ADD_PROC (A IN NUMBER, B
IN NUMBER)**

IS

BEGIN

DBMS_OUTPUT.PUT_LINE ('SUM OF TWO NUMBERS ='||(A+B));

END ADD_PROC;

EXEC ADD_PROC (90,30);

EX6:

**CREATE A SP TO INPUT EMPNO AND DISPLAY THAT EMPLOYEE
NAME, SAL FROM EMP TABLE?**

**SQL> CREATE OR REPLACE PROCEDURE SP1(P_EMPNO IN
NUMBER)**

IS

V_ENAME VARCHAR2(10);

V_SAL NUMBER (10);

BEGIN

**SELECT ENAME, SAL INTO V_ENAME, V_SAL FROM EMP WHERE
EMPNO=P_EMPNO;**

DBMS_OUTPUT.PUT_LINE(V_ENAME||','||V_SAL);

END;

/

PROCEDURE CREATED.

SQL> EXECUTE SP1(7788);

SCOTT,3000

EX ON "OUT" PARAMETERS:

EX7:

SQL> CREATE OR REPLACE PROCEDURE SP2(X IN NUMBER, Y OUT NUMBER)

IS

BEGIN

Y:=X*X*X;

END;

/

PROCEDURE CREATED.

SQL> EXECUTE SP2(5);

ERROR AT LINE 1:

ORA-06550: LINE 1, COLUMN 7:

PLS-00306: WRONG NUMBER OR TYPES OF ARGUMENTS IN CALL TO 'SP2'

NOTE: TO OVERCOME THE ABOVE PROBLEM THEN WE FOLLOW THE FOLLOWING 3 STEPS,

STEP1: DECLARE REFERENCED /BIND VARIABLE FOR "OUT" PARAMETERS IN SP:

SYNTAX:

VAR[TABLE] <REF.VARIABLE NAME> <DT>[SIZE];

STEP2: TO ADD A REFERENCED /BIND VARIABLE TO A SP:

SYNTAX:

EXECUTE <PNAME> (VALUE1, VALUE2,:<REF.VARIABLE NAME>....);

STEP3: PRINT REFERENCED VARIABLES:

SYNTAX:

PRINT <REF.VARIABLE NAME>;

EXECUTION PLAN OF "OUT" PARAMETERS IN SP:

SQL> VAR RY NUMBER;

SQL> EXECUTE SP2(5,:RY);

PL/SQL PROCEDURE SUCCESSFULLY COMPLETED.

SQL> PRINT RY;

RY

125

EX8:

**CREATE A SP TO INPUT EMPNO AS A "IN" PARAMETER AND
RETURNS THAT EMPLOYEE PROVIDENT FUND, PROFESSIONAL TAX
AT 10%,20% ON BASIC SALARY BY USING "OUT" PARAMETERS?**

SQL> CREATE OR REPLACE PROCEDURE SP3(P_EMPNO IN
NUMBER, PF OUT NUMBER, PT OUT NUMBER)

IS

V_SAL NUMBER (10);

BEGIN

SELECT SAL INTO V_SAL FROM EMP WHERE EMPNO=P_EMPNO;

PF:=V_SAL*0.1;

PT:=V_SAL*0.2;

END;

/

PROCEDURE CREATED.

SQL> VAR RPF NUMBER;

SQL> VAR RPT NUMBER;

SQL> EXECUTE SP3(7788,:RPF,:RPT);

PL/SQL PROCEDURE SUCCESSFULLY COMPLETED.

SQL> PRINT RPF RPT;

EX9:

CREATE OR REPLACE PROCEDURE ADD_PROC (A IN NUMBER, B IN NUMBER, C OUT NUMBER)

IS

BEGIN

C:=A+B;

END ADD_PROC;

OUTPUT:

VAR R NUMBER;

EXECUTE ADD_PROC (10,20,:R);

PRINT R;

EX. ON "IN OUT" PARAMETERS:

EX10:

SQL> CREATE OR REPLACE PROCEDURE SP4(X IN OUT NUMBER)

AS

BEGIN

X:= X*X;

END;

/

PROCEDURE CREATED.

SQL> EXECUTE SP4(5);

ERROR AT LINE 1:

ORA-06550: LINE 1, COLUMN 11:

**PLS-00363: EXPRESSION '5' CANNOT BE USED AS AN
ASSIGNMENT TARGET**

**NOTE: TO OVERCOME THE ABOVE PROBLEM THEN WE FOLLOW THE
FOLLOWING 4 STEPS,**

**STEP1: DECLARE REFERENCED VARIABLE FOR "OUT" PARAMETERS
IN SP:**

SYNTAX:

VAR[TABLE] <REF.VARIABLE NAME> <DT>[SIZE];

STEP2: ASSIGN A VALUE TO REFERENCED VARIABLE:

SYNTAX:

EXECUTE <REF.VARIABLE NAME> := <VALUE>;

STEP3: TO ADD A REFERENCED VARIABLE TO A SP:

SYNTAX:

EXECUTE <PNAME> (:<REF.VARIABLE NAME>.....);

STEP4: PRINT REFERENCED VARIABLES:

SYNTAX:

PRINT <REF.VARIABLE NAME>;

OUTPUT:

SQL> VAR RX NUMBER;

SQL> EXECUTE :RX := 10;

SQL> EXECUTE SP4(:RX);

SQL> PRINT RX;

NOTE: ALL PROCEDURES NAMES ARE STORED IN USER_OBJECTS.

SELECT OBJECT_NAME FROM USER_OBJECTS;

EX:

**SELECT OBJECT_NAME FROM USER_OBJECTS WHERE
OBJECT_TYPE='PROCEDURE';**

NOTE: PROCEDURE BODIES ARE STORED IN USER_SOURCE.

EX:

SELECT TEXT FROM USER_SOURCE WHERE NAME='EMP_PROC';

DROPPING PROCEDURES:

SYNTAX:

SQL> DROP PROCEDURE <PROCEDURE_NAME>;

EX: DROP PROCEDURE MY_PROC;

STORED FUNCTIONS:

A FUNCTION IS BLOCK OF CODE TO PERFORM SOME TASK AND MUST RETURN A VALUE.THESE FUNCTIONS ARE CREATED BY USER EXPLICETLY.SO THAT WE CAN ALSO CALLED AS "USER DEFINED FUNCTION"

SYNTAX:

```
CREATE OR REPLACE FUNCTION <FUNCTION_NAME>
[(ARGUMENT DATATYPE,
      ARGUMENT DATATYPE,)]
      RETURN <DATATYPE>
      IS
BEGIN
<EXEC-STATEMENTS>;
RETURN (VALUE);
END <FUNCTION_NAME>;
/
```

HOW TO CALL A STORED FUNCTION:

```
SELECT <FNAME>(VALUES) FROM DUAL;
```

EX: CREATE A SF TO ACCEPT EMPLOYEE NUMBER AND RETURN THAT EMPLOYEE NAME FROM EMP TABLE?

```
CREATE OR REPLACE FUNCTION SF1(P_EMPNO NUMBER)
      RETURN VARCHAR2
      AS
      V_ENAME VARCHAR2(10);
BEGIN
      SELECT ENAME INTO V_ENAME FROM EMP WHERE
      EMPNO=P_EMPNO;
      RETURN V_ENAME;
END;
/
```

FUNCTION CREATED.

SQL> SELECT SF1(7566) FROM DUAL;

EX: CREATE A SF TO INPUT DEPARTMENT NAME AND RETURN SUM OF SALARY OF DEPARTMENT?

```
FUNCTION SF1(P_DNAME VARCHAR2)
RETURN NUMBER
AS
V_TOTSAL NUMBER (10);
BEGIN
SELECT SUM(SAL) INTO V_TOTSAL FROM EMP E,DEPT D
WHERE E. DEPTNO=D.DEPTNO AND DNAME=P_DNAME;
RETURN V_TOTSAL;
END;
/
```

SQL> SELECT SF1('SALES') FROM DUAL;

EX: CREATE A SF TO RETURN NO. OF EMPLOYEE IN BETWEEN GIVEN DATES?

```
FUNCTION SF2(SD DATE, ED DATE)
RETURN NUMBER
AS
V_COUNT NUMBER (10);
BEGIN
SELECT COUNT (*) INTO V_COUNT FROM EMP
WHERE HIREDATE BETWEEN SD AND ED;
RETURN V_COUNT;
END;
/
```

SQL> SELECT SF2('01-JAN-81','31-DEC-81') FROM DUAL;

EX: CREATE A SF TO INPUT EMPLOYEE NUMBER AND RETURN THAT EMPLOYEE GROSS SALARY AS PER GIVEN CONDITIONS ARE

I) HRA ----- 10%

II) DA ----- 20%

III) PF -----10%.

FUNCTION SF3(P_EMPNO NUMBER)

RETURN NUMBER

AS

V_BSAL NUMBER (10);

V_HRA NUMBER (10);

V_DA NUMBER (10);

V_PF NUMBER (10);

V_GROSS NUMBER (10);

BEGIN

SELECT SAL INTO V_BSAL FROM EMP WHERE EMPNO=P_EMPNO;

V_HRA: =V_BSAL*0.1;

V_DA: =V_BSAL*0.2;

V_PF: =V_BSAL*0.1;

V_GROSS: =V_BSAL+V_HRA+V_DA+V_PF;

RETURN V_GROSS;

END;

/

SQL> SELECT SF3(7788) FROM DUAL;

EX: WRITE A FUNCTION TO FIND SIMPLE INTEREST.

CREATE OR REPLACE FUNCTION SI (P NUMBER, T NUMBER, R NUMBER)

RETURN NUMBER

IS

SIMPLE_INT NUMBER;

```
BEGIN  
SIMPLE_INT: =(P*T*R)/100;  
RETURN (SIMPLE_INT);  
END SI;  
  
/  
  
> GENERALLY, FUNCTIONS ARE EXECUTED BY USING  
'SELECT' STATEMENT.
```

```
SQL> SELECT SI (1000,2,10) FROM DUAL;
```

EX: CREATE A SF TO FIND EXPERIENCE OF GIVEN EMPLOYEE?

```
CREATE OR REPLACE FUNCTION EMP_EXP (TEMPNO  
EMP.EMPNO%TYPE)
```

```
RETURN VARCHAR2
```

```
IS
```

```
TDATE EMP.HIREDATE%TYPE;
```

```
TEXP NUMBER;
```

```
BEGIN
```

```
SELECT HIREDATE INTO TDATE FROM EMP
```

```
WHERE EMPNO=TEMPNO;
```

```
TEXP:=ROUND((SYSDATE-TDATE)/365);
```

```
RETURN (TEMPNO||' EMPLOYEE EXPERIENCE IS '||TEXP||'  
YEARS.');
```

```
EXCEPTION
```

```
WHEN NO_DATA_FOUND THEN
```

```
RETURN ('GIVEN EMPLOYEE RECORD NOT FOUND.');
```

```
END EMP_EXP;
```

```
SQL> SELECT EMP_EXP (7788) FROM DUAL;
```

```
SQL> SELECT EMP_EXP(EMPNO) FROM EMP;
```

FUNCTION FOR TO CALCULATE EMPLOYEE EXPERIENCE:

```
CREATE OR REPLACE FUNCTION EMP_EXPE (TEMPNO
EMP.EMPNO%TYPE)
    RETURN NUMBER
    IS
    TEXP NUMBER;
BEGIN
    SELECT ROUND((SYSDATE-HIREDATE)/365) INTO TEXP FROM
EMP
        WHERE EMPNO=TEMPNO;
    RETURN(TEXP);
END EMP_EXPE;
```

NOTE:

ALL FUNCTIONS ARE STORED IN USER_OBJECTS.

ALL FUNCTIONS BODIES ARE STORED IN 'USER_SOURCE' SYSTEM TABLE.

> TO SEE THE FUNCTION BODY.

EX:

```
SQL> SELECT TEXT FROM USER_SOURCE WHERE
NAME='EMP_EXPE';
```

DROPPING FUNCTIONS:

SYNTAX:

```
SQL> DROP FUNCTION <FUNCTION_NAME>;
```

EX:

```
SQL> DROP FUNCTION EMP_EXPE;
```

PACKAGES:

- > IT IS A COLLECTION OF VARIABLES, CURSORS, PROCEDURES & FUNCTIONS ARE STORED IN ONE LOCATION.
- > EASY TO SHARE THE SUBPROGRAMS IN APPLICATION S/W TOOLS
- > THEY IMPROVE PERFORMANCE WHILE ACCESSING SUBPROGRAMS FROM CLIENT LOCATION
- > THEY ARE STORED IN "USER_SOURCE" SYSTEM TABLE
- > THEY SUPPORTS FUNCTION OVERLOADING, ENCAPSULATION & DATABINDING
- > IT HAS TWO PARTS

1. PACKAGE SPECIFICATION:

IT HOLDS THE DECLARATION OF VARIABLES, CURSORS & SUBPROGRAMS.

SYNTAX:

CREATE [OR REPLACE] PACKAGE <PACKAGE NAME>

IS / AS

<DECLARE VARIABLES, CURSORS, SUB BLOCKS>;

END;

/

2. PACKAGE BODY:

IT HOLDS THE BODY OF SUBPROGRAMS.

SYNTAX:

CREATE [OR REPLACE] PACKAGE BODY <PACKAGE NAME>

IS / AS

<IMPLEMENTING SUB BLOCKS>;

END;

/

EX1:

CREATE A PACKAGE TO ENCAPSULATE / BIND MULTIPLE PROCEDURES?

SQL> CREATE OR REPLACE PACKAGE PK1

```
IS  
PROCEDURE SP1;  
PROCEDURE SP2;  
END;  
/
```

PACKAGE CREATED.

SQL> CREATE OR REPLACE PACKAGE BODY PK1

```
IS  
PROCEDURE SP1  
AS  
BEGIN  
DBMS_OUTPUT.PUT_LINE ('MY NAME IS PROCEDURE1');  
END SP1;  
PROCEDURE SP2  
AS  
BEGIN  
DBMS_OUTPUT.PUT_LINE ('MY NAME IS PROCEDURE2');  
END SP2;  
END;  
/
```

PACKAGE BODY CREATED.

SQL> EXECUTE PK1.SP1;

MY NAME IS PROCEDURE

PL/SQL PROCEDURE SUCCESSFULLY COMPLETED.

```
SQL> EXECUTE PK1.SP2;
MY NAME IS PROCEDURE2
PL/SQL PROCEDURE SUCCESSFULLY COMPLETED.

EX2:

CREATE A PACKAGE TO BIND VARIABLE, PROCEDURE &
FUNCTION?

CREATE PACKAGE PK2
IS
A NUMBER (10): =2000;
PROCEDURE SP1;
FUNCTION SF1(X NUMBER) RETURN NUMBER;
END;
/
PACKAGE CREATED.

CREATE PACKAGE BODY PK2
IS
PROCEDURE SP1
AS
X NUMBER (10);
BEGIN
X:=A/2;
DBMS_OUTPUT.PUT_LINE(X);
END SP1;
FUNCTION SF1(X NUMBER)
RETURN NUMBER
AS
BEGIN
RETURN X*A;
END SF1;
END;
/
```

PACKAGE BODY CREATED.

EX3:

PACKAGE SPECIFICATION:

**CREATE OR REPLACE PACKAGE MY_PACK
IS.**

```
RESULT VARCHAR2(50);  
PROCEDURE EMP_EXP (TEMPNO EMP.EMPNO%TYPE);  
FUNCTION EMP_NETSAL (TEMPNO EMP.EMPNO%TYPE) RETURN  
VARCHAR2;  
END MY_PACK;
```

PACKAGE BODY:

**CREATE OR REPLACE PACKAGE BODY MY_PACK
IS**

PROCEDURE EMP_EXP (TEMPNO EMP.EMPNO%TYPE)

IS

TDATE EMP.HIREDATE%TYPE; -- PRIVATE VARIABLES

TEXP NUMBER;

BEGIN

**SELECT HIREDATE INTO TDATE FROM EMP WHERE
EMPNO=TEMPNO;**

TEXP:=ROUND((SYSDATE-TDATE)/365);

**DBMS_OUTPUT.PUT_LINE (TEMPNO||' EMPLOYEE EXPERIENCE
IS'||TEXP||'YEARS.');**

END EMP_EXP;

FUNCTION EMP_NETSAL (TEMPNO EMP.EMPNO%TYPE)

RETURN VARCHAR2

IS

TSAL EMP.SAL%TYPE;

TCOMM EMP.COMM%TYPE;

BEGIN

SELECT SAL+NVL(COMM,0) INTO RESULT FROM EMP

```
        WHERE EMPNO=TEMPNO;  
    RETURN (TEMPNO||'EMPLOYEE NET SALARY RS.'||RESULT);  
END EMP_NETSAL;  
END MY_PACK;
```

TO EXECUTE ABOVE PACKAGE:

```
EXEC MY_PACK.EMP_EXP (7788);  
SELECT MY_PACK.EMP_NETSAL (7788) FROM DUAL;
```

FUNCTION OVERLOADING USING PACKAGE:

EX:

```
SQL> CREATE OR REPLACE PACKAGE PK3
```

```
IS
```

```
    FUNCTION SF1(X NUMBER, Y NUMBER) RETURN NUMBER
```

```
    FUNCTION SF1(A NUMBER, B NUMBER, C NUMBER) RETU
```

```
    END;
```

```
/
```

PACKAGE CREATED.

```
SQL> CREATE OR REPLACE PACKAGE BODY PK3
```

```
IS
```

```
    FUNCTION SF1(X NUMBER, Y NUMBER)
```

```
    RETURN NUMBER
```

```
    AS
```

```
    BEGIN
```

```
    RETURN (X*Y);
```

```
    END SF1;
```

```
    FUNCTION SF1(A NUMBER, B NUMBER, C NUMBER)
```

```
    RETURN NUMBER
```

```
    AS
```

```
    BEGIN
```

```
    RETURN A+B+C;
```

```
END SF1;  
END;  
/
```

PACKAGE BODY CREATED.

```
SQL> SELECT PK3.SF1 (10,20,30) FROM DUAL;
```

```
PK3.SF1 (10,20,30)
```

```
-----  
60
```

```
SQL> SELECT PK3.SF1 (10,20) FROM DUAL;
```

```
PK3.SF1 (10,20)
```

```
-----  
200
```

EX:

```
CREATE OR REPLACE PACKAGE FO_PACK
```

```
IS
```

```
FUNCTION ADDVAL (A NUMBER, B NUMBER) RETURN NUMBER;
```

```
FUNCTION ADDVAL (A NUMBER, B NUMBER, C NUMBER) RETURN  
NUMBER;
```

```
FUNCTION ADDVAL (STR1 VARCHAR2, STR2 VARCHAR2) RETURN  
VARCHAR2;
```

```
FUNCTION ADDVAL (STR1 VARCHAR2, STR2 VARCHAR2, STR3  
VARCHAR2) RETURN VARCHAR2;
```

```
END FO_PACK;
```

PACKAGE BODY:

CREATE OR REPLACE PACKAGE BODY FO_PACK
IS
FUNCTION ADDVAL (A NUMBER, B NUMBER) RETURN NUMBER
IS
BEGIN
RETURN(A+B);
END ADDVAL;

FUNCTION ADDVAL (A NUMBER, B NUMBER, C NUMBER) RETURN NUMBER

IS
BEGIN
RETURN(A+B+C);
END ADDVAL;

FUNCTION ADDVAL (STR1 VARCHAR2, STR2 VARCHAR2) RETURN VARCHAR2

IS
BEGIN
RETURN(STR1||STR2);
END ADDVAL;

FUNCTION ADDVAL (STR1 VARCHAR2, STR2 VARCHAR2, STR3 VARCHAR2) RETURN VARCHAR2

IS
BEGIN
RETURN(STR1||STR2||STR3);
END ADDVAL;
END FO_PACK;
/

CALLING PACKAGE:

```
SELECT FO_PACK.ADDVAL(10,20) FROM DUAL;  
SELECT FO_PACK.ADDVAL('RAMA ','KRISHNA ','RAJU') FROM  
DUAL;  
SELECT FO_PACK.ADDVAL(10,20,50) FROM DUAL;
```

NOTE:

- > ALL PACKAGES BODIES ARE STORED IN 'USER_SOURCE'.
- > TO SEE THE PACKAGE BODY.

EX:

```
SQL> SELECT TEXT FROM USER_SOURCE WHERE  
NAME='FO_PACK';
```

DROPPING PACKAGE BODY:

SYNTAX:

```
SQL> DROP PACKAGE BODY <PACKAGE NAME>;
```

EX:

```
SQL> DROP PACKAGE BODY MY_PACK;
```

DROPPING PACKAGES:

SYNTAX:

```
SQL> DROP PACKAGE <PACKAGE NAME>;
```

EX:

```
SQL> DROP PACKAGE MY_PACK;
```

TRIGGERS:

A SET OF PL/SQL STATEMENTS STORED PERMANENTLY IN DATABASE AND "AUTOMATICALLY" ACTIVATED WHEN EVER AN EVENT RAISING STATEMENT (DML / DDL) IS PERFORMED.

THEY ARE USED TO IMPOSE USER DEFINED RESTRICTIONS(OR)BUSINESS RULES ON TABLE / SCHEMA.THEY ARE ALSO ACTIVATED WHEN TABLES ARE MANIPULATED BY OTHER USERS OR BY OTHER APPLICATION S/W TOOLS.THEY PROVIDE HIGH SECURITY ON TABLES.THEY ARE STORED IN "USER_TRIGGER" SYSTEM TABLE.

1. DML TRIGGERS

2. DDL TRIGGER / DB TRIGGERS

PURPOSE OF TRIGGERS:

- 1. TO INVOKING USER DEFINED MESSAGE (OR) ALERTS AT EVENT RAISE.**
- 2. TO CONTROL DML / DDL OPERATIONS.**
- 3. TO IMPLEMENTING BUSINESS LOGICAL CONDITIONS.**
- 4. TO VALIDATING DATA.**
- 5. FOR AUDITING.**

1. DML TRIGGERS:

- THESE TRIGGERS ARE EXECUTED BY SYSTEM AUTOMATICALLY WHEN USER PERFORM DML (INSERT / UPDATE / DELETE) OPERATIONS ON A SPECIFIC TABLE.

SYNTAX:

```
CREATE OR REPLACE TRIGGER <TRIGGER_NAME>
BEFORE/AFTER INSERT OR UPDATE OR DELETE
[ OF <COLUMNS> ] ON <TABLE NAME>
[ FOR EACH ROW ]
WHEN <CONDITION> (TRUE -> EXECUTES THE TRIGGER,
FALSE - NOT EXECUTE)
DECLARE
<VARIABLE DECLARATION>;]
```

```
BEGIN  
  <EXEC STATEMENTS>;  
  [ EXCEPTION  
    <EXEC STATEMENTS>;]  
END;
```

TRIGGER EVENT:

- INDICATES WHEN TO ACTIVATE THE TRIGGER

BEFORE TRIGGER:

1. FIRST TRIGGER BODY IS EXECUTED
2. LATER DML COMMAND EXECUTED

AFTER TRIGGER:

1. FIRST DML COMMAND EXECUTED
2. LATER TRIGGER BODY EXECUTED

TRIGGER LEVELS:

- TRIGGER CAN CREATE AT TWO LEVELS.

A. ROW LEVEL TRIGGER:

- IN THIS LEVEL, TRIGGER BODY(LOGIC) IS EXECUTING FOR EACH ROW WISE FOR A DML OPERATION.

EX:

CREATE OR REPLACE TRIGGER TR1

AFTER UPDATE ON TEST

FOR EACH ROW

BEGIN

DBMS_OUTPUT.PUT_LINE('HELLO');

END;

/

TRIGGER CREATED.

TESTING:

SQL> UPDATE TEST SET SAL=10000 WHERE SAL=15000;

HELLO

HELLO

2 ROWS UPDATED.

B. STATEMENT TRIGGER:

- IN THIS LEVEL, TRIGGER BODY IS EXECUTING ONLY ONE TIME FOR A DML OPERATION.

EX:

CREATE OR REPLACE TRIGGER TR1

AFTER UPDATE ON TEST

BEGIN

DBMS_OUTPUT.PUT_LINE('HELLO');

END;

/

TRIGGER CREATED.

TESTING:

SQL> UPDATE TEST SET SAL=12000 WHERE SAL=10000;

HELLO

2 ROWS UPDATED.

BIND VARIABLES:

- BIND VARIABLES ARE JUST LIKE VARIABLES WHICH ARE USED TO STORE VALUES WHILE INSERTING, UPDATING, DELETING DATA FROM A TABLE. THESE ARE TWO TYPES,

1.: NEW:

- THIS BIND VARIABLE WILL STORE NEW VALUES WHEN WE INSERT.

SYNTAX:

: NEW. <COLUMN NAME>= <VALUE>;

EX:

: NEW.SAL = 15000;

2.: OLD:

- THIS BIND VARIABLE WILL STORE OLD VALUES WHEN WE DELETE.

SYNTAX:

:OLD. <COLUMN NAME>= <VALUE>;

EX:

: OLD.SAL = 12000;

NOTE: THESE BIND VARIABLES ARE USED IN "ROW LEVEL TRIGGERS ONLY".

1. TO INVOKING USER DEFINED MESSAGE / ALERT AT EVENT FIRE.

EX:

CREATE OR REPLACE TRIGGER TRINSERT

AFTER INSERT ON TEST

BEGIN

DBMS_OUTPUT.PUT_LINE ('SOME ONE INSERTED DATA INTO YOUR TABLE');

END;

/

TRIGGER CREATED.

TESTING:

SQL> INSERT INTO TEST VALUES (105,'SCOTT',36000);

SOME ONE INSERTED DATA INTO YOUR TABLE

1 ROW CREATED.

EX:

CREATE OR REPLACE TRIGGER TRUPDATE

AFTER UPDATE ON TEST

BEGIN

**DBMS_OUTPUT.PUT_LINE ('SOME ONE UPDATED DATA INTO YOUR
TABLE');**

END;

/

TRIGGER CREATED.

TESTING:

UPDATE TEST SET SAL=22000 WHERE EID=1021;

SOME ONE UPDATING DATA IN YOUR TABLE

1 ROW UPDATED.

EX:

CREATE OR REPLACE TRIGGER TRDELETE

AFTER DELETE ON TEST

BEGIN

**DBMS_OUTPUT.PUT_LINE ('SOME ONE DELETED DATA FROM YOUR
TABLE');**

END;

/

TRIGGER CREATED.

TESTING:

DELETE FROM TEST WHERE EDI=1022;

SOME ONE DELETING DATA FROM YOUR TABLE.

1 ROW DELETED.

EX:

```
CREATE OR REPLACE TRIGGER TRDML  
AFTER INSERT OR UPDATE OR DELETE ON TEST  
BEGIN  
DBMS_OUTPUT.PUT_LINE ('SOME ONE PERFORMING DML  
OPERATIONS ON YOUR TABLE');  
END;  
/  
TRIGGER CREATED.
```

2. TO CONTROL / RESTRICTED DML OPERATIONS ON A TABLE:

EX:

```
CREATE OR REPLACE TRIGGER TRIN  
AFTER INSERT ON TEST  
BEGIN  
RAISE_APPLICATION_ERROR (-20487,'SOME ONE INSERTING  
DATA INTO YOUR TABLE');  
END;  
/
```

TESING:

INSERT INTO TEST VALUES (106,'MILLER',52000)

ERROR AT LINE 1:

ORA-20487: SOME ONE INSERTED DATA INTO UR TABLE

EX:

```
CREATE OR REPLACE TRIGGER TRUP  
AFTER UPDATE ON TEST  
BEGIN  
RAISE_APPLICATION_ERROR (-20481,'SOME ONE UPDATING  
DATA IN YOUR TABLE');  
END;  
/
```

TESTING:

UPDATE TEST SET SAL=22000 WHERE EID=1021;

ERROR AT LINE 1:

ORA-20487: SOME ONE UPDATING DATA IN YOUR TABLE

EX:

CREATE OR REPLACE TRIGGER TRDEL

AFTER DELETE ON TEST

BEGIN

**RAISE_APPLICATION_ERROR (-20481,'SOME ONE DELETING
DATA FROM YOUR TABLE');**

END;

/

TESTING:

DELETE FROM TEST WHERE EDI=1022;

ERROR AT LINE 1:

ORA-20487: SOME ONE DELETING DATA FROM YOUR TABLE.

EX:

CREATE OR REPLACE TRIGGER TRDEL

AFTER INSERT OR UPDATE OR DELETE ON TEST

BEGIN

**RAISE_APPLICATION_ERROR (-20481,'SOME ONE PERFORMING
DML OPERATIONS ON YOUR TABLE');**

END;

/

TESTING:

INSERT INTO TEST VALUES (106,'MILLER',52000);

UPDATE TEST SET SAL=22000 WHERE EID=1021;

DELETE FROM TEST WHERE EDI=1022;

ERROR AT LINE 1:

**ORA-20781: SOME ONE PERFORMING DML OPERATIONS ON YOUR
TABLE.**

3. TO IMPLEMENTING BUSINESS LOGICAL CONDITIONS:

EX:

CREATE A TRIGGER TO RESTRICTED DML OPERATIONS ON EVERY SATURDAY?

```
CREATE OR REPLACE TRIGGER TRDAY
AFTER INSERT OR UPDATE OR DELETE ON TEST
BEGIN
IF TO_CHAR(SYSDATE,'DY') = 'SAT' THEN
RAISE_APPLICATION_ERROR (-20456,'YOU CANNOT PERFORM
DML OPERATIONS ON EVERY SATURDAY');
END IF;
END;
/
```

EX:

CREATE A TRIGGER TO RESTRICTED DML OPERATIONS ON TEST TABLE IN BETWEEN 9AM TO 5PM?

```
CREATE OR REPLACE TRIGGER TRTIME
AFTER INSERT OR UPDATE OR DELETE ON TEST
BEGIN
IF TO_CHAR(SYSDATE,'HH24') BETWEEN 9 AND 16 THEN
RAISE_APPLICATION_ERROR (-20456,'YOU CANNOT PERFORM
DML OPERATIONS BETWEEN 9AM TO 5PM');
END IF;
END;
/
```

EX:

TRIGGER NAME: HOLI_TRIG

TABLE NAME: EMP

TRIGGER EVENT: BEFORE INSERT OR UPDATE OR DELETE

SOL:

CREATE OR REPLACE TRIGGER HOLI_TRIG

BEFORE INSERT OR UPDATE OR DELETE

ON EMP

DECLARE

CNT NUMBER;

BEGIN

IF TO_CHAR(SYSDATE,'HH24') NOT BETWEEN 10 AND 16 THEN

**RAISE_APPLICATION_ERROR (-20001,'OFFTIMINGS, TRANS.
ARE NOT ALLOWED.');**

END IF;

IF TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN') THEN

**RAISE_APPLICATION_ERROR (-20002,'WEEKENDS, TRANS.
ARE NOT ALLOWED.');**

END IF;

SELECT COUNT(HDATE) INTO CNT FROM HOLIDAY

**WHERE TO_CHAR(SYSDATE,'DD/MM/YY')
=TO_CHAR(HDATE,'DD/MM/YY');**

IF CNT>0 THEN

**RAISE_APPLICATION_ERROR (-20003,'TODAY PUBLIC
HOLIDAY, TRANS. ARE NOT ALLOWED.');**

END IF;

END;

4. TO VALIDATING DATA:

EX:

CREATE A TRIGGER TO VALIDATE INSERT OPERATION IF NEW SALARY IS LESS THAN 5000?

SQL> CREATE OR REPLACE TRIGGER TRSAL1

BEFORE INSERT ON TEST

FOR EACH ROW

BEGIN

IF: NEW.SAL < 5000 THEN

RAISE_APPLICATION_ERROR (-20348,'NEW SAL SHOULD NOT BE LESS THAN TO 5000');

END IF;

END;

/

TESTING:

INSERT INTO TEST VALUES (1021,'SMITH',4500); -----NOT ALLOWED

INSERT INTO TEST VALUES (1021,'SMITH',5500); -----ALLOWED

EX:

CREATE OR REPLACE TRIGGER DEPT_TRIG

BEFORE INSERT ON DEPT

FOR EACH ROW

BEGIN

: NEW.DNAME := UPPER (: NEW.DNAME);

: NEW.LOC := UPPER (: NEW.LOC);

END;

TESTING:

SQL> INSERT INTO DEPT VALUES (50,'ECONOMICS','HYD');

EX:

CREATE A TRIGGER TO VALIDATE UPDATE OPERATION ON TEST TABLE IF NEW SALARY IS LESS THAN TO OLD SALARY?

SQL> CREATE OR REPLACE TRIGGER TRSAL2

BEFORE UPDATE ON TEST

FOR EACH ROW

BEGIN

IF: NEW.SAL <: OLD.SAL THEN

RAISE_APPLICATION_ERROR (-20748,'NEW SAL SHOULD NOT BE LESS THAN TO OLD SALARY');

END IF;

END;

/

TESTING:

UPDATE TEST SET SAL=6000 WHERE SAL=8000; -----NOT ALLOWED

UPDATE TEST SET SAL=9000 WHERE SAL=8000; ----- ALLOWED

EX:

CREATE A TRIGGER TO VALIDATE DELETE OPERATION ON TEST TABLE IF WE TRY TO DELETE THE EMPLOYEE "SMITH" DETAILS?

SQL> CREATE OR REPLACE TRIGGER TRDATA

BEFORE DELETE ON TEST

FOR EACH ROW

BEGIN

IF: OLD.ENAME = 'SMITH' THEN

RAISE_APPLICATION_ERROR (-20648,'WE CANNOT DELETE SMITH EMPLOYEE DETAILS');

END IF;

END;

/

TESTING:

DELETE FROM TEST WHERE ENAME='SMITH'; -----NOT ALLOWED

DELETE FROM TEST WHERE ENAME='ALLEN'; ----ALLOWED

5. FOR AUDITING:

- WHEN WE MANIPULATE DATA IN A TABLE THOSE TRANSACTIONAL VALUES ARE STORED IN ANOTHER TABLE IS CALLED AS AUDITING TABLE.

EX:

SQL> CREATE TABLE EMP1(EID INT, ENAME VARCHAR2(10), SAL NUMBER (10));

TABLE CREATED.

SQL> CREATE TABLE AUDITEMP1(EID INT, AUDIT_INFOR VARCHAR2(100));

TABLE CREATED.

EX:

CREATE OR REPLACE TRIGGER TRAUDIT_INSERT

AFTER INSERT ON EMP1

FOR EACH ROW

BEGIN

**INSERT INTO AUDITEMP1 VALUES (: NEW.EID,'SOME ONE
INSERTED A NEW ROW INTO EMP1 TABLE ON'||' '||**

TO_CHAR (SYSDATE,'DD-MON-YYYY HH:MI: SS AM'));

END;

/

TESTING:

INSERT INTO EMP1 VALUES (1021,'SMITH',4500);

EX:

```
CREATE OR REPLACE TRIGGER TRAUDIT_UPDATE
AFTER UPDATE ON EMP1
FOR EACH ROW
BEGIN
INSERT INTO AUDITEMP1 VALUES (:OLD.EID,'SOME ONE
UPDATED DATA IN EMP1 TABLE ON'||' '||
TO_CHAR(SYSDATE,'DD-MON-YYYY HH:MI: SS AM'));
END;
/
```

TESTING:

```
UPDATE EMP1 SET SAL=6000 WHERE EID=1021;
```

EX:

```
CREATE OR REPLACE TRIGGER TRAUDIT_DELETE
AFTER DELETE ON EMP1
FOR EACH ROW
BEGIN
INSERT INTO AUDITEMP1 VALUES (:OLD.EID,'SOME ONE
DELETED DATA FROM EMP1 TABLE ON'||' '||
TO_CHAR(SYSDATE,'DD-MON-YYYY HH:MI: SS AM'));
END;
/
```

TESTING:

```
DELETE FROM EMP1 WHERE ENAME='SMITH';
```

EX:

```
CREATE OR REPLACE TRIGGER TRAUDIT_DML
AFTER INSERT OR UPDATE OR DELETE ON EMP1
FOR EACH ROW
BEGIN
INSERT INTO AUDITEMP1 VALUES (: OLD.EID,'SOME ONE
PERFORMING DML OPERATIONS ON EMP1 TABLE ON'||' '||
TO_CHAR (SYSDATE,'DD-MON-YYYY HH:MI: SS AM'));
END;
/
```

TESTING:

```
INSERT INTO EMP1 VALUES (1021,'SMITH',4500);
UPDATE EMP1 SET SAL=6000 WHERE EID=1021;
DELETE FROM EMP1 WHERE ENAME='SMITH';
```

DDL TRIGGERS / DB TRIGGERS:

- THESE TRIGGERS ARE EXECUTED BY SYSTEM AUTOMATICALLY WHEN USER PERFORM DDL (CREATE / ALTER / DROP / RENAME) OPERATIONS ON A SPECIFIC SCHEMA / DATABASE.

- THESE TRIGGERS ARE HANDLED BY DBA ONLY.

SYNTAX:

```
CREATE OR REPLACE TRIGGER <TRIGGER_NAME>
BEFORE/AFTER CREATE OR ALTER OR DROP OR RENAME
ON USERNAME.SCHEMA
[ FOR EACH ROW]
[ DECLARE
  <VARIABLE DECLARATION>]
BEGIN
  <EXEC STATEMENTS>;
END;
```

EX:

```
SQL> CREATE OR REPLACE TRIGGER TRDDL  
      AFTER CREATE ON MYDB9AM.SCHEMA  
      BEGIN  
        RAISE_APPLICATION_ERROR (-20456,'WE CANNOT CREATE A  
        NEW TABLE IN MYDB9AM SCHEMA');  
      END;
```

TRIGGER CREATED.

```
SQL> CREATE TABLE T1(SNO INT);
```

ERROR AT LINE 1:

**ORA-20456: WE CANNOT CREATE A NEW TABLE IN MYDB9AM
SCHEMA**

EX:

```
SQL> CREATE OR REPLACE TRIGGER TRDDL
```

```
      AFTER ALTER ON MYDB9AM.SCHEMA  
      BEGIN
```

```
        RAISE_APPLICATION_ERROR (-20456,'WE CANNOT ALTER A  
        TABLE IN MYDB9AM SCHEMA');
```

```
      END;
```

```
/  
SQL> ALTER TABLE EMP1 ADD EADD VARCHAR2(10);
```

ORA-20456: WE CANNOT ALTER TABLE IN MYDB9AM SCHEMA

```
SQL> CREATE OR REPLACE TRIGGER TRDDL
```

```
      AFTER DROP ON MYDB9AM.SCHEMA  
      BEGIN
```

```
        RAISE_APPLICATION_ERROR (-20456,'WE CANNOT DROP A  
        TABLE FROM MYDB9AM SCHEMA');
```

```
      END;
```

```
/
```

TRIGGER CREATED.

```
SQL> DROP TABLE EMP1 PURGE;
ORA-20456: WE CANNOT DROP A TABLE FROM MYDB9AM SCHEMA
```

```
SQL> CREATE OR REPLACE TRIGGER TRDDL
AFTER RENAME ON MYDB9AM.SCHEMA
BEGIN
RAISE_APPLICATION_ERROR (-20456,'WE CANNOT RENAME A
TABLE IN MYDB9AM SCHEMA');
END;
/
TRIGGER CREATED.
```

```
SQL> RENAME EMP1 TO EMPDETAILS;
ORA-20456: WE CANNOT RENAME A TABLE IN MYDB9AM SCHEMA
ORA-06512: AT LINE 2
```

EX:

```
SQL> CREATE OR REPLACE TRIGGER TRDDL
AFTER CREATE OR ALTER OR RENAME OR DROP ON
MYDB9AM.SCHEMA
BEGIN
RAISE_APPLICATION_ERROR (-20456,'WE CANNOT PERFORM
DDL OPERATIONS ON MYDB9AM SCHEMA');
END;
/
TRIGGER CREATED.
```

EX:

**CREATE A TRIGGER TO RESTRICTED DDL OPERATIONS ON
MYDB9AM SCHEMA IN BETWEEN 9AM TO 5PM?**

CREATE OR REPLACE TRIGGER TRDDLTIME

**AFTER CREATE OR ALTER OR RENAME OR DROP ON
MYDB9AM.SCHEMA**

BEGIN

**IF TO_CHAR(SYSDATE,'HH24') BETWEEN 9 AND 16 THEN
RAISE_APPLICATION_ERROR (-20456,'YOU CANNOT PERFORM
DDL OPERATIONS ON MYDB9AM BETWEEN 9AM TO 5PM');**

END IF;

END;

/

DROPPING TRIGGERS:

SQL> DROP TRIGGER <TRIGGERNAME>;

EX:

DROP TRIGGER TRDDLTIME;

