
ORACLE

INDEX

Oracle 19c ----------

Oracle 19c

Introduction of DBMC

- DBMC stands for Database management system.

Data:

- It is a raw fact (unprocessed) i.e. number, character, special character, etc.
- Never provide meaningful statement/ information.

Ex:

SMITH 10021
MILLER 10022

Information:

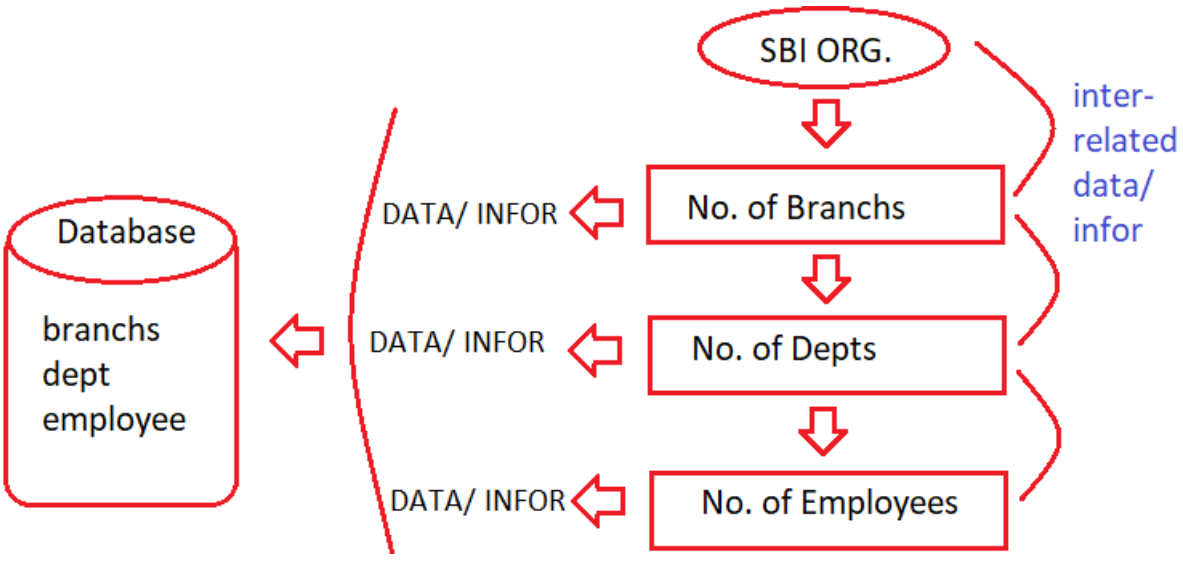
- Processed data is called as information.
- Information is always providing accurate & meaningful statement to user.

Ex:

empid	ename	salary	doj	deptname
10021	SMITH	25000	05-08-2020	DB
10022	MILLER	45000	24-12-2029	HR

Database:

- It is collection of inter-related information. Which has been organized in proper systematically to in a memory is called as Database.
- By using database we can store, modify, select and delete data from database with security manner.



Inter-related information Ex:

SBI Organization

- Group of Branches
  - Group of Departments
    - Group of Employees

Types databases:

- There are two types of databases in the Real world.
  a. OLTP (Online Transaction Processing)
  b. OLAP (Online Analytical Processing)

OLTP (Online Transaction Processing):

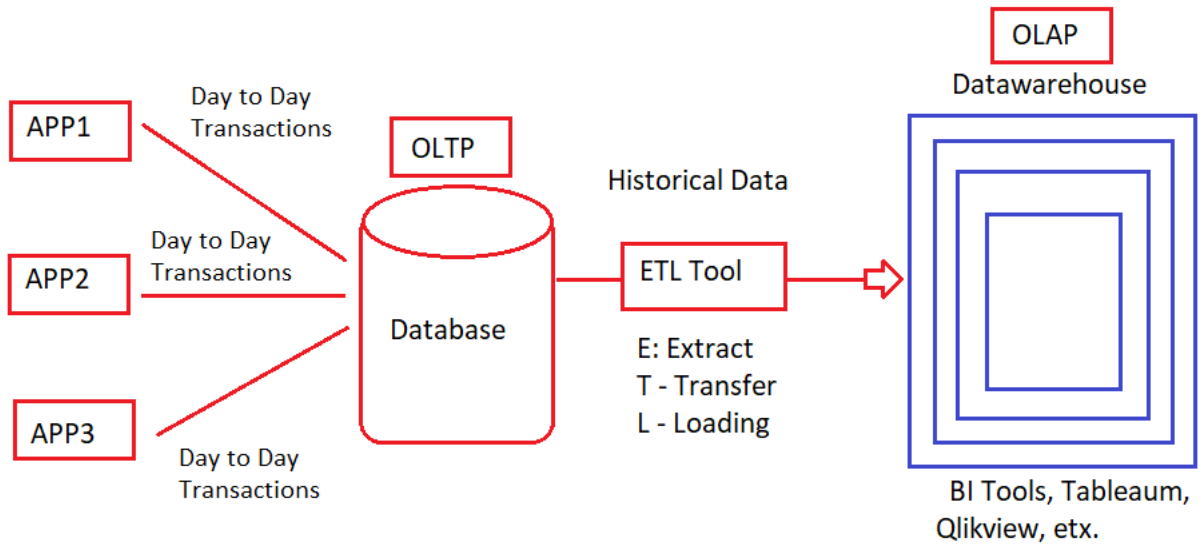
- Organizations are maintained OLTP to storing “day - to – day” transactions data/ information of a particular running business.
- Live

Ex: SqlServer, Oracle, MySQL, etc.

OLAP (Online Analytical Processing):

- Used for data analysis (or) data summarized (or) history of data of particular business (Big data).

Ex: Datawarehouse.



Data storages:

- It a location where we can store data/ information. We have different types of data storages.

- Books & papers
- Flat file (file management system)
- DBMS/ Database

Disadvantages of books & papers:

- It is complete manual process/ system
- Required more man power
- Costly in maintenance
- There is no security
- Handling a very small data
- Retrieving data will be time consume.

Flat file (file management system):

- In file management data can be stored in files.

Disadvantages:

- Data redundancy & data inconsistency: These problems come 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 number 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.
- Data integrity: This is about maintain proper data in every organization impose set integrity rules on data and we will call these rules are business rules. Database provides an option for imposing the business rules with the help of constraints and triggers.
- 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. Whereas if you want to retrieve data from database then we are using sql language.
- Data security: Data is never secure under books and flat file whereas database is 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.
- Data indexing: Indexes are using for accessing data much faster but flat files does not provide any index mechanism whereas database will provide indexing mechanism.

DBMS:

- Stands for Database management system.

- It is a software which is used to manage data in database. Like,
    - to connect
    - to create table
    - to insert data
    - to update data
    - to select data
    - to delete data

Advantages of DBMS:

- To reduce data redundancy
- To avoid data inconsistency. Problem
- Easy to access/ retrieve data
- Easy to manipulate data
- More security (authentication & authorization)
- It supports data integrity rules
- Supporting data sharing
- Supports transactions with “ACID” properties.

DBMS/ Database Models

- How data can be organized/ store in different database models. Ther 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 existence 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 relations every child is having only one parent. This tree is containing the following three level root, parent and child level.

Ex: IMS software (Information Management System)

Disadvantages:

- 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.

- 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.
- This model designed based on one-many relations i.e. ever child is having only one parent so that there is a chance to occurs data duplicate.

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).

Advantages of NDBMS:

- To reduce duplicate data because supporting man-to-many relation (a child can have multiple parents).
- By using pointers mechanism, we can add new level (parent/ child) to an existing structure without reconstruction.
- Accessing data in this model is very fast because it uses pointers.

Disadvantages of NDBMS:

- 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.
- 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. EF. Codd from IBM came with a new concept on storing data in a table structure i.e. rows and columns format.
- EF. 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 corporation) 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 data redundancy & data inconsistency 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 data integrity rules with constraints & triggers
- Supporting SQL & PL/SQL languages.

Object Relational DBMS (ORDBMS):

- These databases are depending on “SQL”.
- Data can be stored in the form of “table” (i.e. rows X columns).
- Column is called as “attribute/ field” & Row is called as “record/ tuple”.
- Here database is a collection of tables. A table is a collection of rows and columns. A row is a group of columns.

Ex: Oracle, SqlServer, MySQL, etc.

Object Oriented Relational DBMS (OORDBMS):

- Also called as “No-SQL” database.
- It does not depend on “SQL” language.
- It’s completely depending on “OOPS”.
- Data can be stored in form of “Object”.

Ex: MongoDB, Cassandra, etc.

Introduction to Oracle

- It is a RDBMS (ORDBMS) from oracle corporation in 1979. It is used to store data (or) information permanently i.e., in hard disk along with

security.

- Oracle is a platform independent RDBMS product. It means that it can deployed (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.
    - Platform independent: It supports any OS with the combination of any microprocessor.
      Ex: Oracle, MySQL, java, .net, etc.

    - Platform dependent: It supports only one OS with combination of any microprocessor.
      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 potable 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 datatype 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 datatype, using sequence, enable and disable triggers
2013	Oracle 12c (cloud technology)	Truncate table, cascade, multiple indexes, invisible column, sequence

		session, new auto increment by using identity
2018	Oracle 18c	Polymorphic table functions, Active directory integration
2019	Oracle 19c	Active data guard DML redirection, Automatic index creation, SQL queries on object stores

Note:

- ✓ Oracle Software is available in two editions in the market.
  - Oracle Expression Edition
    - Supporting partial features
    - recyclebin, flashback, purge, partition table etc. not supporting
  - Oracle Enterprise Edition
    - Supporting all features.

Working with oracle:

- Once we installed Oracle software internally there are two components are installed in system those are
  1. Oracle client
  2. Oracle server

Oracle client:

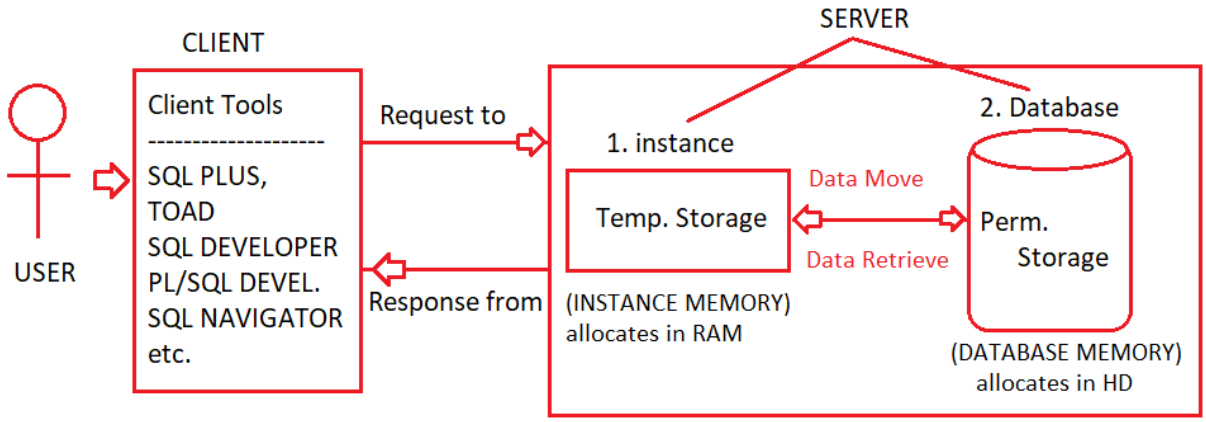
- By using oracle client tool user can perform the following three operations are
  - user can connect to Oracle DB server
  - user can send request to Oracle DB server
  - user can receive response from Oracle DB server

Ex: SQL plus, toad, Sql developer, Sql navigator, etc.

Oracle server:

- Oracle server manage two more sub component Internally those are,
  - Instance:
    - It will act as temporary memory which is allocating from RAM.
    - It stores data in temporary manner.
  - Database
    - Permanent memory which is allocating from hard disk.

- It stores data permanently.

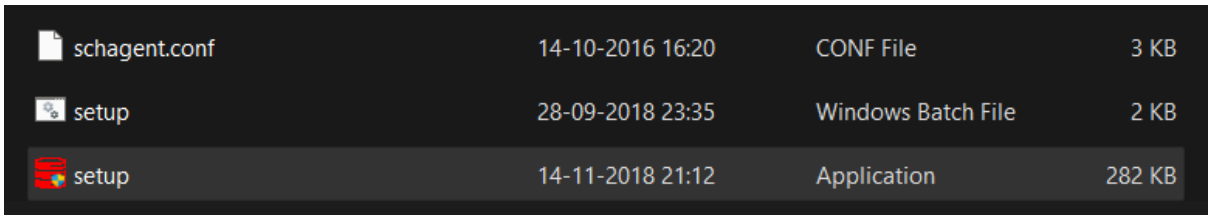


[Dig: Client- Server Architecture]

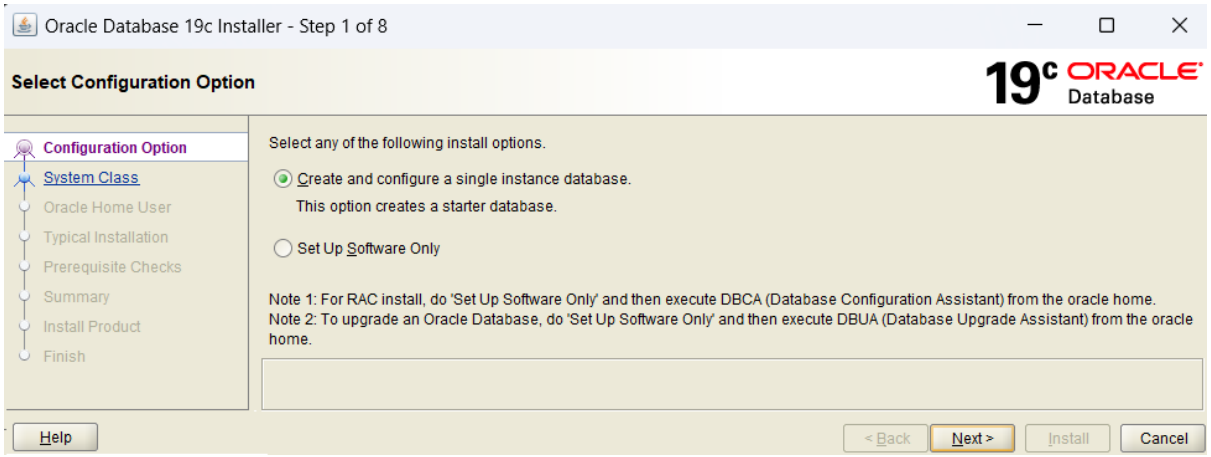
How to download and install Oracle 19C

Step 1: Go to the below link and download the Oracle 19C, by login into your oracle account if you don't have just create it [Oracle 19c Enterprise Edition].

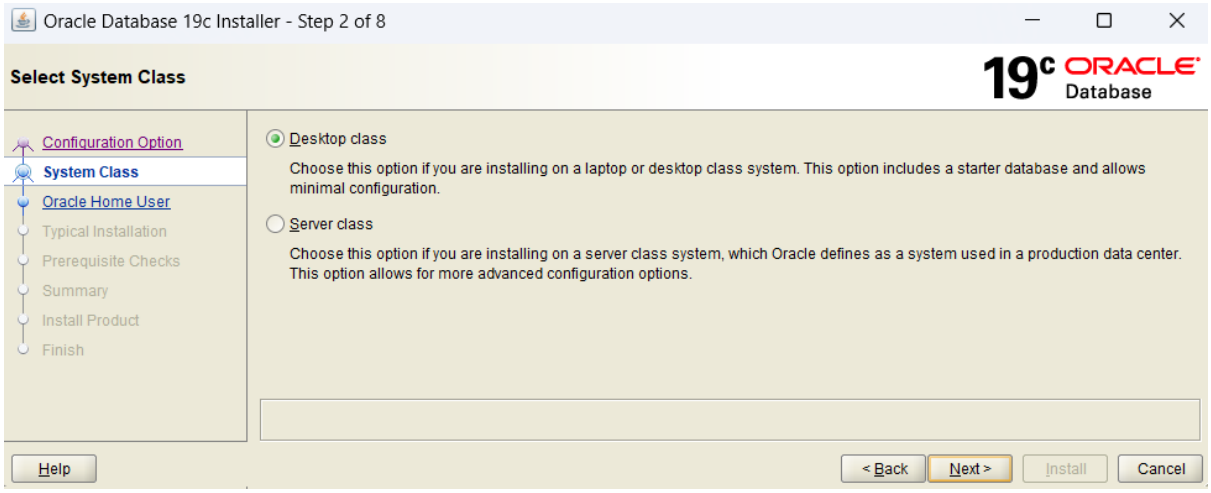
Step 2: After download, extract the ZIP file and double click on **setup.exe** file.



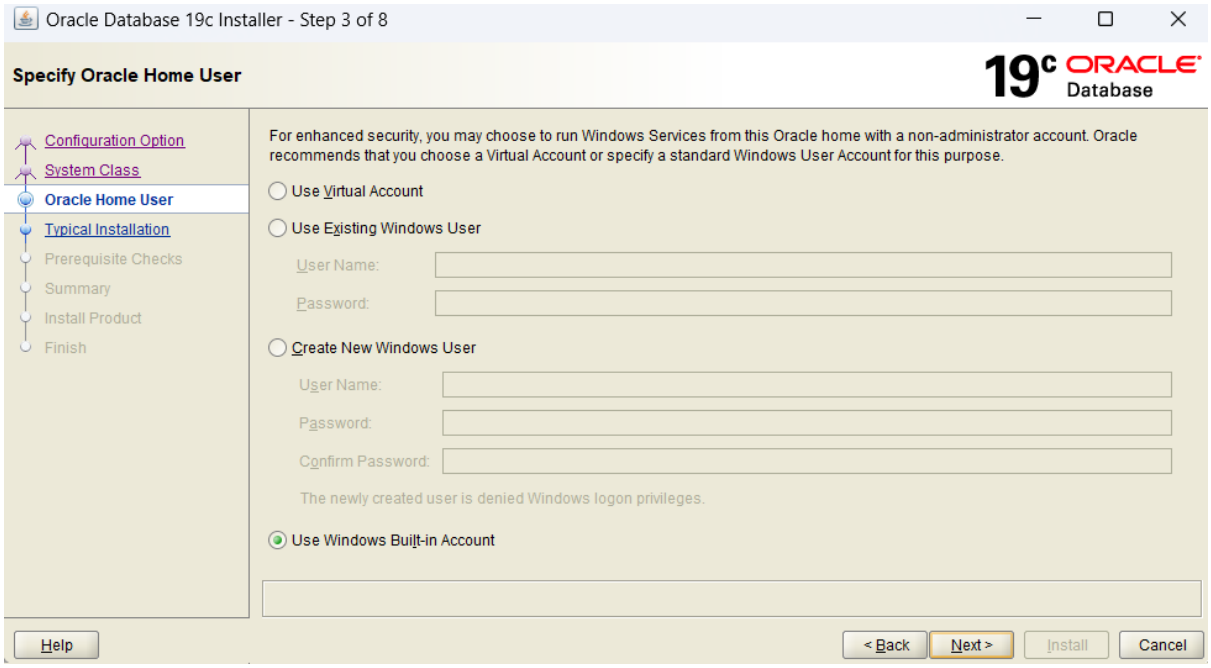
Step 3: Then the below screen will appear make sure **Create and configure a single instance database** option selected then click on **Next** button.



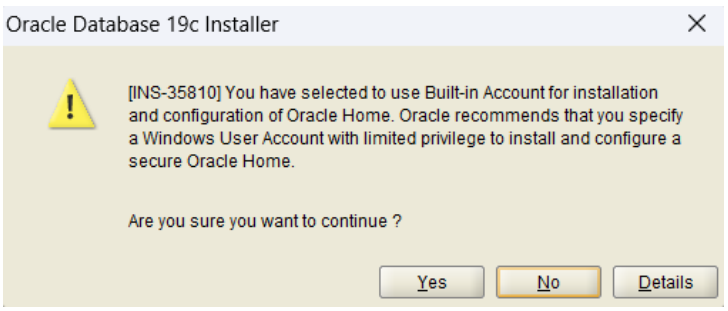
Step 4: Then choose **Desktop class** option then click on **Next** button.



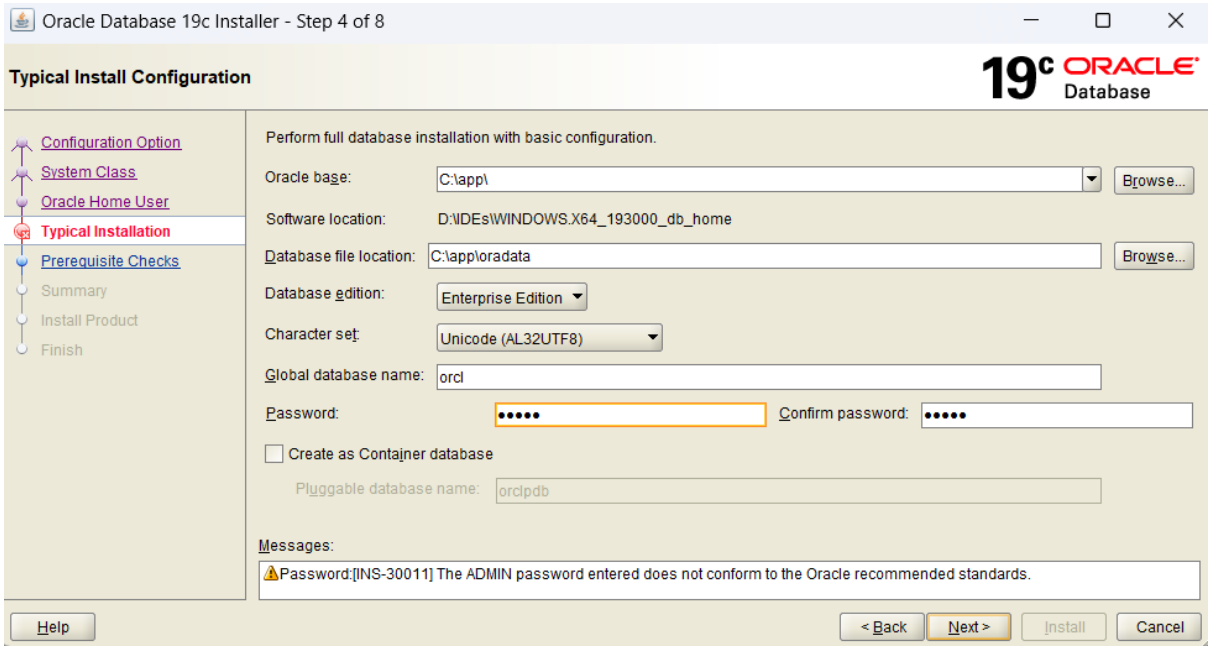
Step 5: After that select **Use windows Built-in Account** then click on **Next** button.



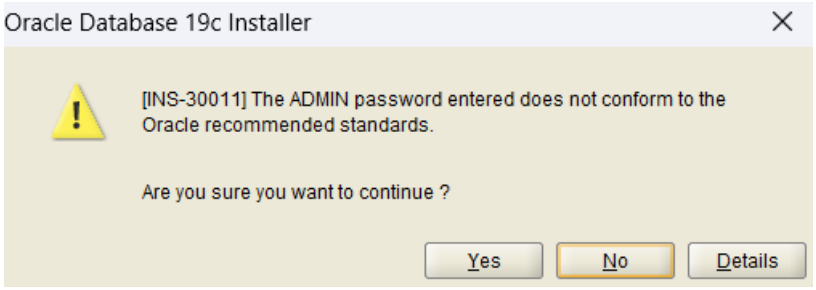
Step 6: It will give a warning just click on **Yes** button.



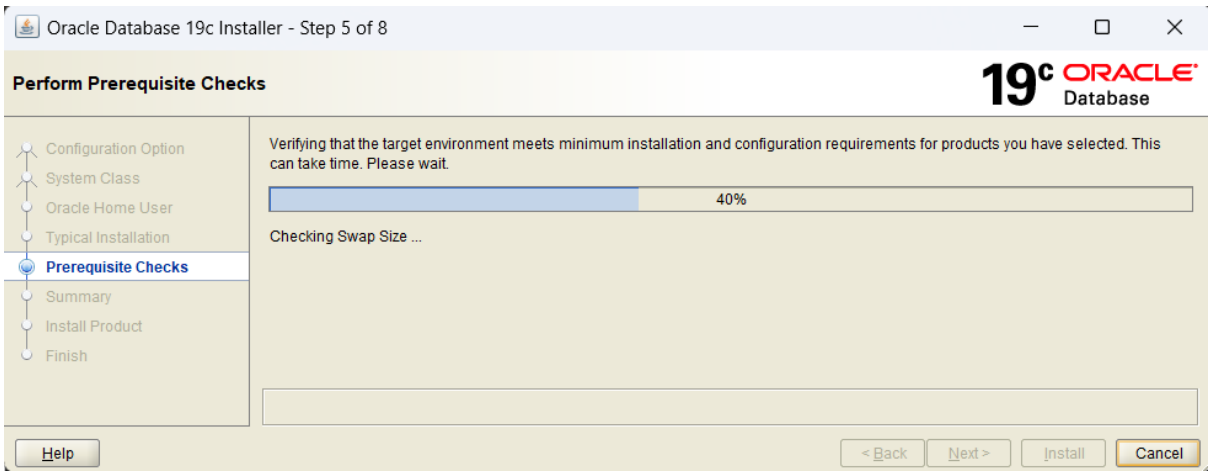
Step 7: Give **Oracle base** then enter the **Password** & **Confirm password** make sure to uncheck **Create as Container database** then click on **Next**.



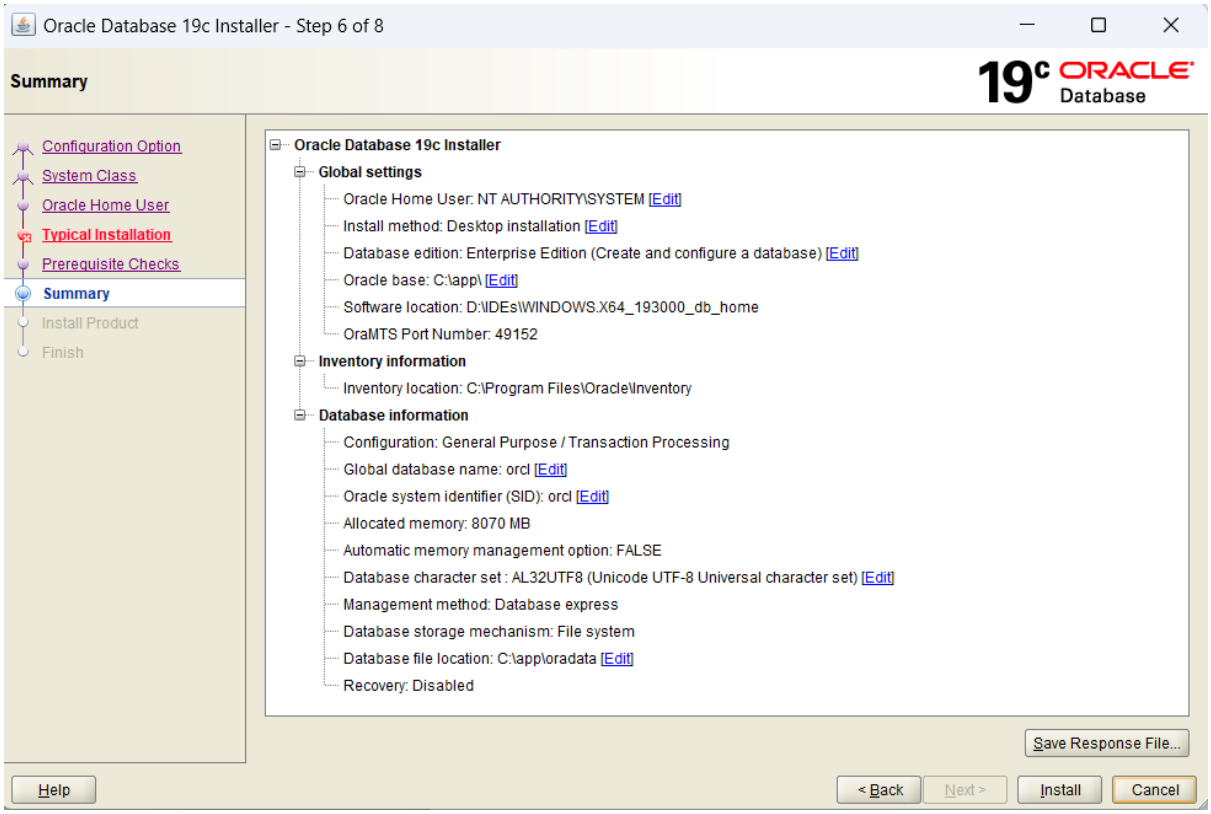
Step 8: Again, we will get a warning just click on **Yes** button.



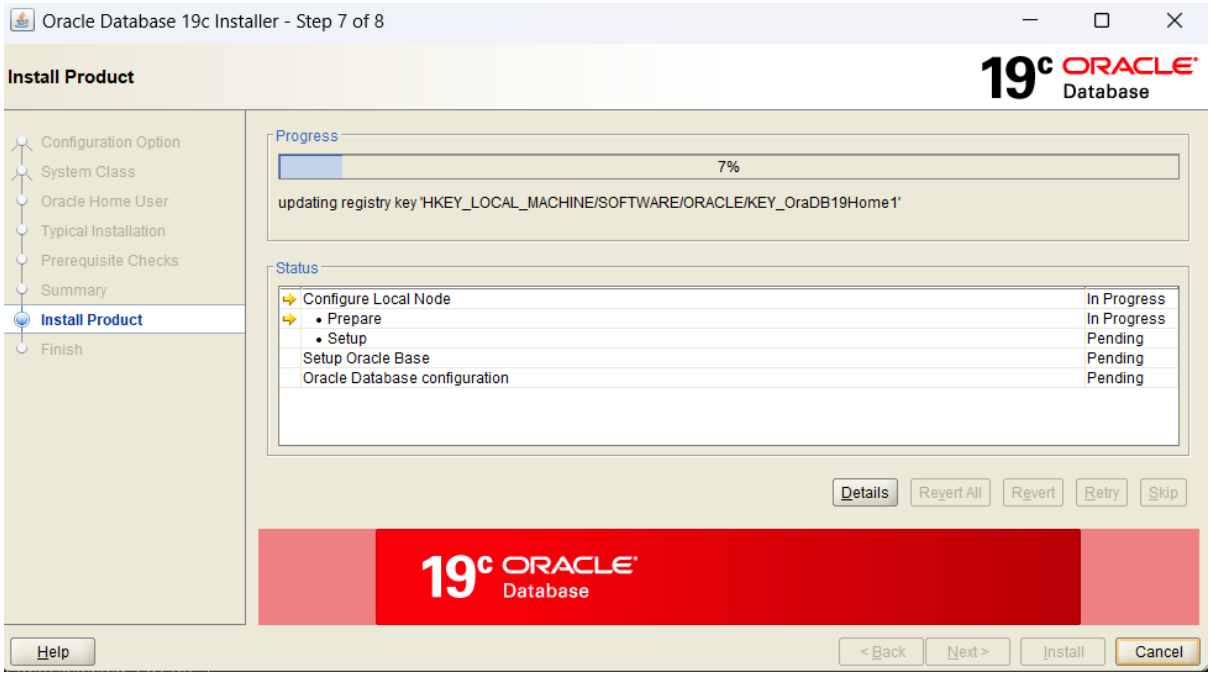
Step 9: Then it will take little process



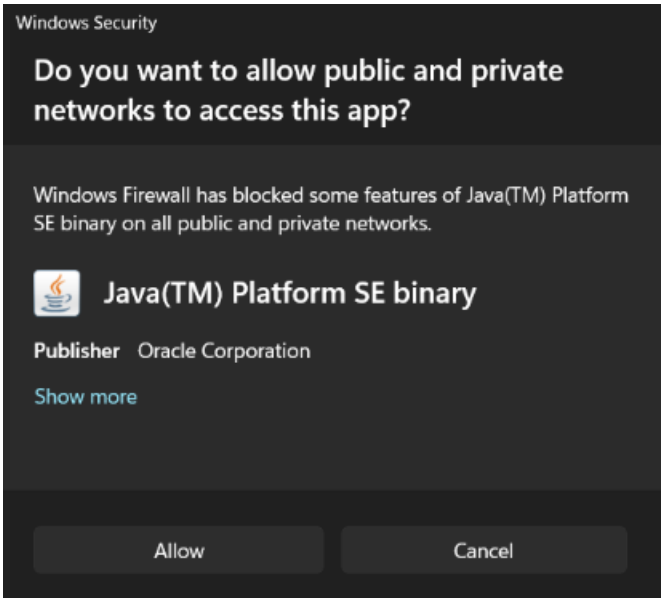
Step 10: Then a confirmation screen will con then click on **Install** button.



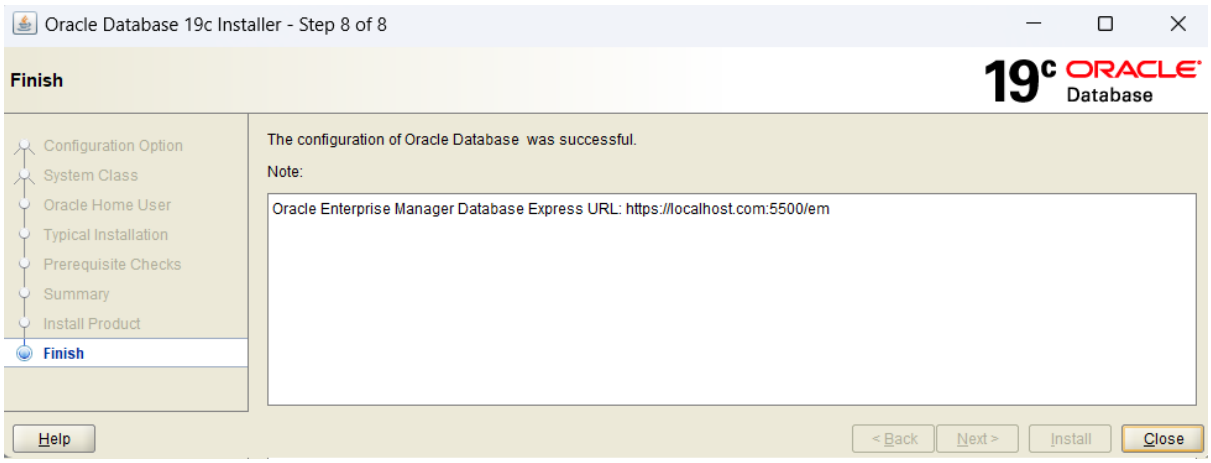
Step 11: Now the installation will start and it take some time have some patience.



While install, will ask to allow the private access click on **allow** button.



Step 12: After installation we will get successful screen as like below the click on **Close** button.

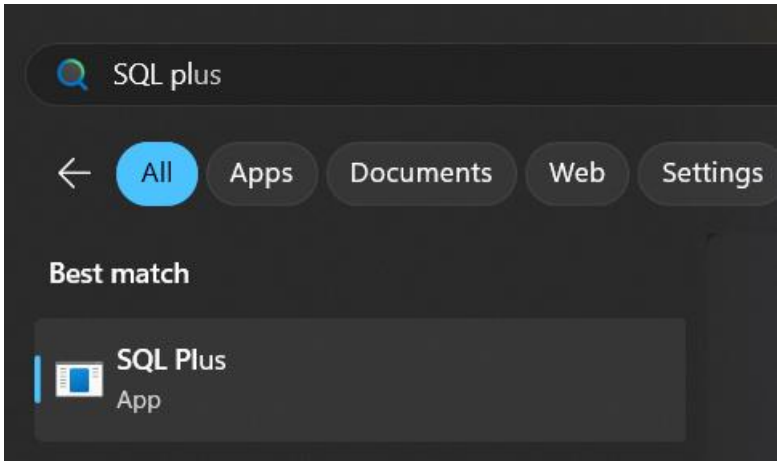


Note: When we want to work on oracle database then we have to follow the following two steps procedure

1. Connect to oracle: If user wants to connect to oracle, then we required a database tool is called as “SQL plus”, which was inbuilt 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”.

Steps to connect to oracle:

Step 1: Go to Window search and type **SQL Plus** then the SQL plus app will appear click on that.



Step 2: Then give the username and password (at installation time password) then click on enter.

```
SQL*Plus: Release 19.0.0.0.0 - Production on Sun Mar 3 01:15:38 2024
Version 19.3.0.0.0

Copyright (c) 1982, 2019, Oracle.  All rights reserved.

Enter user-name: system
Enter password:

Connected to:
Oracle Database 19c Enterprise Edition Release 19.0.0.0.0 -
Production
Version 19.3.0.0.0
```

To create a new username & password in Oracle DB

Syntax:

SQL> CREATE USER <username> IDENTIFIED BY <password>;

```
SQL> conn
Enter user-name: system / tiger
Connected.
SQL> CREATE USER nirmala IDENTIFIED BY nirmala;

User created.
```

Note:

- ✓ User is created but this user is dummy user (i.e. no privileges), it is not having permission to connect and create new table in DB. So, permissions must be given to user by using "**grant**" command by DBA i.e. system for connect to Oracle DB server.

- ✓ Every user in oracle server is called as “schema”. Granting permissions to user.

```
SQL> CONN
Enter user-name: nirmala / nirmala
ERROR:
ORA-01045: user NIRMALA lacks CREATE SESSION privilege; logon denied

Warning: You are no longer connected to ORACLE.
```

Run the following command to give the permission to the new user

```
SQL> CONN
Enter user-name: system / tiger
Connected.
SQL> GRANT CONNECT TO nirmala;

Grant succeeded.

SQL> CONN
Enter user-name: nirmala / nirmala
Connected.
```

How to change a password:

Step 1: Connect to that user which password you want to change.

```
SQL> CONN
Enter user-name: nirmala / nirmala
Connected.
```

Step 2: Type **PASSWORD** and hit enter then give old password, new password and retype new password

```
SQL> PASSWORD
Changing password for NIRMALA
Old password:
New password:
Retype new password:
Password changed
```

Step 3: Now you can connect to that particular user again.

```
SQL> conn
Enter user-name: nirmala / tiger
Connected
```

How to create a new password if we forgot:

Syntax:

SQL> ALTER USER <user name> IDENTIFIED BY <new password>;

Step 1: Connect with administrator user.

```
SQL> CONN
Enter user-name: nirmala / nirmala
Connected.
```

Step 2: Now alter the user with new password.

```
SQL> ALTER USRE nirmala IDENTIFIED BY nirmala

User altered.
```

How to know username if we forgot:

Syntax:

SQL> SELECT username FROM all_users;

```
SQL> conn
Enter user-name: system / tiger
Connected.
SQL> SELECT username FROM all_users;

USERNAME
------------------------------------------------------------
OLAPSYS
DVSYS
LBACSYS
DVF
HR
NIRMALA
```

How to drop a user:

Syntax:

SQL> DROP USER <username> CASCADE;

```
SQL> conn
Enter user-name: system/tiger
Connected.
SQL> DROP USER nirmala CASCADE;

User dropped.
```

Note: When we want to connect to Oracle DB server sometimes, we faced another problem is called as “TNS:protocol adapter error”.

```
SQL> conn
Enter user-name: system/tiger
ERROR:
ORA-03113: end-of-file on communication channel
Process ID: 0
Session ID: 128 Serial number: 18562

ERROR:
ORA-12560: TNS:protocol adapter error

Warning: You are no longer connected to ORACLE.
```

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, now you can connect.

```
SQL> conn
Enter user-name: nirmala / tiger
Connected.
```

Note: When we connect to Oracle DB sometimes, 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:
Step2: To unlock user:
Syntax:
SQL> ALTER USER <user name> ACCOUNT UNLOCK / LOCK;
Step3: Now you can connect to oracle with the user that locked.

Datatypes in Oracle

- Datatype is an attribute which is used to specifies what type of data is stored into a column.
- Oracle supports the following datatypes are
  - Numeric datatypes

- Character datatypes/ String datatypes
- Long datatype
- Date datatypes
- Raw & long raw datatypes
- LOB datatypes (large objects datatypes)

Numeric datatypes:

1. int
2. number (p, s)

int:

- Storing integer format values only.
- int ---> convert ---> 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): then store integer values
- When we use number (p, s): then store float values

Precision (p): counting all digits including left & right sides of given float expression.

Scale (s): counting only right digits of a float expression.

Ex: 56.23 [p: 4 & s: 2]

Character datatypes:

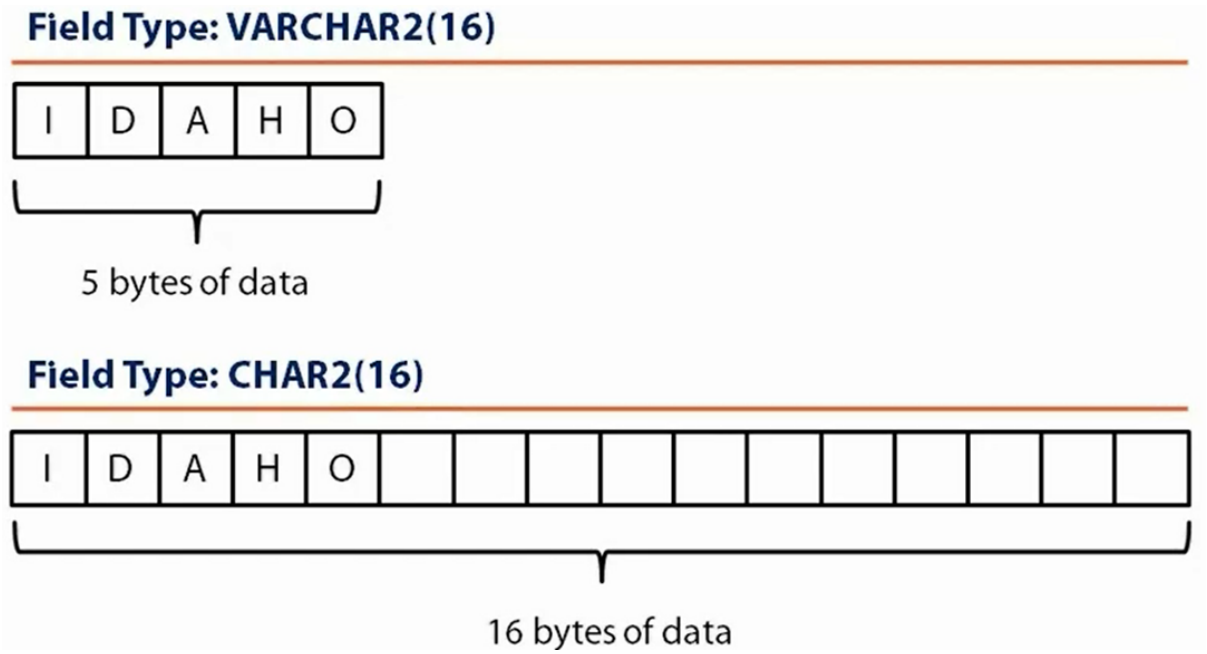
- Storing "string" format data only.
- In database string is represent with single quotes ' <string> ‘.
- Character datatypes are storing two types string format data, those are
  - Characters only string data [A-Z, a-z]
  - Alphanumeric char’s string data [A-z, a-z, 0-9, @, #, $, %, _ … etc.]

Note: Character datatypes are again classified into two categories those are

a. Non-Unicode datatypes
b. Unicode datatypes

Non-Unicode datatypes:

- Supporting to store localized data (only English language). These are again two types.
    - char (size):
        - It is a fixed length datatype (static).
        - It will store non-Unicode chars in the form of 1 char = 1 byte.
        - Maximum size of char datatype is 2000 bytes (2000 chars).
        - Biggest problem is memory wasted.



    - varchar2 (size):
        - It is a variable length datatype (dynamic).
        - It will store non-Unicode chars in the form of 1 char = 1 byte.
        - Maximum size of varchar2 datatype is 4000 bytes (4000 chars).
        - Memory saved.
          etc.

Unicode datatypes:

- These datatypes are storing "globalized data", supporting "all national languages". These are two types.

- nchar (size):
  - It is fixed length datatype (static).
  - It Will store Unicode char's (all national language's) in the form of 1 char = 1 byte.
  - Maximum size of nchar datatype is 2000 bytes (2000 chars).
  - Biggest problem is memory wasted.

- nvarchar2 (size):
  - It is a variable length datatype (dynamic).
  - It will store Unicode chars in the form of 1 char = 1 byte.
  - Maximum size of nvarchar2 datatype is 4000 bytes (4000 chars).
  - Memory saved.

Long datatypes:

- It is variable length datatype (dynamic).
- It will store Non-Unicode & Unicode chars in the form of 1 char = 1byte.
- Maximum size of long datatype is 2GB.

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".
  - Date:
    - Storing date & time information but time is optional.
    - If user not given time information, then by default Oracle will take time is "00:00:00 am".
    - Default format is "dd-mon-yyyy/yy hh:mi:ss".
    - It occupied 7 bytes of memory (fixed memory).

  - Timestamp:
    - Storing date & time information along with milliseconds.
    - Default format is 'dd-mon-yyyy/yy hh:mi:ss.ms'.
    - 1 second is 1000 milliseconds.
    - It occupied 11 bytes of memory (fixed memory).

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.

LOB datatypes:

- LOB stands for Large Objects.
- There are three types of LOBS,
  - CLOB (Character Large Object):
    - Stores non-Unicode characters.
    - The maximum size is 4 GB (dynamic).
  - NCLOB (National Characters Large Object):
    - Storing Unicode Characters.
    - The maximum size is 4 GB (dynamic).
  - BLOB (Binary Large Object):
    - Storing image/ audio/ video files in the form 010010101001 binary format.
    - The maximum size is 4 GB (dynamic).

Note:

- ✓ Non-Unicode characters
  - char is up to 2000 bytes
  - varchar2 is up to 4000 bytes
  - long up to 2 GB
  - CLOB up to 4 GB
- ✓ Unicode characters
  - Nchar is up to 2000 bytes
  - Nvarchar2 is up to 4000 bytes
  - long up to 2 GB
  - NCLOB up to 4 GB
- ✓ Binary datatypes
  - raw is up to 2000 bytes
  - long raw is up to 2 G
  - BLOB is up to 4 GB

SQL

- SQL stands for Structure query language.
- SQL is a database language which was used to communicate with database.
- Introduced by “IBM” and initial name of this language was "SEQUEL" (Structure English Query Language) 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 ";".
- SQL is having sub languages, those are
    a. DDL (Data Definition Language):
        - Create
        - Alter
        - Rename
        - Truncate
        - Drop

      New Commands
        - Recyclebin
        - Flashback
        - Purge

    b. DML (Data Manipulation Language):
        - Insert
        - Update
        - Delete

      New commands
        - Insert all
        - Merge

    c. DQL/ DRL (Data Query/ Data Retrieval Language):
        - Select

    d. TCL (Transaction Control Language):
        - Commit
        - Rollback
        - Savepoint

    e. DCL (Data Control Language):
        - Grant
        - Revoke

DDL (Data Definition Language)

Create:

- To create a new table/ object database (i.e. table, view, synonym, procedure, function, etc.).

Syntax:

SQL> CREATE TABLE <table name> (<column name1> <datatype>[size], <column name2> <datatype>[size], ……);

```
SQL> CONN
Enter user-name: nirmala/nirmala
Connected.

SQL> CREATE TABLE STUDENT (ID INT, SNAME CHAR(20), FEE NUMBER(6,2));
CREATE TABLE STUDENT (ID INT, SNAME CHAR(20), FEE NUMBER(6,2))
*
ERROR at line 1:
ORA-01031: insufficient privileges

SQL> CONN
Enter user-name: system/tiger
Connected.

SQL> GRANT CREATE TABLE TO nirmala;

Grant succeeded.

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

SQL> CREATE TABLE STUDENT (ID INT, SNAME CHAR(20), FEE NUMBER(6,2));

Table created.
```

Note: Make sure we have to give all privileges to the new user.

To view list of tables in Oracle DB:

Syntax:

SQL> SELECT * FROM TAB;

```
SQL> SELECT * FROM TAB;
```

To view the structure of a table in Oracle DB:

Syntax:

SQL> DESC <table name>;

```
SQL> DESC STUDENT;

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

```
ID                                                NUMBER(38)
SNAME                                             VARCHAR2(20)
FEE                                               NUMBER(6,2)
```

Alter:

- It is used to modify/ change the structure of a table in database.
- This command is having the following four sub commands are
    - alter – modify
    - alter – add
    - alter – rename
    - alter – drop

alter - modify:

- To change datatype and also size of a particular column.
- Syntax:
  SQL> ALTER TABLE <table name> MODIFY <column name> <new datatype> [new size];

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

Table altered.
```

alter - add:

- Adding a new column to an existing table.
- Syntax:
  SQL> ALTER TABLE <table name> ADD <new column name> <datatype> [size];

```
SQL> ALTER TABLE STUDENT ADD ADDRESS VARCHAR2(20);

Table altered.
```

alter - rename:

- To change a column name in a table.
- Syntax:
  SQL> ALTER TABLE <table name> RENAME COLUMN <old column name> TO <new column name>;

```
SQL> ALTER TABLE STUDENT RENAME COLUMN SNAME TO NAME;

Table altered.
```

alter - drop:

- To drop/ delete an existing column from a table.
- Syntax:
  SQL> ALTER TABLE <table name> DROP <COLUMN> <column name>;

```
SQL> ALTER TABLE STUDENT DROP COLUMN FEE;

Table altered.
```

Rename:

- It is used to change a table name in database.

  Syntax:
  SQL> RENAME <old table name> TO <new table name>;

```
SQL> RENAME STUDENT TO STUDENTDETAILS;

Table renamed.
```

Truncate:

- To delete all rows from a table at a time.
- We can delete rows but not columns of a table.
- By using truncate command, we cannot delete a specific row from a table because truncate does not supports "where clause" condition.

  Syntax:
  SQL> TRUNCATE TABLE <table name>;

```
SQL> TRUNCATE TABLE STUDENT;

Table truncated.
```

Drop:

- To drop a table (i.e., rows and columns) from database.

  Syntax:
  SQL> DROP TABLE <table name>;

```
SQL> DROP TABLE STUDENTDETAILS;

Table dropped.
```

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,

a. Recyclebin
b. Flashback
c. purge

Recyclebin:

- It is a pre-define/ system defined table which is used to stored information about dropped/ deleted tables.
- It will work as like windows recyclebin in system.

To view the structure of recyclebin:
Syntax:
SQL> DESC RECYCLEBIN;

```
SQL> DESC RECYCLEBIN;

 Name                           Null?              Type
 ------------------------------ --------        ---------------
 OBJECT_NAME                    NOT NULL            VARCHAR2(128)
 ORIGINAL_NAME                                      VARCHAR2(128)
```

To view information about dropped tables in recyclebin:
Syntax:
SQL> SELECT OBJECT_NAME, ORIGINAL_NAME FROM RECYCLEBIN;

```
SQL> SELECT OBJECT_NAME, ORIGINAL_NAME FROM RECYCLEBIN;

OBJECT_NAME                    ORIGINAL_NAME
------------------------------ --------------------------------
BIN$qZ3fLBtKTfuji7LSKAjBYQ==$0   STUDENTDETAILS
```

Flashback:

- This command is used to restore a dropped table from recyclebin to database.

  Syntax:
  SQL> FLASHBACK TABLE <table name> TO BEFORE DROP;

```
SQL> FLASHBACK TABLE STUDENTDETAILS TO BEFORE DROP;

Flashback complete.
```

Purge:

- This command is used to drop a table from recyclebin permanently (or) to drop a table from database permanently.

  Syntax 1: dropping a specific table from recyclebin
  SQL> PURGE TABLE <table name>;

```
SQL> PURGE TABLE TEST1;

Table purged.
```

  Syntax 2: dropping all tables from recyclebin
  SQL> PURGE RECYCLEBIN;

```
SQL> PURGE RECYCLEBIN;

Recyclebin purged.
```

  Syntax 3: drop a table from database permanently
  SQL> DROP TABLE <table name> PURGE;

```
SQL> DROP TABLE TEST1 PURGE;

Table dropped.
```

DML (Data Manipulation Language)

Insert:

- Inserting a new row data into a table.

  Syntax 1:
  SQL> INSERT INTO <table name> VALUES (value1, value2, ….);

Note:

- ✓ Using the above method, we should insert values to all columns otherwise we will get error.
- ✓ Before inserting any data, we have given the tablespace permission to the particular user, for that use below commands.

```
SQL> CONN
Enter user-name: system/tiger
Connected.
```

```
SQL> GRANT UNLIMITED TABLESPACE TO nirmala;

Grant succeeded.
```

```
SQL> INSERT INTO STUDENT VALUES (101, 'ALLEN', 'HYD');

1 row created.
```

Syntax 2:

SQL> INSERT INTO <table name> (required column names) VALUES (….);

Note: Using the above method, we can insert values for required columns only and remaining columns will take "NULL" by default.

```
SQL> INSERT INTO STUDENT (ID, NAME) VALUES (102, 'ALLEN');

1 row created.
```

How to insert multiple rows into a table continues:

- Substitutional operators: these operators are used to insert multiple rows data into a table continually.
- These are two types,
  - &: we can insert values to columns dynamically.
  - &&: 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.

Note: "/" – To execute the last executed query in SQL Plus environment multiple time we can use this once while inserting multiple data at a time.

Syntax 1 (&):

SQL> INSERT INTO <table name> values (&<column1>, &<column2>, ….);

```
SQL> INSERT INTO STUDENT VALUES (&ID, '&NAME', '&ADDRESS');
Enter value for id: 103
Enter value for name: WARD
Enter value for address: BLG
old   1: INSERT INTO STUDENT VALUES (&ID, '&NAME', '&ADDRESS')
new   1: INSERT INTO STUDENT VALUES (103, 'WARD', 'BLG')

1 row created.

SQL> /
Enter value for id: 104
```

```
Enter value for name: SMITH
Enter value for address: CNH
old   1: INSERT INTO STUDENT VALUES (&ID, '&NAME', '&ADDRESS')
new   1: INSERT INTO STUDENT VALUES (104, 'SMITH', 'CNH')

1 row created.
```

Syntax 2 (&):

SQL> INSERT INTO <table name> (required column names) values (&<column name1>, &<column name2>, ...);

```
SQL> INSERT INTO STUDENT (ID, NAME) VALUES (&ID, '&NAME');
Enter value for id: 105
Enter value for name: SCOTT
old   1: INSERT INTO STUDENT (ID, NAME) VALUES (&ID, '&NAME')
new   1: INSERT INTO STUDENT (ID, NAME) VALUES (105, 'SCOTT')

1 row created.

SQL> /
Enter value for id: 106
Enter value for name: JONES
old   1: INSERT INTO STUDENT (ID, NAME) VALUES (&ID, '&NAME')
new   1: INSERT INTO STUDENT (ID, NAME) VALUES (106, 'JONES')

1 row created.
```

Update:

- To update all rows data at a time in a table (or) updating a single row data in a table by using "WHERE" clause condition.

  ### Syntax:

  SQL> UPDATE <table name> SET <column name1> = <value1>, <column name2> = <value2>, ... [WHERE <condition>];

```
SQL> UPDATE STUDENT SET ADDRESS='PUNE' WHERE ID=106;

1 row updated.

SQL> UPDATE STUDENT SET FEE=6000;

5 rows updated.
```

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:

SQL> DELETE FROM <table name> [WHERE <condition>];

```
SQL> DELETE FROM STUDENT WHERE ID=101;

1 row deleted.

SQL> DELETE FROM STUDENT;

5 rows deleted.
```

Q. What is the difference between delete & truncate command?

Ans.

DELETE	TRUNCATE
a. It is a DML command	a. It is a DDL command
b. It can delete a specific row from a table	b. It cannot delete a specific row from a table
c. It supports "where clause" condition	c. It does not support "where clause" condition
d. Temporary data deletion	d. Permanent data deletion.
e. We can restore deleted data by using “rollback” command	e. We can’t restore deleted by using “rollback”.
f. Execution speed is slow (deleting rows in one-by-one manner)	f. Execution speed is fast (deleting all rows as a set/ at a time)

Note: flashback restore a table into database where as rollback restore data into a table.

Insert all:

- It is a DML command (oracle 9i). Which is used to insert rows into multiple tables at a time.
- But those rows should be an existing table only.

Syntax:

SQL> INSERT ALL INTO <table name 1> values (<column name1>, <column name2>, ...) INTO <table name 2> VALUES (<column name1>, <column name2>, …) INTO <table name 3> VALUES (<column name1>, <column name2>, …) …… INTO <table name n> VALUES (<column name1>, <column name2>, ……);

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

Table created.

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

Table created.

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

Table created.

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

12 rows created.
```

Merge

- 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" whereas data is not matching then those unmatching data/ rows are transferring from source table to destination table by using "INSERT command".

  Syntax:

  SQL> MERGE INTO <destination table name> <alias name> USING <source table name> <alias name> ON (<joining condition>) WHEN MATCHED THEN UPDATE SET <destination table alias name>. <column name1>=<source table alias name>.<column name1>, ... WHEN NOT MATCHED THEN INSERT (<destination table columns>) VALUES (<source table columns>);

```
SQL> CREATE TABLE NEWDEPT AS SELECT * FROM DEPT;

Table created.

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

1 row created.

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

```
1 row created.

SQL> SELECT * FROM NEWDEPT;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 DBA            HYD
        60 SAP            MUMBAI

6 rows selected.

SQL> SELECT * FROM DEPT;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON

SQL> MERGE INTO DEPT D USING NEWDEPT S ON (D. DEPTNO=S.DEPTNO)
  2  WHEN MATCHED THEN UPDATE SET D. DNAME=S.DNAME, D.LOC=S.LOC
  3  WHEN NOT MATCHED THEN INSERT (D. DEPTNO, D. DNAME, D.LOC)
VALUES (S. DEPTNO, S. DNAME, S.LOC);

6 rows merged.

SQL> SELECT * FROM DEPT;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 DBA            HYD
        60 SAP            MUMBAI

6 rows selected.
```

DQL/ DRL (Data Query/ Data Retrieval Language)

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:

SQL> SELECT */ <list of column> FROM <table name> [WHERE <condition>];

Note:

- ✓ Here, " * " is represent all columns in a table.
- ✓ To get the default table like EMP, DEPT, SALGRADE click on the link and copy all queries and paste into the “SQL Plus” and hit enter [Link].

```
SQL> SELECT * FROM DEPT WHERE DEPTNO=30;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        30 SALES          CHICAGO
```

Note:

- ✓ When we want to display the large-scale information/ data of a particular table in proper systematically then we need to set the following two properties are,
  1. pagesize n:
     - To display number of rows in a single page.
     - Here "n" is represented number of rows.
     - By default, a single page is displayed 14 rows.
     - Range is 0 to 50000.
     - Syntax
       SQL> SET PAGESIZE n;
       Ex: SET PAGESIZE 100;

  2. lines n:
     - Number of bytes in a single line.
     - Here "n" is representing number of bytes.
     - By default, each line allocates 80 bytes.
     - Range is 1 to 32767.
     - Syntax:
       SQL> SET LINES n;
       Ex: SET LINES 100;

- ✓ To clear SQL Plus editor screen:
  SQL> CL SCR;

- ✓ To disconnect/ exit from oracle database:
  SQL> EXIT;

Alias names:

- It is an alternate (or) temporary name for columns/ table.
- User can create alias names on two levels in DB:
  - column level
    - In this level we are creating alias names on columns.
    - Syntax:
      <column name> <column alias name>
      Ex: deptno dpno
  - table level:
    - in this level we creating alias names on table.
    - Syntax:
      <table name> <table alias name>
      Ex: dept d

Syntax to combined column + table level alias names by using "select" query:
SQL> SELECT <column name1> <column name1 alias name>, <column name2> <column name2 alias name>, ... FROM <table name> <table alias name>;

```
SQL> SELECT DEPTNO ID, DNAME NAME, LOC LOCATION FROM DEPT
DEPARTMENT;

        ID NAME           LOCATION
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

Concatenation operator (||):

- This operator is used to join two string values (or) two expressions in a select query.
- Syntax: <string 1> || <string 2>

```
SQL> SELECT 'WEL' || 'COME' MSG FROM DUAL;

MSG
-------
WELCOME

SQL> SELECT 'Mr. ' || ENAME || ' is working as a ' || JOB EMP_MSG
FROM EMP;

EMP_MSG
```

```
-----------------------------------
Mr. SMITH is working as a CLERK
Mr. ALLEN is working as a SALESMAN
Mr. WARD is working as a SALESMAN
Mr. JONES is working as a MANAGER
```

Distinct keyword:

- This keyword is used to eliminate duplicate values from a column and display unique values in query result.
- Syntax: DISTINCT <column name>

```
SQL> SELECT DISTINCT JOB FROM EMP;

JOB
---------
CLERK
SALESMAN
ANALYST
MANAGER
PRESIDENT
```

How to create a new table from the old table:

Syntax 1: Create a new table with copy of all rows & columns from old table.
SQL> CREATE TABLE <new table name> AS SELECT * FROM <old table name> [WHERE <true condition>];

```
SQL> CREATE TABLE NEWDEPT AS SELECT * FROM DEPT;

Table created.

SQL> CREATE TABLE NEWDEPT AS SELECT * FROM DEPT 1=1;

Table created.
```

Syntax 2: Create a new table with copy of all columns but not rows from old table.
SQL> CREATE TABLE <new table name> AS SELECT * FROM <old table name> WHERE <false condition>;

```
SQL> CREATE TABLE NEWDEPT AS SELECT * FROM DEPT 1=2;

Table created.
```

Syntax 3: Create a new table with copy of specific columns from old table.

SQL> CREATE TABLE <new table name> AS SELECT <column name1>, <column name2>, … FROM <old table name>;

```
SQL> CREATE TABLE SPEMP AS SELECT EMPNO, ENAME, SAL FROM EMP;

Table created.
```

Syntax 4: Create a new table with copy of specific rows from old table.
SQL> CREATE TABLE <new table name> AS SELECT <column name1>, <column name2>, … FROM <old table name> WHERE <condition>;

```
SQL> CREATE TABLE SPROWEMP AS SELECT EMPNO, ENAME, SAL FROM EMP
WHERE DEPTNO=20;

Table created.
```

How to copy data from one table to another table:

Syntax:
SQL> INSERT INTO <destination table name> SELECT * FROM <source table name>;

```
SQL> INSERT INTO NEWDEPT2 SELECT * FROM DEPT;

4 rows created.
```

Note: Number of columns must be name of destination and source table.

TCL (Transaction Control Language)

- Transaction is a unit of work that is performed against database.
- 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
  - Commit
  - Rollback
  - Savepoint

Commit:

- This command is used to make a transaction is permanent. These are two types.
  - Implicit commit:
    - These transactions are committed by Oracle DB by default.

        - DDL commands
    - Explicit commit:
        - These transactions are committed by user as per requirement.
        - DML commands

- Syntax: COMMIT;

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

Table created.

SQL> SELECT * FROM BRANCH;

no rows selected

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

1 row created.

SQL> SELECT * FROM BRANCH;

     BCODE BNAME
---------- ----------
      1021 SBI
```

Now let's close the SQL plus and open again and connect to the same user.

```
SQL> SELECT * FROM BRANCH;

no rows selected

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

1 row created.

SQL> COMMIT;

Commit complete.
```

If you now close now and open again and connect to the same user.

```
SQL> SELECT * FROM BRANCH;

     BCODE BNAME
---------- ----------
      1021  SBI
```

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)".

  Syntax:
  SQL> ROLLBACK;

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

1 row deleted.

SQL> SELECT * FROM BRANCH;

no rows selected

SQL> ROLLBACK;

Rollback complete.

SQL> SELECT * FROM BRANCH;

     BCODE BNAME
---------- ----------
      1021 SBI
```

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).
- For success COMMIT, for cancel ROLLBACK command we have to use after any transaction.

Savepoint:

- Whenever a user creates Savepoint with in the transaction then internally system is allocating a special memory for storing rows or transaction information in which we want to roll back in the future.

How to create a Savepoint:
Syntax:
SQL> SAVEPOINT <pointer name>;

How to rollback a Savepoint:

Syntax:

SQL> ROLLBACK TO <pointer name>;

```
SQL> SELECT * FROM BRANCH;

     BCODE BNAME
---------- ----------
      1021 SBI
      1022 HDFC
      1023 BOI
      1024 AXIS
      1025 ICICI

SQL> DELETE FROM BRANCH WHERE BCODE=1021;

1 row deleted.

SQL> DELETE FROM BRANCH WHERE BCODE=1025;

1 row deleted.

SQL> SAVEPOINT S1;

Savepoint created.

SQL> DELETE FROM BRANCH WHERE BCODE=1023;

1 row deleted.

SQL> SELECT * FROM BRANCH;

     BCODE BNAME
---------- ----------
      1022 HDFC
      1024 AXIS

SQL> ROLLBACK TO S1;

Rollback complete.

SQL> SELECT * FROM BRANCH;

     BCODE BNAME
---------- ----------
      1022 HDFC
      1023 BOI
      1024 AXIS
```

```
SQL> ROLLBACK;

Rollback complete.

SQL> SELECT * FROM BRANCH;

     BCODE BNAME
---------- ----------
      1021 SBI
      1022 HDFC
      1023 BOI
      1024 AXIS
      1025 ICICI
```

Note:

- ✓ Generally, by default all databases are having "ACID" properties to manage and maintain transactions in accurate and consistency data.
  - Atomicity: The entire transaction takes place at once or doesn't happen at all.
  - Consistency: The database must be consistent before and after the transaction.
  - Isolation: Multiple transactions occurred independently without interference.
  - Durability: Means once a transaction has been committed it will remain so, even in the event of errors, power loss, etc.
- ✓ DB Security
  - Authentication:
    - Authentication is a process of verifying the credentials (username & password) of a user before connect to Oracle server.
  - Authorization:
    - Authorization is process of verifying whether the user as permissions to perform any operation on the database.
    - These permissions are giving by DBA only, by using "DCL" commands.

DCL (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:
  SQL> GRANT <privilege name> ON <object name> TO {user} ;

Revoke:

- Revoke command removes user access rights/ privileges to the database or taking back the permission that is given to a user.

  Syntax:
  SQL> REVOKE <privilege name> ON <object name > FROM {user};

Note:

- ✓ 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.

Privileges:

- Privilege is right/ permission given to the users. All databases are having two types of privileges.
  - System privileges
  - Object privileges

System privileges:

- System privileges are given by DBA only.
- Such as create synonym, create view, create materialized view, create index, etc.

  Syntax:
  SQL> GRANT <system privilege> TO <user>;

```
-- Connect to DBA
SQL> CONN system/tiger
Connected.
-- Creating a new user
SQL> CREATE USER A1 IDENTIFIED BY A1;

User created.
```

```
-- Connect to the new user
SQL> CONN
Enter user-name: A1/A1
ERROR:
ORA-01045: user A1 lacks CREATE SESSION privilege; logon denied


Warning: You are no longer connected to ORACLE.

-- Again, connect to DBA for giving login permission to new user.
SQL> CONN
Enter user-name: system/tiger
Connected.
SQL> GRANT CONNECT TO A1;

Grant succeeded.

-- Now connected to the new user.
SQL> CONN
Enter user-name: A1/A1
Connected.

-- Creating table, but not able to create because don't have the
permission.
SQL> CREATE TABLE TEST1(SNO INT, NAME VARCHAR2(10));
CREATE TABLE TEST1(SNO INT, NAME VARCHAR2(10))
*
ERROR at line 1:
ORA-01031: insufficient privileges

-- Connecting to the DBA
SQL> CONN
Enter user-name: system/tiger
Connected.

-- Giving permission for table creation.
SQL> GRANT CREATE TABLE TO A1;

Grant succeeded.

-- Connecting to the new user.
SQL> CONN
Enter user-name: A1/A1
Connected.

-- Creating table, successfully created
SQL> CREATE TABLE TEST1(SNO INT, NAME VARCHAR2(10));
```

```
Table created.

-- Try to insert data but unable to do because of we don't have
tablespace.
SQL> INSERT INTO TEST1 VALUES(1, 'SMITH');
INSERT INTO TEST1 VALUES(1, 'SMITH')
            *
ERROR at line 1:
ORA-01950: no privileges on tablespace 'USERS'

-- Connect to DBA to give permission for tablespace.
SQL> CONN
Enter user-name: system/tiger
Connected.

-- Giving tablespace
SQL> GRANT UNLIMITED TABLESPACE TO A1;

Grant succeeded.

-- Connecting to new user
SQL> CONN
Enter user-name: A1/A1
Connected.

-- Now able to insert data into table.
SQL> INSERT INTO TEST1 VALUES(1, 'SMITH');

1 row created.
```

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:

SQL> GRANT <object privileges> ON <object name> to <user>;

```
SQL> CONN
Enter user-name: system/tiger
Connected.

SQL> SELECT * FROM DEPT;
```

```
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON

SQL> CREATE USER B1 IDENTIFIED BY B1;

User created.

SQL> GRANT CONNECT, UNLIMITED TABLESPACE TO B1;

Grant succeeded.

SQL> CONN
Enter user-name: B1/B1
Connected.

SQL> SELECT * FROM DEPT;
SELECT * FROM DEPT
              *
ERROR at line 1:
ORA-00942: table or view does not exist

SQL> CONN
Enter user-name: system/tiger
Connected.

SQL> GRANT SELECT ON DEPT TO B1;

Grant succeeded.
SQL> CONN
Enter user-name: B1/B1
Connected.

SQL> SELECT * FROM DEPT;
SELECT * FROM DEPT
              *
ERROR at line 1:
ORA-00942: table or view does not exist

SQL> SELECT * FROM SYSTEM.DEPT;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
```

```
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON

SQL> INSERT INTO SYSTEM.DEPT VALUES(50, 'DBA', 'HYD');
INSERT INTO SYSTEM.DEPT VALUES(50, 'DBA', 'HYD')
                    *
ERROR at line 1:
ORA-01031: insufficient privileges

SQL> UPDATE SYSTEM.DEPT SET LOC='PUNE' WHERE DEPTNO=30;
UPDATE SYSTEM.DEPT SET LOC='PUNE' WHERE DEPTNO=30
               *
ERROR at line 1:
ORA-01031: insufficient privileges

SQL> DELETE FROM SYSTEM.DEPT WHERE DEPTNO=10;
DELETE FROM SYSTEM.DEPT WHERE DEPTNO=10
                    *
ERROR at line 1:
ORA-01031: insufficient privileges

SQL> CONN
Enter user-name: system/tiger
Connected.

SQL> GRANT INSERT, UPDATE, DELETE ON DEPT TO B1;

Grant succeeded.

SQL> INSERT INTO SYSTEM.DEPT VALUES(50, 'DBA', 'HYD');

1 row created.

SQL> UPDATE SYSTEM.DEPT SET LOC='PUNE' WHERE DEPTNO=30;

1 row updated.

SQL> DELETE FROM SYSTEM.DEPT WHERE DEPTNO=10;

1 row deleted.

SQL> SELECT * FROM DEPT;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        20 RESEARCH       DALLAS
        30 SALES          PUNE
        40 OPERATIONS     BOSTON
        50 DBA            HYD
```

```
SQL> CONN
Enter user-name: system/tiger
Connected.

SQL> REVOKE ALL ON DEPT FROM B1;

Revoke succeeded.

SQL> CONN
Enter user-name: B1/B1
Connected.

SQL> SELECT * FROM SYSTEM.DEPT;
SELECT * FROM SYSTEM.DEPT
                     *
ERROR at line 1:
ORA-00942: table or view does not exist
```

Note: When user A1 want to give permission to user B1 then user A1 must be take permissions from DBA “WITH GRANT OPTION” statement.

Syntax:

SQL> GRANT <object privileges> ON <object name> to <user> WITH GRANT OPTION;

```
SQL> CONN
Enter user-name: system/tiger
Connected.

SQL> GRANT SELECT ON DEPT TO A1;

Grant succeeded.

SQL> CONN
Enter user-name: A1/A1
Connected.

SQL> SELECT * FROM SYSTEM.DEPT;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

```
SQL> GRANT SELECT ON SYSTEM.DEPT TO B1;
GRANT SELECT ON SYSTEM.DEPT TO B1
                       *
ERROR at line 1:
ORA-01031: insufficient privileges

SQL> CONN
Enter user-name: system/tiger
Connected.

SQL> GRANT SELECT ON DEPT TO A1 WITH GRANT OPTION;

Grant succeeded.

SQL> CONN
Enter user-name: A1/A1
Connected.

SQL> SELECT * FROM SYSTEM.DEPT;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON

SQL> GRANT SELECT ON SYSTEM.DEPT TO B1;

Grant succeeded.

SQL> CONN
Enter user-name: B1/B1
Connected.

SQL> SELECT * FROM SYSTEM.DEPT;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

- To view all users in Oracle DB then we have to use “ALL_USERS” data dictionary.

```
SQL> DESC ALL_USERS;
```

```
SQL> SELECT USERNAME FROM ALL_USERS;

USERNAME
------------------------------
HR
NIRMALA
A1
B1

SQL> DROP USER A1 CASCADE;

User dropped.
```

Role:

- Role is a collection of system/ object privileges and created/ giving by DBA only.

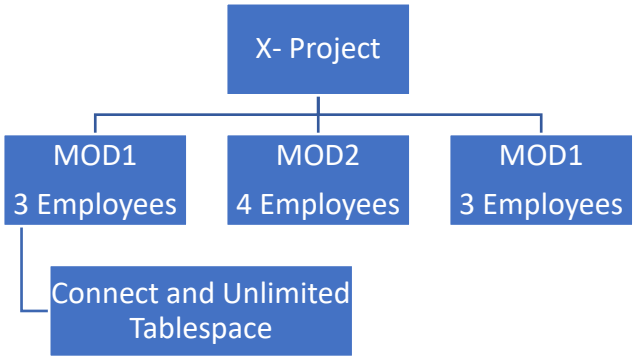
  Syntax to create role:
  SQL> CREATE ROLE <role name>;

  Syntax to assign system/ object privileges to role:
  SQL> GRANT <system/ object privileges> To <role name>;

  Syntax to assign role to users:
  SQL> GRANT <role name> TO <user1, user2, user3>

Why we need to create role:

- In real-time environment number 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 number of users.




```
SQL> CONN
```

```
Enter user-name: system/tiger
Connected.

SQL> CREATE USER A IDENTIFIED BY A;

User created.

SQL> CREATE USER B IDENTIFIED BY B;

User created.

SQL> CREATE USER C IDENTIFIED BY C;

User created.

SQL> CONN
Enter user-name: A/A
ERROR:
ORA-01045: user A lacks CREATE SESSION privilege; logon denied

SQL> CONN
Enter user-name: B/B
ERROR:
ORA-01045: user B lacks CREATE SESSION privilege; logon denied

SQL> CONN
Enter user-name: C/C
ERROR:
ORA-01045: user C lacks CREATE SESSION privilege; logon denied

SQL> CONN
Enter user-name: system/tiger
Connected.
SQL>

SQL> CREATE ROLE R1;

Role created.

SQL> GRANT CONNECT, CREATE TABLE TO R1;

Grant succeeded.

SQL> GRANT R1 TO A, B, C;

Grant succeeded.

SQL> CONN
```

```
Enter user-name: A/A
Connected.

SQL> CREATE TABLE T1(SNO INT);

Table created.
```

Note: "WITH GRANT OPTION" doesn't work on role.

```
SQL> conn
Enter user-name: system/tiger
Connected.
SQL> GRANT INSERT ON DEPT TO R1 WITH GRANT OPTION;
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>;

```
SQL> DROP ROLE R1;

Role dropped.
```

Syntax Granting permissions to different levels:

SQL> GRANT <system privileges> TO <user>/ <role>/ <public>;

&

SQL> GRANT <object privileges> ON <object name> TO <user>/ <role>/ <public>;

Syntax Revoking permissions to different levels:

SQL> REVOKE <system privileges> FROM <user>/ <role>/ <public>;

SQL> REVOKE <object privileges> ON <object name> FROM <user>/ <role>/ <public>;

Operators

- Assignment Operators =
- Arithmetic Operators +, -, *, /
- Relational Operators <, >, <=, >=, <> or !=
- Logical Operators AND, OR, NOT
- Set Operators Union, Union all, Intersect, Minus

Special Operators	(+v) operators	(-v) operators
	In	Not In
	Between	Not Between
	Is null	Is not null
	Like	Not Like

Assignment Operators:

- To assign a value to a variable/ attribute.
- Syntax: <column name> = <value>

Ex: SELECT * FROM EMP WHERE EMPNO = 7788;

Arithmetic Operators:

- To perform add, sub, multiplication, div operation.
- Syntax: <column name> <arithmetic operator> <value>

Ex: WAQ to display all employees' salaries after adding 1000.

```
SQL> SELECT SAL OLD_SAL, SAL+1000 NEW_SAL FROM EMP;

   OLD_SAL    NEW_SAL
---------- ----------
       800       1800
      1600       2600
```

Ex: WAQ to display ENAME, SAL and annual salary of employee from EMP table.

```
SQL> SELECT ENAME, SAL, SAL*12 ANNUAL_SAL FROM EMP;

ENAME             SAL ANNUAL_SAL
---------- ---------- ----------
SMITH             800       9600
ALLEN            1600      19200
WARD             1250      15000
JONES            2975      35700
```

Ex: WAQ to display ENAME, SAL and annual salary of employee from EMP table where JOB = 'MANAGER'.

```
SQL> SELECT ENAME, SAL, SAL*12 ANNUAL_SAL FROM EMP WHERE
JOB='MANAGER';
```

```
ENAME             SAL ANNUAL_SAL
---------- ---------- ----------
JONES            2975      35700
BLAKE            2850      34200
CLARK            2695      32340
```

Ex: WAQ to update employee salaries with increment of 10% who are working under DEPTNO=10.

```
SQL> UPDATE EMP SET SAL=SAL+SAL*10/100 WHERE DEPTNO=10;

3 rows updated.
```

Relational Operators:

- We are using for comparing a specific column value with given condition in the query.
- Syntax: WHERE <column name> <relational operator> <value>

Ex: WAQ to display employee whose salary is more than 1500.

```
SQL> SELECT * FROM EMP WHERE SAL > 1500;

EMPNO ENAME      JOB       MGR HIREDATE     SAL    COMM     DEPTNO
---------- ---------- --------- ---------- --------- ---------- ----
7499 ALLEN      SALESMAN 7698 20-FEB-81   1600        300        30
7566 JONES      MANAGER  7839 02-APR-81   2975                   20
```

Ex: WAQ to display employee who are join before 1981.

```
SQL> SELECT ENAME, JOB, HIREDATE FROM EMP WHERE HIREDATE<'01-JAN-
81';

ENAME      JOB       HIREDATE
---------- --------- ---------
SMITH      CLERK     17-DEC-80
```

Ex: WAQ to display employee who are join after 1981.

```
SQL> SELECT ENAME, JOB, HIREDATE FROM EMP WHERE HIREDATE>'31-DEC-
81';

ENAME      JOB       HIREDATE
---------- --------- ---------
ADAMS      CLERK     12-JAN-83
```

Logical Operators:

- To check more than one condition.

AND:

- It returns a value if all given conditions are true.
- Syntax: WHERE <condition 1> AND <condition 2> AND ….

Condition 1	Condition 2	Output
T	T	T
T	F	F
F	T	F
F	F	F

Ex: WAQ to display employee who are working as a “MANAGER” and whose name is “CLARK”.

```
SQL> SELECT * FROM EMP WHERE JOB='MANAGER' AND ENAME='CLARK';

EMPNO ENAME      JOB       MGR HIREDATE       SAL       COMM     DEPTNO
---------- ---------- --------- ---------- --------- ---------- ------
7782 CLARK      MANAGER   7839 09-JUN-81    2695                    10
```

OR:

- It returns a value if any one given condition is true.
- Syntax: WHERE <condition 1> OR <condition 2> OR ….

Condition 1	Condition 2	Output
T	T	T
T	F	T
F	T	T
F	F	F

Ex: WAQ to display employee whose name is “SMITH” or whose salary is more than 1500.

```
SQL> SELECT * FROM EMP WHERE ENAME='SMITH' OR SAL>1500;

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
```

NOT:

- It returns all values except the given condition values.
- Syntax: WHERE NOT <condition 1> AND NOT <condition 2> ….

Ex: WAQ to display employee who are not working as a “CLERK” and “SALESMAN”.

```
SQL> SELECT * FROM EMP WHERE NOT JOB='CLERK' AND NOT JOB='SALESMAN';

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
```

Set Operators:

- These operators are used to combined the result of two select statements as single set of values.
- Syntax: <select query1> <set operator> <select query2>

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

Union:

- A u B = {10, 20, 30, 40, 50}

Union All:

- A ul B = {10, 20, 30, 30, 40, 50}

Intersect:

- A i B = {30}

Minus:

- A – B = {10, 20}
- B – A = {40, 50}

Ex: WAQ to display all employees who are working in NIT organization.

```
SQL> SELECT * FROM EMP_HYD UNION SELECT * FROM EMP_CHENNAI;

       EID ENAME             SAL
---------- ---------- ----------
      1021 SMITH           85000
      1022 ALLEN           65000
      1023 WARD            48000
```

```
      1024 JONES           25000

SQL> SELECT * FROM EMP_HYD UNION ALL SELECT * FROM EMP_CHENNAI;

       EID ENAME             SAL
---------- ---------- ----------
      1021 SMITH           85000
      1022 ALLEN           65000
      1023 WARD            48000
      1021 SMITH           85000
      1024 JONES           25000
```

Ex: WAQ to display employees who are working in both branches.

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

       EID ENAME             SAL
---------- ---------- ----------
      1021 SMITH           85000
```

Ex: WAQ to display employees who are working in HYD but not in CHENNAI.

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

       EID ENAME             SAL
---------- ---------- ----------
      1022 ALLEN           65000
      1023 WARD            48000
```

Basic rules of Set operators:

a. Number of columns should be same in both select statements.
b. Order of column and datatypes of columns must match.

Special Operators:

In:

- It is Comparing list of values with a single condition.
- Syntax: WHERE <column name> IN (values1, values2, …);

Ex: WAQ to display list of employees whose EMPNO is 7349, 7788, 7900.

```
SQL> SELECT * FROM EMP WHERE EMPNO IN (7349, 7788, 7900);

EMPNO ENAME      JOB        MGR HIREDATE     SAL       COMM   DEPTNO
---------- ---------- --------- ---------- --------- ---------- ----
7788 SCOTT      ANALYST    7566 09-DEC-82    3000                 20
```

```
7900 JAMES      CLERK      7698 03-DEC-81    950                    30
```

Not In:

Ex: WAQ to display list of employees whose EMPNO is not 7349, 7788, 7900.

```
SQL> SELECT * FROM EMP WHERE EMPNO NOT IN (7349, 7788, 7900);

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
7566 JONES      MANAGER    7839 02-APR-81    2975                   20
```

Between:

- It is comparing based on a given range value.
- Syntax: WHERE <column name> BETWEEN <low value> AND <high value>

Basic rules for between:

a. Return all values including source and destination values also.
b. Apply from low value to high value.

Ex: WAQ to display employee whose salary between 1500 to 3000.

```
SQL> SELECT * FROM EMP WHERE SAL BETWEEN 1500 AND 3000;

EMPNO ENAME      JOB       MGR HIREDATE      SAL    COMM     DEPTNO
--------- ---------- --------- --------- --------- ---------- ----
7499 ALLEN      SALESMAN   7698 20-FEB-81   1600     300          30
7566 JONES      MANAGER    7839 02-APR-81   2975                  20
7698 BLAKE      MANAGER    7839 01-MAY-81   2850                  30
```

Ex: WAQ to display employee who are joined in 1981.

```
SQL> SELECT * FROM EMP WHERE HIREDATE BETWEEN '01-JAN-81' AND '31-
DEC-81';

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
```

Not Between:

Ex: WAQ to display employee who are not joined in 1981.

```
SQL> SELECT * FROM EMP WHERE HIREDATE NOT BETWEEN '01-JAN-81' AND
'31-DEC-81';

EMPNO ENAME      JOB       MGR HIREDATE    SAL       COMM     DEPTNO
---------- ---------- --------- ---------- --------- ---------- ----
7369 SMITH      CLERK     7902 17-DEC-80   800                    20
7788 SCOTT      ANALYST   7566 09-DEC-82   3000                   20
```

Is Null:

- It is comparing NULLs in a table then we need to use “is” keyword.
- NULL is a unknow/ undefined value in DB.
- NULL is not quals to zero also i.e. NULL != 0 & NULL != space.
- Syntax: WHERE <column name> IS NULL

Ex: WAQ to display employee whose commission is null.

```
SQL> SELECT * FROM EMP WHERE COMM IS NULL;

EMPNO ENAME      JOB       MGR HIREDATE      SAL      COMM     DEPTNO
---------- ---------- --------- ---------- --------- ---------- ----
7369 SMITH      CLERK     7902 17-DEC-80    800                   20
7566 JONES      MANAGER   7839 02-APR-81    2975                  20
```

Is not null:

Ex: WAQ to display whose commission is not null?

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

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
```

Ex: WAQ to display EMPNO, ENAME, SAL, COMM and also SAL+COMM from EMP table whose ENAME is “ALLEN”.

```
SQL> SELECT EMPNO, ENAME, SAL, COMM, SAL+COMM TOTAL_SAL FROM EMP
WHERE ENAME='ALLEN';

     EMPNO ENAME             SAL       COMM   TOTAL_COMO
---------- ---------- ---------- ---------- ----------
      7499 ALLEN            1600        300       1900
```

Ex: WAQ to display EMPNO, ENAME, SAL, COMM and also SAL+COMM from EMP table whose ENAME is “SMITH”.

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

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

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

NVL (Exp1, Exp2):

- NVL stands for Null Value.
- It's a predefined function which is used to replace a user defined value in place of NULL.
- Two arguments are expression 1 and expression 2
- If expression 1 is null then it returns expression 2 value
- If expression 1 is not null then it returns expression 1 value only.

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

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

SQL> SELECT NVL(50, 100) FROM DUAL;

NVL(50,100)
-----------
         50
```

Let's rewrite the above query using NVL function.

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

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

NVL2 (Exp1, Exp2, Exp3):

- It is an extension of NVL ().
- Here we have 3 arguments.
- If expression 1 is null then it will return expression 3 i.e. user defined value.

- If expression 1 is not null then it will return expression 2 i.e. also user defined value.

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

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

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

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

Ex: WAQ to update all employee commission based on the following conditions are

a. If employee comm is null then update those employee commission as 500
b. If employee comm is not null then update those employee commission as commission + 500;

```
SQL> UPDATE EMP SET COMM = NVL2(COMM, COMM+500, 500);

14 rows updated.
```

Like:

- It is used to comparing a specific string character pattern.
- When we are using like operator, we should user “wildcard” operators.
  - %: represent the remaining group of characters after selected character.
  - _: counting a single character from an expression
- Syntax: WHERE <column name> LIKE <value> <wildcard operator>

Ex: WAQ to display employee whose name starts with ‘S’ character.

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

       EID ENAME             SAL
---------- ---------- ----------
      1021 SMITH           85000
```

Ex: WAQ to display employee whose name is having 4 characters.

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

       EID ENAME             SAL
---------- ---------- ----------
      1023 WARD            48000
```

Ex: WAQ to display employee whose name ends with ‘N’ character.

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

       EID ENAME             SAL
---------- ---------- ----------
      1022 ALLEN           65000
```

Ex: WAQ to display employee whose name is having ‘I’ character.

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

       EID ENAME             SAL
---------- ---------- ----------
      1021 SMITH           85000
```

Ex: WAQ to display employee whose name’s second character is ‘L’.

```
SQL> SELECT * FROM EMP_HYD WHERE ENAME LIKE '_L%';

       EID ENAME             SAL
---------- ---------- ----------
      1022 ALLEN           65000
      1022 SLIM            75000
```

Note: Create a table and filled the data with special characters.

Like with special characters:

Ex: WAQ to display employee whose name is having “@”.

```
SQL> SELECT * FROM EMP1 WHERE ENAME LIKE '%@%';

       EID ENAME
---------- ----------
       103 MILL@ER
```

Ex: WAQ to display employee whose name having “#”.

```
SQL> SELECT * FROM EMP1 WHERE ENAME LIKE '%#%';
```

```
       EID ENAME
---------- ----------
       101 SMI#TH
```

Note:

- ✓ If in data we have "_" and "%" we will not get the exact data when we will check having "_" and "%" respectively.
- ✓ Because by default "_, %" are treated as wildcard operators but not special characters.
- ✓ To avoid we have to use "escape '\'".

Ex: WAQ to display employee whose name having "_".

```
SQL> SELECT * FROM EMP1 WHERE ENAME LIKE '%\_%'ESCAPE'\';

       EID ENAME
---------- ----------
       102 _WARD
```

Not Like:

Ex: WAQ to display employee whose name is not starts with "S" character.

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

       EID ENAME             SAL
---------- ---------- ----------
      1022 ALLEN           65000
      1023 WARD            48000
```

Functions

- To perform some task & must return a 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)

Pre-define functions:

- There are built in functions used in SQL and PL/SQL
- These are again classified into two categories.
  - Single row functions (scalar functions)
  - Multiple row functions (grouping functions)

Single Row Functions

- These functions are always returning a single row (or) a single value.
    - Numeric functions
    - String functions
    - Date functions
    - Conversion functions

How to call a function:

Syntax:

SQL> SELECT <function name>(values) FROM DUAL;

Q. What is dual?

Ans.

- It is system defined/ pre-define table in oracle.
- Having single column & single row
- It is called as dummy table in oracle.
- Testing functions (pre-define & user define) functionalities.

To view structure of DUAL table:

```
SQL> DESC DUAL;

 Name                                      Null?    Type
----------------------------------------- -------- ----------------
 DUMMY                                              VARCHAR2(1)
```

To view data of DUAL table:

```
SQL> SELECT * FROM DUAL;

D
-
X
```

Numeric functions:

1. ABS ():
    - To converts (-ve) value into (+ve) value.
    - Syntax: ABS (number)

```
SQL> SELECT ABS (-12) FROM DUAL;
  ABS (-12)
----------
        12
```

2. CEIL ():
    - Returns a value which is greater than or equal to given value.
    - Syntax: CEIL (number)

```
SQL> SELECT CEIL (9.3) FROM DUAL;
 CEIL (9.3)
----------
        10
```

3. FLOOR ():
    - Return a value which is greater than to the given expression.
    - Syntax: FLOOR (number)

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

FLOOR (9.3)
----------
         9
```

4. MOD ():
    - Returns a remainder value.
    - Syntax: MOD (m, n)

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

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

5. POWER ():
    - Return the power of given expression.
    - Syntax: POWER (m, n)

```
SQL> SELECT POWER (2,3) FROM DUAL;

POWER (2,3)
----------
         8
```

6. ROUND ():
    - Return the nearest value of given expression.
    - Syntax: ROUND (number, [decimal places])

```
SQL> SELECT ROUND (35.45) FROM DUAL;
```

```
ROUND (35.45)
-------------
           35

SQL> SELECT ROUND (35.75) FROM DUAL;

ROUND (35.75)
-------------
           36
```

7. TRUNC ():
    - Returns an exact value from give expression.
    - Syntax: TRUNC (number)

```
SQL> SELECT TRUNC (35.45) FROM DUAL;

TRUNC (35.45)
-------------
           35

SQL> SELECT TRUNC (35.75) FROM DUAL;

TRUNC (35.75)
-------------
           35
```

8. TRUNC (number, decimal places):
    - Returns a value which will specify number of decimal places.
    - Syntax: TRUNC (number, decimal places):

```
SQL> SELECT TRUNC (35.456, 2) FROM DUAL;

TRUNC (35.456,2)
----------------
           35.45
```

String functions:

1. LENGTH ():
    - It returns amount of characters/ length of given string.
    - Syntax: LENGTH (string)

```
SQL> SELECT LENGTH ('HELLO') FROM DUAL;

LENGTH('HELLO')
```

```
---------------
              5
```

2. LOWER ():
    - To convert upper case characters into lower case characters.
    - Syntax: LOWER (string)

```
SQL> SELECT LOWER ('SMITH') FROM DUAL;

LOWER
-----
smith
```

3. UPPER ():
    - To convert lower case characters into upper case characters.
    - Syntax: UPPER (string)

```
SQL> SELECT UPPER ('smith') FROM DUAL;

UPPER
-----
SMITH
```

4. INITCAP ():
    - To convert first character into capital.
    - Syntax: INITCAP (string)

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

INITC
-----
Hello
```

5. LTIRM ():
    - To remove unwanted spaces (or) unwanted characters from left side of given string.
    - Syntax: LTRIM (string1, [string2])

```
SQL> SELECT LTRIM ('      HELLO') FROM DUAL;

LTRIM
-----
HELLO

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

```
LTR
---
SAI
```

6. RTRIM ():
    - To remove unwanted spaces (or) unwanted characters from right side of given string.
    - Syntax: RTRIM (string1, [string2])

```
SQL> SELECT RTRIM ('HELLO          ') FROM DUAL;

RTRIM
-----
HELLO

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

RTR
---
SAI
```

7. TRIM ():
    - To remove unwanted spaces (or) unwanted characters from both sides of given string.
    - Syntax: TRIM ('trimming char' FROM 'string')

```
SQL> SELECT TRIM ('X' FROM 'XXXXXXXXXXSAIXXXXXXXXXX') FROM DUAL;

TRI
---
SAI
```

8. LPAD ():
    - To fill a string with specific character on left side of given string.
    - Syntax: LPAD (string1, length, string2)

```
SQL> SELECT LPAD ('HELLO', 3) FROM DUAL;

LPA
---
HEL

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

```
LPAD ('HELL
----------
     HELLO

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

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

9. RPAD ():
    - To fill a string with specific character on right side of given string.
    - Syntax: RPAD (string1, length, string2)

```
SQL> SELECT RPAD ('HELLO', 3) FROM DUAL;

RPA
---
HEL

SQL> SELECT RPAD ('HELLO', 10) FROM DUAL;

RPAD ('HELL
----------
HELLO

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

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

10. CONCAT ():
    - adding two string expressions.
    - Syntax: CONCAT (string1, string2)

```
SQL> SELECT CONCAT ('GOOD ', 'MORNING') FROM DUAL;

CONCAT ('GOOD
-------------
GOOD MORNING
```

11. REPLACE ():
    - To replace one string with another string.
    - Syntax: REPLACE (string1, string2, string3)

```
SQL> SELECT REPLACE ('WELCOME', 'LC', 'XYZ') FROM DUAL;

REPLACE (
--------
WEXYZOME
```

12.TRANSLATE ():

- To translate a single character with another single character.
- Syntax: TRANSLATE (string1, string2, string3)

```
SQL> SELECT TRANSLATE ('WELCOME', 'LC', 'XYZ') FROM DUAL;

TRANSLA
-------
WEXYOME

SQL> SELECT TRANSLATE ('WELCOME', 'LCO', 'XYZ') FROM DUAL;

TRANSLA
-------
WEXYZME

SQL> SELECT TRANSLATE ('345262562569044', '0123456789',
'ABCDEFGHIJ') FROM DUAL;

TRANSLATE ('3452
---------------
DEFCGCFGCFGJAEE
```

13.SUBSTR ():

- It returns required substring from given string expression.
- Syntax: SUBSTR (string1, <starting position of char>, <length of characters>)

```
SQL> SELECT SUBSTR ('WELCOME', 3, 2) FROM DUAL;

SU
--
LC

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

SUB
---
ELC
```

14.INSTR ():

- Returns occurrence position of a character in the given string.
- Syntax: INSTR (string1, string2, <starting position of character>, <occurrence position of character>)

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

INSTR('HELLOWELCOME','L')
-------------------------
                        3

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

INSTR('HELLOWELCOME','L',1,2)
-----------------------------
                            4

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

INSTR('HELLOWELCOME','L',1,1)
-----------------------------
                            3

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

INSTR('HELLOWELCOME','L',1,3)
-----------------------------
                            9

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

INSTR('HELLOWELCOME','L',1,4)
-----------------------------
                            0
```

Note: Position of character always fixed either count from left to right (or) right to left.

Date functions:

1. SYSDATE:
   - It returns current date information of the system.

```
SQL> SELECT SYSDATE FROM DUAL;

SYSDATE
---------
```

```
04-MAR-24

SQL> SELECT SYSDATE+10 FROM DUAL;

SYSDATE+1
---------
14-MAR-24

SQL> SELECT SYSDATE-10 FROM DUAL;

SYSDATE-1
---------
23-FEB-24
```

2. ADD_MONTHS ():
    - To add number of months to the given date expression.
    - Syntax: ADD_MONTHS (date, <number of months>)

```
SQL> SELECT ADD_MONTHS (SYSDATE, 5) FROM DUAL;

ADD_MONTH
---------
04-AUG-24

SQL> SELECT ADD_MONTHS (SYSDATE, -5) FROM DUAL;

ADD_MONTH
---------
04-OCT-23
```

3. MONTHS_BETWEEN ():
    - Returns number of months between the give two date expressions.
    - Syntax: MONTHS_BETWEEN (date1, date2)

```
SQL> SELECT MONTHS_BETWEEN ('04-MAR-2024', '04-OCT-2024') FROM DUAL;

MONTHS_BETWEEN('04-MAR-2024','04-OCT-2024')
-------------------------------------------
                                         -7

SQL> SELECT MONTHS_BETWEEN ('04-OCT-2024', '04-MAR-2024') FROM DUAL;

MONTHS_BETWEEN('04-OCT-2024','04-MAR-2024')
-------------------------------------------
                                          7
```

Note: Here, date1 is always greater than date2 otherwise oracle returns negative value.

4. LAST_DAY ():
    - Returns the last day of the given month.
    - Syntax: LAST_DAY (date)

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

LAST_DAY (
---------
31-MAR-2
```

5. NEXT_DAY ():
    - returns the next specified day from the given date.
    - Syntax: NEXT_DAY (date, '<day name>')

```
SQL> SELECT NEXT_DAY (SYSDATE,'SUNDAY') FROM DUAL;

NEXT_DAY (
---------
10-MAR-24
```

Conversion functions:

1. TO_CHAR ():
    - To convert date type to char type and also display date expression in different formats.
    - Syntax: TO_CHAR (date, [<format>])

a. Year formats:
- YYYY – display in 4 digits format of year.
- YY – display the last 2 digit of year.
- YEAR – display complete spell of year.
- CC – centaury
- AD/ BC - AD year/ BC year

```
SQL> SELECT TO_CHAR (SYSDATE, 'YYYY') FROM DUAL;

TO_C
----
2024

SQL> SELECT TO_CHAR (SYSDATE, 'YY') FROM DUAL;
```

```
TO
--
24

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

TO_CHAR(SYSDATE,'YEAR')
----------------------------------------
TWENTY TWENTY-FOUR

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

TO_CHAR(SYSDATE,'YYYYYYYEARCCBCAD')
------------------------------------------------------------
2024 24 TWENTY TWENTY-FOUR 21 AD AD
```

Ex: WAQ to display employee who are joined in year 1982 by using TO_CHAR function.

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

EMPNO ENAME      JOB         MGR HIREDAT    SAL       COMM     DEPTNO
---------- ---------- --------- ---------- --------- ---------- ----
7788 SCOTT      ANALYST    7566 09-DEC-82  3000      500          20
7934 MILLER     CLERK      7782 23-JAN-82  1430      500          10
```

b. Month formats:
   - MM – month in number format
   - MON - first three characters from month spelling
   - MONTH – full name of month

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

TO_CHAR(SYSDATE,'MMMONMONTH')
-------------------------------------------------
03 MAR MARCH
```

c. 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 characters of day spelling

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

TO_CHAR(SYSDATE,'DDDDDDDAYDY')
------------------------------------------------------------
064 04 2 MONDAY    MON
```

d. Quarter formats:

- Q - one-digit quarter of the year
  - 1 - JAN – MA
  - 2 - APR – JUN
  - 3 - JUL – SEP
  - 4 - OCT – DEC

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

T
-
1
```

e. Week formats:

- WW - week of the year
- W - week of month

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

TO_C
----
10 1
```

f. time format:

- HH – 12 Hrs format
- HH24 – 24 Hrs format
- MI – minute part
- SS – seconds part
- AM / PM - am time (or) pm time

```
SQL> SELECT TO_CHAR (SYSDATE, 'HH HH24 MI SS AM') FROM DUAL;

TO_CHAR (SYSDAT
---------------
12 00 08 47 AM
```

2. TO_DATE ():
    - To convert char type to oracle default date format type (DD-MON-YY).
    - Syntax: TO_DATE (string, [format])

```
SQL> SELECT TO_DATE ('11/AUGUST/2020') FROM DUAL;

TO_DATE ('
---------
11-AUG-20

SQL> SELECT TO_DATE ('11/AUGUST/2020') + 10 FROM DUAL;

TO_DATE ('
---------
21-AUG-20
```

Multiple Row/ Aggregative Functions

- These functions are called as “grouping functions/ aggregative functions”.
- These functions are returning either group of value or Sigle value

1. SUM ():
    - It returns sum or total of a specific column value.

```
SQL> SELECT SUM(SAL) FROM EMP;

  SUM(SAL)
----------
     29900

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

  SUM(SAL)
----------
      8520
```

2. AVG ():
    - It returns average of a specific column values.

```
SQL> SELECT AVG(SAL) FROM EMP;

  AVG(SAL)
----------
2135.71429

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

  AVG(SAL)
----------
3208.33333
```

3. MIN ():
    - It returns min value from group of values.

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

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

SQL> SELECT MIN(HIREDATE) FROM EMP;

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

4. MAX ():
    - It returns max value from group of values.

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

  MAX(SAL)
----------
      5500

SQL> SELECT MAX(HIREDATE) FROM EMP;

MAX (HIRED
---------
12-JAN-83
```

5. COUNT ():
    - It returns number of rows in a table/ number of values in a column.
    - There are three types,

i. COUNT (*)
ii. COUNT (<column name>)
iii. COUNT (DISTINCT <column name>)

a. COUNT (*):
   - Counting all rows (duplicates & nulls) in a table.

```
SQL> SELECT COUNT (*) FROM TEST;

  COUNT(*)
----------
         6
```

b. COUNT (<column name>):
   - Counting all values including duplicate values but not null values.

```
SQL> SELECT COUNT(NAME) FROM TEST;

COUNT(NAME)
-----------
          5
```

c. COUNT (DISTINCT <column name>):
   - Counting unique values from a column, here "distinct" keyword is eliminating duplicate values.

```
SQL> SELECT COUNT (DISTINCT NAME) FROM TEST;

COUNT(DISTINCTNAME)
-------------------
                  3
```

Clauses

- Clause is a 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 subtotal 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 subtotal & grand total (single column)
- CUBE - finding subtotal & grand total (multiple columns)

WHERE:

- Filtering rows in one-by-one manner before grouping data in table.
- It can be used in “select, update, delete” commands.
- Syntax: WHERE <filtering condition>

```
SQL> SELECT * FROM EMP WHERE EMPNO = 7788;

EMPNO ENAME      JOB       MGR   HIREDATE     SAL     COMM     DEPTNO
---------- ---------- --------- ---------- --------- ---------- ----
7788 SCOTT      ANALYST    7566 09-DEC-82    3000     500         20
```

ORDER BY:

- To arrange a specific columns value in ascending/ descending order.
- 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.
- It can be used in "select" command only.

Syntax:
SQL> SELECT */ <list of columns> FROM <table name> ORDER BY <column name1> <ASC/ DESC>, <column name2> <ASC/ DESC >, ...

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

ENAME      JOB       HIREDATE         SAL
---------- --------- --------- ----------
SMITH      CLERK     17-DEC-80        800
JAMES      CLERK     03-DEC-81        950
ADAMS      CLERK     12-JAN-83       1100
WARD       SALESMAN  22-FEB-81       1250
MARTIN     SALESMAN  28-SEP-81       1250

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

ENAME      JOB       HIREDATE         SAL
---------- --------- --------- ----------
WARD       SALESMAN  22-FEB-81       1250
TURNER     SALESMAN  08-SEP-81       1500
SMITH      CLERK     17-DEC-80        800
SCOTT      ANALYST   09-DEC-82       3000
```

Ex: WAQ to display employee details who are working under DEPTNO is 20 and arrange those employee salaries in descending order?

```
SQL> SELECT ENAME, JOB, HIREDATE, SAL FROM EMP WHERE DEPTNO=20 ORDER
BY SAL DESC;

ENAME      JOB       HIREDATE         SAL
---------- --------- --------- ----------
SCOTT      ANALYST   09-DEC-82       3000
FORD       ANALYST   03-DEC-81       3000
JONES      MANAGER   02-APR-81       2975
ADAMS      CLERK     12-JAN-83       1100
```

Ex: WAQ to arrange employee deptno in ascending order and those employee salaries in descending order from each deptno wise?

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

ENAME      JOB       HIREDATE         SAL     DEPTNO
---------- --------- --------- ---------- ----------
KING       PRESIDENT 17-NOV-81       5500         10
CLARK      MANAGER   09-JUN-81       2695         10
MILLER     CLERK     23-JAN-82       1430         10
SCOTT      ANALYST   09-DEC-82       3000         20
FORD       ANALYST   03-DEC-81       3000         20
JONES      MANAGER   02-APR-81       2975         20
```

Note: Order by clause not only on column names even though we can apply on position of column in select query.

```
SQL> SELECT ENAME, JOB, HIREDATE, SAL, DEPTNO FROM EMP ORDER BY 2;

ENAME      JOB       HIREDATE         SAL     DEPTNO
---------- --------- --------- ---------- ----------
SCOTT      ANALYST   09-DEC-82       3000         20
FORD       ANALYST   03-DEC-81       3000         20
MILLER     CLERK     23-JAN-82       1430         10
```

ORDER BY with “NULL” values:

- Using order by clause on "null" values column, 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 have to use null clauses are “NULLS FIRST" and "NULLS LAST".

- NULLS FIRST – In ascending order when we want to display nulls in the first then we need to use this.
- NULLS FIRST – In ascending order when we want to display nulls in the last then we need to use this.

```
SQL> SELECT ENAME, JOB, HIREDATE, SAL, COMM FROM EMP ORDER BY COMM
NULLS FIRST;

ENAME      JOB       HIREDATE         SAL       COMM
---------- --------- --------- ---------- ----------
SMITH      CLERK     17-DEC-80         800
CLARK      MANAGER   09-JUN-81        2450
FORD       ANALYST   03-DEC-81        3000
JAMES      CLERK     03-DEC-81         950

SQL> SELECT ENAME, JOB, HIREDATE, SAL, COMM FROM EMP ORDER BY COMM
NULLS LAST;

ENAME      JOB       HIREDATE         SAL       COMM
---------- --------- --------- ---------- ----------
ALLEN      SALESMAN  20-FEB-81       1600        300
WARD       SALESMAN  22-FEB-81       1250        500
MARTIN     SALESMAN  28-SEP-81       1250       1400
MILLER     CLERK     23-JAN-82       1300
SCOTT      ANALYST   09-DEC-82       3000
```

GROUP BY:

- It is used to grouping of 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 1st grouping similar data-based columns and later an aggregative function will execute on each group of data to produce accurate result.

Syntax:

SQL> SELECT <column name1>, <column name2>, ..., <aggregative function name1>, …. FROM <table name> GROUP BY <column name1>, <column name2>, ...;

Ex: WAQ to find out number of employees from each deptno wise.

```
SQL> SELECT DEPTNO, COUNT (*) FROM EMP GROUP BY DEPTNO ORDER BY
DEPTNO;
```

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

Ex: WAQ to display sum of salaries of each JOB wise.

```
SQL> SELECT JOB, SUM (SAL) FROM EMP GROUP BY JOB;

JOB         SUM(SAL)
--------- ----------
CLERK           4150
SALESMAN        5600
ANALYST         6000
MANAGER         8275
PRESIDENT       5000

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

    DEPTNO  EMPLOYEES    SUM_SAL    AVG_SAL    MAX_SAL    MIN_SAL
---------- ---------- ---------- ---------- ---------- ----------
        10          3       8750 2916.66667       5000       1300
        20          5      10875       2175       3000        800
        30          6       9400 1566.66667       2850        950
```

HAVING:

- It is used to filtering rows after grouping data in a table.
- It can be used along with "group by" clause, without “group by” we can’t use “having” clause.

Syntax:

SQL> SELECT <column name1>, <column name2>, ..., <aggregative function name1>, ... FROM <table name> GROUP BY <column name1>, <column name2>, ... HAVING <filtering condition>;

Ex: WAQ to displays in which deptno number of employees are more than 3.

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

    DEPTNO   COUNT (*)
---------- ----------
```

```
        30          6
        20          5
```

Ex: WAQ to display sum of salaries of JOB if sum of salary of JOB is less than 8000?

```
SQL> SELECT JOB, SUM (SAL) TOTAL_SAL FROM EMP GROUP BY JOB HAVING
SUM (SAL)<8000;

JOB        TOTAL_SAL
--------- ----------
CLERK           4150
SALESMAN        5600
ANALYST         6000
PRESIDENT       5000
```

Using all clauses in a single select statement:

Syntax:

SQL> SELECT <col1>, <col2>, …. <aggregative function name>, …. FROM <table name> [WHERE <filtering condition> GROUP BY <col1>, <col2> HAVING <filter condition> ORDER BY <col1> <ASC/ DESC>, ….;

```
SQL> SELECT DEPTNO, COUNT (*) FROM EMP WHERE SAL>100 GROUP BY DEPTNO
HAVING COUNT (*)>=5 ORDER BY DEPTNO DESC;

    DEPTNO   COUNT (*)
---------- ----------
        30          6
        20          5
```

Order of execution of above query:

- FROM
- WHERE
- GROUP BY
- HAVING
- SELECT
- ORDER BY

Q. What are the differences between "where" and "having" clause?

Ans.

WHERE	HAVING
a. Where clause condition is executed on each row of a table.	a. Having clause condition is executed on group of rows of a table.

b. It can be apply before group by clause.	b. It can be apply after group by clause.
c. It can't support aggregative functions.	c. It can supports aggregative functions.
d. Without group by we can use where clause.	d. Without group by we can't use having clause.

ROLLUP & CUBE:

- These are the special clauses.
- To finding subtotal & grand total based on columns.
- Working along with "group by" clause.
- ROLLUP will find sub & grand total based on a single column.
- CUB will find sub & grand total based on multiple columns.

Syntax for ROLLUP:

SQL> SELECT <col1>, <col2>, …., <aggregative function name>, …. FROM <table name> GROUP BY ROLLUP (<col1>, <col2>, <col3>, ..., <col n>);

Note: Here first column is called as operational column and other columns are supporting columns.

ROLLUP with single columns:

```
SQL> SELECT DEPTNO, COUNT (*) FROM EMP GROUP BY ROLLUP(DEPTNO);

    DEPTNO   COUNT (*)
---------- ----------
        10          3
        20          5
        30          6
                   14
```

ROLLUP with multiple columns:

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

    DEPTNO JOB         COUNT(*)
---------- --------- ----------
        10 CLERK              1
        10 MANAGER            1
        10 PRESIDENT          1
        10                    3
```

```
        20 CLERK                2
        20 ANALYST              2
        20 MANAGER              1
        20                      5
        30 CLERK                1
        30 MANAGER              1
        30 SALESMAN             4
        30                      6
                               14
```

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 for CUBE:

SQL> SELECT <col1>, <col2>, …., <aggregative function name>, …. FROM <table name> GROUP BY CUBE (<col1>, <col2>, <col3>, ..., <col n>);

Note: Here all columns are operational columns.

CUBE with single columns:

```
SQL> SELECT DEPTNO, COUNT (*) FROM EMP GROUP BY CUBE(DEPTNO);

    DEPTNO   COUNT (*)
---------- ----------
                   14
        10          3
        20          5
        30          6
```

CUBE with multiple columns:

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

    DEPTNO JOB         COUNT(*)
---------- --------- ----------
                            14
           CLERK             4
           ANALYST           2
           MANAGER           3
           SALESMAN          4
           PRESIDENT         1
        10                   3
        10 CLERK             1
        10 MANAGER           1
```

```
        10 PRESIDENT           1
        20                     5
        20 CLERK               2
        20 ANALYST             2
        20 MANAGER             1
        30                     6
        30 CLERK               1
        30 MANAGER             1
        30 SALESMAN            4
```

GROUPING_ID ():

- It is used more compact way to identify sub and grand total rows.
- Id number
  - 1: To represent subtotal of first grouping column.
  - 2: To represent subtotal of second grouping column.
  - 3: To represent subtotal of third grouping column.
  - 4: grand total row.

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

    DEPTNO JOB         COUNT(*) GROUPING_ID(DEPTNO,JOB)
---------- --------- ---------- -----------------------
        10 CLERK              1                       0
        10 MANAGER            1                       0
        10 PRESIDENT          1                       0
        10                    3                       1
        20 ANALYST            2                       0
        20 CLERK              2                       0
        20 MANAGER            1                       0
        20                    5                       1
        30 CLERK              1                       0
        30 MANAGER            1                       0
        30 SALESMAN           4                       0
        30                    6                       1
           ANALYST            2                       2
           CLERK              4                       2
           MANAGER            3                       2
           PRESIDENT          1                       2
           SALESMAN           4                       2
                             14                       3
```

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, order details, products details, etc. To combined data and retrieve data from those multiple tables then we need joins.

- Joins are again classified into two ways
  - Non-Ansi joins: (oracle 8i joins)
    - Equi join
    - Non-Equi join
    - Self-join
  - Ansi joins: (oracle 9i joins)
    - Inner join
    - Outer join
      - Left outer join
      - Right outer join
      - Full outer join
    - Cross join (or) Cartesian join
    - Natural join

Non-Ansi Joins	Ansi Joins
a. Oracle 8i version	a. Oracle 9i version
b. old style format	b. new style format
c. based on “Where” clause	c. “on” clause condition
d. not a potability	d. portability

Note:

- ✓ 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.
- ✓ When we are retrieving data from multiple tables based on "where" clause condition then we called as non-Ansi join.

Syntax:

SQL> SELECT * FROM <table name1>, <table name2> WHERE <join condition>;

- ✓ When we are retrieving data from multiple tables with "on”/ "using" clause condition then we called as Ansi join.

Syntax:

SQL> SELECT * FROM <table name1> <join key> <table name2 > on <join condition>;

First create the following table with data

```
SQL> SELECT * FROM STUDENT;

       SID SNAME                                 CID
---------- ------------------------------ ----------
         1 SMITH                                  10
         2 ALLEN                                  10
         3 WARD                                   20
         4 JONES                                  30

SQL> SELECT * FROM COURSE;

       CID CNAME                               CFEE
---------- ------------------------------ ----------
        10 ORACLE                               2500
        20 JAVA                                 5000
        40 PYTHON                              45000
```

Equi Join/ Inner Join:

- This join is retrieving data from multiple tables based on "equal operator (=) ".
- We should have a common column in tables and that common column datatype must be same.
- It is always retrieval matching rows (data) only.
- When we perform any join operation between tables there is no need to have relationship(optional). (i.e. primary key & foreign key relation).

Ex: WAQ to retrieve student and the corresponding course details from Student, course tables?

Non-Ansi format:

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

       SID SNAME             CID        CID CNAME             CFEE
---------- ---------- ---------- ---------- ---------- ----------
         1 SMITH              10         10 ORACLE           2500
         2 ALLEN              10         10 ORACLE           2500
         3 WARD               20         20 JAVA             5000

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

       SID SNAME             CID        CID CNAME             CFEE
---------- ---------- ---------- ---------- ---------- ----------
         1 SMITH              10         10 ORACLE           2500
```

```
         2 ALLEN                10          10 ORACLE              2500
         3 WARD                 20          20 JAVA                5000
```

Note: We have to use table name or alias name while check the condition otherwise we will get ambiguously error.

Ansi format:

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

       SID SNAME              CID        CID CNAME              CFEE
---------- ---------- ---------- ---------- ---------- ----------
         1 SMITH                10          10 ORACLE              2500
         2 ALLEN                10          10 ORACLE              2500
         3 WARD                 20          20 JAVA                5000
```

Rule of join: A row in a first table is comparing the given join condition with all rows of second table.

Ex: WAQ to retrieve student and corresponding course details from who selected “ORACLE” course.

Non-Ansi format:

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

       SID SNAME              CID        CID CNAME              CFEE
---------- ---------- ---------- ---------- ---------- ----------
         1 SMITH                10          10 ORACLE              2500
         2 ALLEN                10          10 ORACLE              2500
```

Ansi format:

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

       SID SNAME              CID        CID CNAME              CFEE
---------- ---------- ---------- ---------- ---------- ----------
         1 SMITH                10          10 ORACLE              2500
         2 ALLEN                10          10 ORACLE              2500
```

Ex: WAQ to display sum of salaries of department from emp, dept tables.

Non-Ansi format:

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

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

Ansi format:

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

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

Ex: WAQ to display sum of salaries of departments from EMP, DEPT tables in which department sum of salary is more than 10000?

Non-Ansi format:

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

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

Ansi format:

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

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

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:
  - Left outer join
  - Right outer join
  - Full outer join

Left Outer Join:

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

Ansi format:

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

       SID SNAME             CID        CID CNAME             CFEE
---------- ---------- ---------- ---------- ---------- ----------
         1 SMITH              10         10 ORACLE           2500
         2 ALLEN              10         10 ORACLE           2500
         3 WARD               20         20 JAVA             5000
         4 JONES              30
```

Non-Ansi format:

- when we write outer joins in non-Ansi format then we should use join operator (+).

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

       SID SNAME             CID        CID CNAME             CFEE
---------- ---------- ---------- ---------- ---------- ----------
         1 SMITH              10         10 ORACLE           2500
         2 ALLEN              10         10 ORACLE           2500
         3 WARD               20         20 JAVA             5000
         4 JONES              30
```

Right outer join:

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

Ansi format:

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

       SID SNAME             CID        CID CNAME             CFEE
---------- ---------- ---------- ---------- ---------- ----------
         1 SMITH              10         10 ORACLE           2500
         2 ALLEN              10         10 ORACLE           2500
         3 WARD               20         20 JAVA             5000
                                         40 PYTHON          45000
```

Non-Ansi format:

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

```
       SID SNAME              CID        CID CNAME              CFEE
---------- ---------- ---------- ---------- ---------- ----------
         1 SMITH              10         10 ORACLE           2500
         2 ALLEN              10         10 ORACLE           2500
         3 WARD               20         20 JAVA             5000
                                         40 PYTHON          45000
```

Full outer join:

- Retrieving matching and unmatching rows from both side tables.

Anis format:

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

       SID SNAME              CID        CID CNAME              CFEE
---------- ---------- ---------- ---------- ---------- ----------
         1 SMITH              10         10 ORACLE           2500
         2 ALLEN              10         10 ORACLE           2500
         3 WARD               20         20 JAVA             5000
         4 JONES              30
                                         40 PYTHON          45000
```

Non-Ansi format:

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

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

       SID SNAME              CID        CID CNAME              CFEE
---------- ---------- ---------- ---------- ---------- ----------
         1 SMITH              10         10 ORACLE           2500
         2 ALLEN              10         10 ORACLE           2500
         3 WARD               20         20 JAVA             5000
         4 JONES              30
                                         40 PYTHON          45000
```

Non-Equi join:

- Retrieving data from multiple tables based on any condition except "=" equal operator condition is called as non-Equi join.
- In this join we can use the following operators are <, >, <=, >=, !=, and, between, etc.

First create the below table with the give data

```
SQL> SELECT * FROM TEST1;

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

SQL> SELECT * FROM TEST2;

       SNO        SAL
---------- ----------
        10      15000
        30      23000
```

Non-Ansi format:

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

       SNO NAME              SNO        SAL
---------- ---------- ---------- ----------
        20 ALLEN              10      15000
```

Ansi format:

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

       SNO NAME              SNO        SAL
---------- ---------- ---------- ----------
        20 ALLEN              10      15000
```

Ex: WAQ to display ename, salary, low salary, high salary from emp, salgrade tables whose salary between low salary and high salary.

Non-Ansi format:

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

ENAME             SAL      LOSAL      HISAL
---------- ---------- ---------- ----------
```

```
SMITH              800         700        1200
JAMES              950         700        1200
ADAMS             1100         700        1200
WARD              1250        1201        1400
MARTIN            1250        1201        1400
MILLER            1300        1201        1400
TURNER            1500        1401        2000

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

ENAME              SAL       LOSAL      HISAL
---------- ---------- ---------- ----------
SMITH              800         700        1200
JAMES              950         700        1200
ADAMS             1100         700        1200
WARD              1250        1201        1400
MARTIN            1250        1201        1400
MILLER            1300        1201        1400
TURNER            1500        1401        200
```

Cross Join/ cartesian Join:

- 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" number of rows and a second table is having "n" number of rows then the result is “m X n” number of rows.

Ansi format:

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

       SID SNAME              CID        CID CNAME              CFEE
---------- ---------- ---------- ---------- ---------- ----------
         1 SMITH              10         10 ORACLE           2500
         2 ALLEN              10         10 ORACLE           2500
         3 WARD               20         10 ORACLE           2500
         4 JONES              30         10 ORACLE           2500
         1 SMITH              10         20 JAVA             5000
         2 ALLEN              10         20 JAVA             5000
         3 WARD               20         20 JAVA             5000
         4 JONES              30         20 JAVA             5000
         1 SMITH              10         40 PYTHON          45000
         2 ALLEN              10         40 PYTHON          45000
         3 WARD               20         40 PYTHON          45000
         4 JONES              30         40 PYTHON          45000
```

Non-Ansi format:

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

       SID SNAME              CID        CID CNAME              CFEE
---------- ---------- ---------- ---------- ---------- ----------
         1 SMITH              10         10 ORACLE           2500
         2 ALLEN              10         10 ORACLE           2500
         3 WARD               20         10 ORACLE           2500
         4 JONES              30         10 ORACLE           2500
         1 SMITH              10         20 JAVA             5000
         2 ALLEN              10         20 JAVA             5000
         3 WARD               20         20 JAVA             5000
         4 JONES              30         20 JAVA             5000
         1 SMITH              10         40 PYTHON          45000
         2 ALLEN              10         40 PYTHON          45000
         3 WARD               20         40 PYTHON          45000
         4 JONES              30         40 PYTHON          45000
```

Now create the below tables with the following data.

```
SQL> SELECT * FROM ITEMS1;

       SNO INAME           PRICE
---------- ---------- ----------
         1 PIZZA             180
         2 BURGER             80

SQL> SELECT * FROM ITEMS2;

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

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

INAME           PRICE INAME           PRICE TOTAL_AMOUNT
---------- ---------- ---------- ---------- ------------
PIZZA             180 PEPSI              25          205
PIZZA             180 COCACOLA           20          200
BURGER             80 PEPSI              25          105
BURGER             80 COCACOLA           20          100
```

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.

- By using natural join, we can retrieve date from multiple tables without duplicate columns.
- 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 name automatically.

```
SQL> SELECT * FROM STUDENT NATURAL JOIN COURSE;
       CID        SID SNAME      CNAME           CFEE
---------- ---------- ---------- ---------- ----------
        10          1 SMITH      ORACLE           2500
        10          2 ALLEN      ORACLE           2500
        20          3 WARD       JAVA             5000
```

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.
- Comparing a table data by itself is called as self-join.
- 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 will create virtual table (copy) on each alias name.
- We can create any number of alias names on a single table by each alias name should be different name.
- Without alias name we can't implement self-join mechanism.
- Self-join can be implemented at two situations:
    a. Comparing a single column value by itself with in the table.
    b. Comparing two different columns values to each other with in the table.

First create the following tables with data

```
SQL> SELECT * FROM TEST;

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

Comparing a single column value by itself with in the table:

Ex: WAQ to display employee who are working in the same location of the

employee is "SMITH" is also working.

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

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

Comparing two different columns values to each other with in the table:

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

```
SQL> SELECT M.ENAME MANAGER, E.ENAME EMPLOYEE FROM EMP E, EMP M
WHERE M.EMPNO=E.MGR;

MANAGER    EMPLOYEE
---------- ----------
JONES      FORD
JONES      SCOTT
BLAKE      TURNER
BLAKE      ALLEN
BLAKE      WARD
BLAKE      JAMES
BLAKE      MARTIN
CLARK      MILLER
SCOTT      ADAMS
KING       BLAKE
KING       JONES
KING       CLARK
FORD       SMITH
```

Ex: WAQ to display employee who are getting more than their manager salary

```
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;

MANAGER       MSALARY EMPLOYEE      ESALARY
---------- ---------- ---------- ----------
JONES            2975 FORD             3000
JONES            2975 SCOTT            3000
```

Ex: WAQ to display employee who are joined before their manager

```
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;
```

```
MANAGER    MDOJ      EMPLOYEE   EDOJ
---------- --------- ---------- ---------
BLAKE      01-MAY-81 ALLEN      20-FEB-81
BLAKE      01-MAY-81 WARD       22-FEB-81
KING       17-NOV-81 BLAKE      01-MAY-81
KING       17-NOV-81 JONES      02-APR-81
KING       17-NOV-81 CLARK      09-JUN-81
FORD       03-DEC-81 SMITH      17-DEC-80
```

First make sure you must have the following tables with corresponding data

```
SQL> SELECT * FROM STUDENT;

       SID SNAME              CID
---------- ---------- ----------
         1 SMITH              10
         2 ALLEN              10
         3 WARD               20
         4 JONES              30

SQL> SELECT * FROM COURSE;

       CID CNAME           CFEE
---------- ---------- ----------
        10 ORACLE           2500
        20 JAVA             5000
        40 PYTHON          45000

SQL> SELECT * FROM REGISTER;

     REGNO REGDATE          CID
---------- --------- ----------
      1021 24-MAR-22         10
      1022 25-MAR-22         20
      1023 26-MAR-22         50
```

How to Join more than two tables:

Syntax for non-Ansi format:

SQL> SELECT * FROM <table name1>, <table name2>, <table name3>, ….
WHERE <condition 1> AND <condition 2> AND <condition 3> AND ….;

```
SQL> SELECT S.SID, S.SNAME, C.CNAME, C.CFEE, R.REGDATE FROM STUDENT
S, COURSE C, REGISTER R WHERE S.CID=C.CID AND C.CID=R.CID;

       SID SNAME      CNAME           CFEE REGDATE
```

```
---------- ---------- ---------- ---------- ---------
         1 SMITH      ORACLE           2500 24-MAR-22
         2 ALLEN      ORACLE           2500 24-MAR-22
         3 WARD       JAVA             5000 25-MAR-22
```

Syntax for Ansi format:

SQL> SELECT * FROM <table name1> <Join key> <table name2> on <condition>
<Join key> <table name2> on <condition>
<Join key> <table name2> on <condition> …. [WHERE <condition>] ….;

```
SQL> SELECT S.SID, S.SNAME, C.CNAME, C.CFEE, R.REGDATE FROM STUDENT
S INNER JOIN COURSE C ON S.CID = C.CID INNER JOIN REGISTER R ON
C.CID = R.CID;

       SID SNAME      CNAME           CFEE REGDATE
---------- ---------- ---------- ---------- ---------
         1 SMITH      ORACLE           2500 24-MAR-22
         2 ALLEN      ORACLE           2500 24-MAR-22
         3 WARD       JAVA             5000 25-MAR-22
```

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.

```
SQL> SELECT * FROM STUDENT S INNER JOIN COURSE C USING (S.CID);
SELECT * FROM STUDENT S INNER JOIN COURSE C USING (S.CID)
                                                   *
ERROR at line 1:
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.

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

       CID        SID SNAME      CNAME           CFEE
---------- ---------- ---------- ---------- ----------
        10          1 SMITH      ORACLE           2500
        10          2 ALLEN      ORACLE           2500
        20          3 WARD       JAVA             5000
```

Q. What is the differences between joins and set operators?

Ans.

We have the following differences

Joins	Set Operators
Combined data in columns wise.	Combined data in rows wise.
Combined data horizontal	combined data vertically
Different structure tables are joined	similar structure tables are joined

Data Integrity

- To maintain accurate & consistency data in DB tables.
- This data integrity again classified into two ways those are
  - Declarative/ pre-define data integrity (using constraints)
  - Procedural/ user- define data integrity (using triggers)

Declarative/ pre -define data integrity:

- This data integrity can be implemented with help of "constraints". These are again three types,
  - Entity integrity
    - It ensures that each row uniquely identify in a table.to implement this mechanism we use primary key or unique constraint.
  - Referential integrity
    - It ensures that to create relationship between tables.to implement this mechanism then we use foreign key (referential key).
  - Domain integrity
    - Domain is nothing but column. It ensures that to check values with user define condition before accepting values into a column. Io perform this mechanism we use check, default, not null constraints.

Constraints

- Constraints are used to enforce/ 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

- Column level:
  - In this method we are defining constraints on individual columns / columns wise.
  - This type of constraints is called as simple constraints.

  Syntax:

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

- table level:
  - In this method we are defining constraints after all columns are declared/ end of the table definition.
  - This type of constraints is called as composite constraints.

  Syntax:

  SQL> CREATE TABLE <table name> (<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.

Column level:

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

Table created.

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

1 row created.

SQL> INSERT INTO TEST1 VALUES(1, 'B');
INSERT INTO TEST1 VALUES(1, 'B')
*
ERROR at line 1:
ORA-00001: unique constraint (NIRMALA.SYS_C007499) violated

SQL> INSERT INTO TEST1 VALUES(2, 'A');
INSERT INTO TEST1 VALUES(2, 'A')
*
ERROR at line 1:
ORA-00001: unique constraint (NIRMALA.SYS_C007500) violated

SQL> INSERT INTO TEST1 VALUES (2, 'B');
```

```
1 row created.

SQL> INSERT INTO TEST1 VALUES (NULL, NULL);

1 row created.
```

Table level:

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

Table created.

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

1 row created.

SQL> INSERT INTO TEST2 VALUES(1, 'B');

1 row created.

SQL> INSERT INTO TEST2 VALUES(1, 'A');
INSERT INTO TEST2 VALUES(1, 'A')
*
ERROR at line 1:
ORA-00001: unique constraint (NIRMALA.SYS_C007501) violated

SQL> INSERT INTO TEST2 VALUES(1, 'B');
INSERT INTO TEST2 VALUES(1, 'B')
*
ERROR at line 1:
ORA-00001: unique constraint (NIRMALA.SYS_C007501) violated

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

1 row created.
```

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 can be defined at column level only, but not supports table level.

```
SQL> CREATE TABLE TEST3 (SNO INT NOT NULL, NAME VARCHAR2(10) NOT
NULL);

Table created.

SQL> INSERT INTO TEST3 VALUES(1, 'A');

1 row created.

SQL> INSERT INTO TEST3 VALUES(1, 'A');

1 row created.

SQL> INSERT INTO TEST3 VALUES(NULL, 'B');
INSERT INTO TEST3 VALUES(NULL, 'B')
                         *
ERROR at line 1:
ORA-01400: cannot insert NULL into ("NIRMALA"."TEST3"."SNO")

SQL> INSERT INTO TEST3 VALUES(2, NULL);
INSERT INTO TEST3 VALUES(2, NULL)
                            *
ERROR at line 1:
ORA-01400: cannot insert NULL into ("NIRMALA"."TEST3"."NAME")

SQL> INSERT INTO TEST3 VALUES(NULL, NULL);
INSERT INTO TEST3 VALUES(NULL, NULL)
                         *
ERROR at line 1:
ORA-01400: cannot insert NULL into ("NIRMALA"."TEST3"."SNO")

SQL> CREATE TABLE TEST3 (SNO INT, NAME VARCHAR2(10), NOT NULL(SNO,
NAME));
CREATE TABLE TEST3 (SNO INT, NAME VARCHAR2(10), NOT NULL(SNO, NAME))
                                                 *
ERROR at line 1:
ORA-00904: : invalid identifier
```

Combination of unique and not null:

```
SQL> CREATE TABLE TEST4 (SNO INT UNIQUE NOT NULL, NAME VARCHAR2(10)
UNIQUE NOT NULL);

Table created.

SQL> INSERT INTO TEST4 VALUES(1, 'A');

1 row created.

SQL> INSERT INTO TEST4 VALUES(1, 'A');
```

```
INSERT INTO TEST4 VALUES(1, 'A')
*
ERROR at line 1:
ORA-00001: unique constraint (NIRMALA.SYS_C007506) violated

SQL> INSERT INTO TEST4 VALUES(NULL, NULL);
INSERT INTO TEST4 VALUES(NULL, NULL)
                         *
ERROR at line 1:
ORA-01400: cannot insert NULL into ("NIRMALA"."TEST4"."SNO")
```

Check:

- To check values with user defined condition before accepting values into a column.

Column level:

```
SQL> CREATE TABLE TEST5 (ENAME VARCHAR2(10), SAL NUMBER(10)
CHECK(SAL>5000));

Table created.

SQL> INSERT INTO TEST5 VALUES('A', 5000);
INSERT INTO TEST5 VALUES('A', 5000)
*
ERROR at line 1:
ORA-02290: check constraint (NIRMALA.SYS_C007508) violated

SQL> INSERT INTO TEST5 VALUES('A', 5001);

1 row created.
```

Table level:

```
SQL> CREATE TABLE TEST6 (ENAME VARCHAR2(10), SAL NUMBER(10),
CHECK(ENAME=LOWER(ENAME) AND SAL>5000));

Table created.

SQL> INSERT INTO TEST6 VALUES('SMITH', 5001);
INSERT INTO TEST6 VALUES('SMITH', 5001)
*
ERROR at line 1:
ORA-02290: check constraint (NIRMALA.SYS_C007509) violated

SQL> INSERT INTO TEST6 VALUES('smith', 5000);
INSERT INTO TEST6 VALUES('smith', 5000)
*
ERROR at line 1:
```

```
ORA-02290: check constraint (NIRMALA.SYS_C007509) violated

SQL> INSERT INTO TEST6 VALUES('smith', 5001);

1 row created.
```

Primary key:

- It is combination of “unique and not null”.
- To restricted duplicates & nulls into a column.
- A table should have only "one primary key".

Column level:

```
SQL> CREATE TABLE TEST7 (PCODE INT PRIMARY KEY, PNAME VARCHAR2(10)
PRIMARY KEY);
CREATE TABLE TEST7 (PCODE INT PRIMARY KEY, PNAME VARCHAR2(10)
PRIMARY KEY)
                                                              *
ERROR at line 1:
ORA-02260: table can have only one primary key

SQL> CREATE TABLE TEST7 (PCODE INT PRIMARY KEY, PNAME VARCHAR2(10));

Table created.

SQL> INSERT INTO TEST7 VALUES(1, 'A');

1 row created.

SQL> INSERT INTO TEST7 VALUES(1, 'B');
INSERT INTO TEST7 VALUES(1, 'B')
*
ERROR at line 1:
ORA-00001: unique constraint (NIRMALA.SYS_C007510) violated

SQL> INSERT INTO TEST7 VALUES(NULL, 'B');
INSERT INTO TEST7 VALUES(NULL, 'B')
                         *
ERROR at line 1:
ORA-01400: cannot insert NULL into ("NIRMALA"."TEST7"."PCODE")
```

Table level:

```
SQL> CREATE TABLE TEST8 (PCODE INT, PNAME VARCHAR2(10), PRIMARY
KEY(PCODE, PNAME));

Table created.

SQL> INSERT INTO TEST8 VALUES(1021, 'C');
```

```
1 row created.

SQL> INSERT INTO TEST8 VALUES(1021, 'C++');

1 row created.

SQL> INSERT INTO TEST8 VALUES(1021, 'C++');
INSERT INTO TEST8 VALUES(1021, 'C++')
*
ERROR at line 1:
ORA-00001: unique constraint (NIRMALA.SYS_C007511) violated

SQL> INSERT INTO TEST8 VALUES(1021, 'NULL');

1 row created.

SQL> INSERT INTO TEST8 VALUES(NULL, 'NULL');
INSERT INTO TEST8 VALUES(NULL, 'NULL')
                         *
ERROR at line 1:
ORA-01400: cannot insert NULL into ("NIRMALA"."TEST8"."PCODE")

SQL> INSERT INTO TEST8 VALUES(NULL, 'C++');
INSERT INTO TEST8 VALUES(NULL, 'C++')
                         *
ERROR at line 1:
ORA-01400: cannot insert NULL into ("NIRMALA"."TEST8"."PCODE")
```

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 rules:
    - We have a common column name (optional) but recommended.
    - Common column datatype must match.
    - One table foreign key must belong to another table primary key. And here primary key & foreign key column must be common column.
    - Primary key table is called as "parent table" and foreign key table is called as "chid table" (i.e. parent & child relationship).
    - Foreign key column values should be match with primary key

column values only.
    - Generally primary key is not allowed duplicate and null values where as foreign key is allowed duplicate & null values.

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

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

Table created.

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

1 row created.

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

1 row created.

SQL> SELECT * FROM DEPT1;

    DEPTNO DNAME
---------- ----------
        10 SALES
        20 PRODUCTION

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

Table created.

SQL> INSERT INTO EMP1 VALUES (1, 'SMITH', 10);

1 row created.

SQL> INSERT INTO EMP1 VALUES (2, 'ALLEN', 10);

1 row created.

SQL> INSERT INTO EMP1 VALUES (3, 'JONES', 20);

1 row created.

SQL> INSERT INTO EMP1 VALUES (4, 'ADAMS', NULL);

1 row created.

SQL> SELECT * FROM EMP1;
```

```
       EID ENAME          DEPTNO
---------- ---------- ----------
         1 SMITH              10
         2 ALLEN              10
         3 JONES              20
         4 ADAMS
```

- Once we establish relationship between tables there are two rules are come into picture. Those are
  - Insertion rule
  - Deletion rule.

Insertion rule:

- We cannot insert values into foreign key (references key) column those values are not existing under primary key column of parent table.

```
SQL> UPDATE EMP1 SET DEPTNO=30 WHERE DEPTNO IS NULL;
UPDATE EMP1 SET DEPTNO=30 WHERE DEPTNO IS NULL
*
ERROR at line 1:
ORA-02291: integrity constraint (NIRMALA.SYS_C007513) violated -
parent key not found
```

Deletion rule:

- We can't delete rows from a parent table those child rows are having in foreign key column child table without addressing to child.
- When we try to delete a record from parent table and those associated records are available in child table then oracle returns an error.

```
SQL> DELETE FROM DEPT1 WHERE DEPTNO=10;
DELETE FROM DEPT1 WHERE DEPTNO=10
*
ERROR at line 1:
ORA-02292: integrity constraint (NIRMALA.SYS_C007513) violated -
child record found
```

How to address to child table

- 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".
  - on delete cascade
  - on delete set null

On delete cascade:

- Whenever we are deleting a record from parent table then that associated child records are deleted from child table automatically.

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

Table created.

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

1 row created.

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

1 row created.

SQL> CREATE TABLE EMP2(EID INT, ENAME VARCHAR2(10), DEPTNO INT
REFERENCES DEPT2(DEPTNO) ON DELETE CASCADE);

Table created.

SQL> INSERT INTO EMP2 VALUES (1, 'SMITH', 10);

1 row created.

SQL> INSERT INTO EMP2 VALUES (2, 'ALLEN', 20);

1 row created.

SQL> INSERT INTO EMP2 VALUES (3, 'JONES', 30);
INSERT INTO EMP2 VALUES (3, 'JONES', 30)
*
ERROR at line 1:
ORA-02291: integrity constraint (NIRMALA.SYS_C007517) violated -
parent key not found

SQL> SELECT * FROM DEPT2;

    DEPTNO DNAME
---------- ----------
        10 SALES
        20 PRODUCTION

SQL> SELECT * FROM EMP2;

       EID ENAME          DEPTNO
---------- ---------- ----------
         1 SMITH              10
```

```
         2 ALLEN               20

SQL> DELETE FROM DEPT2 WHERE DEPTNO=10;

1 row deleted.

SQL> SELECT * FROM DEPT2;

    DEPTNO DNAME
---------- ----------
        20 PRODUCTION

SQL> SELECT * FROM EMP2;

       EID ENAME          DEPTNO
---------- ---------- ----------
         2 ALLEN              20
```

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.

```
SQL> CREATE TABLE DEPT3(DEPTNO INT PRIMARY KEY, DNAME VARCHAR2(10));

Table created.

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

1 row created.

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

1 row created.

SQL> CREATE TABLE EMP3(EID INT, ENAME VARCHAR2(10), DEPTNO INT
REFERENCES DEPT3(DEPTNO) ON DELETE SET NULL);

Table created.

SQL> INSERT INTO EMP3 VALUES (1, 'SMITH', 10);

1 row created.

SQL> INSERT INTO EMP3 VALUES (2, 'ALLEN', 20);

1 row created.

SQL> INSERT INTO EMP3 VALUES (3, 'JONES', 30);
```

```
INSERT INTO EMP3 VALUES (3, 'JONES', 30)
*
ERROR at line 1:
ORA-02291: integrity constraint (NIRMALA.SYS_C007525) violated -
parent key not found

SQL> SELECT * FROM DEPT3;

    DEPTNO DNAME
---------- ----------
        10 SALES
        20 PRODUCTION

SQL> SELECT * FROM EMP3;

       EID ENAME          DEPTNO
---------- ---------- ----------
         1 SMITH              10
         2 ALLEN              20

SQL> DELETE FROM DEPT3 WHERE DEPTNO=10;

1 row deleted.

SQL> SELECT * FROM DEPT3;

    DEPTNO DNAME
---------- ----------
        20 PRODUCTION

SQL> SELECT * FROM EMP3;

       EID ENAME          DEPTNO
---------- ---------- ----------
         1 SMITH
         2 ALLEN              20
```

Default:

- It a special type of constraint which is used to assign a user define default value to a column.

```
SQL> CREATE TABLE TEST10(SNAME VARCHAR2(10), SFEE NUMBER(10) DEFAULT
500);

Table created.

SQL> INSERT INTO TEST10 VALUES ('SMITH', 1500);
```

```
1 row created.

SQL> INSERT INTO TEST10 (SNAME) VALUES ('SMITH');

1 row created.

SQL> SELECT * FROM TEST10;

SNAME            SFEE
---------- ----------
SMITH            1500
SMITH             500
```

Data Dictionaries (or) Read Only tables

- Whenever we are installing oracle s/w internally some pre-define tables created by oracle server. These pre-defined tables are called as "data dictionaries".
- These data dictionaries are used to store the information which is related to table such as constraints, views, synonyms, sequences, indexes 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.

To view all data dictionaries in oracle DB:

Syntax:

SQL> SELECT * FROM DICT;

First create the below table

```
SQL> CREATE TABLE TEST11(SNO INT PRIMARY KEY, SNAME VARCHAR(20)
UNIQUE, SAL NUMBER(10) CHECK(SAL>8000));

Table created.
```

- If we want to view constraint name along with column name of a particular table then we have to use "USER_CONS_COLUMNS" data dictionary/ table.

```
SQL> DESC USER_CONS_COLUMNS;

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

```
 OWNER                          NOT NULL       VARCHAR2(128)
 CONSTRAINT_NAME                NOT NULL       VARCHAR2(128)
 TABLE_NAME                     NOT NULL       VARCHAR2(128)
 COLUMN_NAME                                   VARCHAR2(4000)
 POSITION                                      NUMBER

SQL> SELECT COLUMN_NAME, CONSTRAINT_NAME FROM USER_CONS_COLUMNS
WHERE TABLE_NAME='TEST11';

COLUMN_NAME                    CONSTRAINT_NAME
------------------------------ ------------------------------
SAL                            SYS_C007530
SNO                            SYS_C007531
SNAME                          SYS_C007532
```

Note: You can see the constrains name are note under stable because of system generated. To get a proper constraint name we have to give user define constraint name.

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:
  SQL> CREATE TABLE <table name> (<column name> <datatype> [size] CONSTRAINT <user defined constraint name> <constraint type>, <column name> <datatype> [size] CONSTRAINT <user defined constraint name> <constraint type>, ….);

```
SQL> CREATE TABLE TEST12(SNO INT CONSTRAINT PK_NO PRIMARY KEY, SNAME
VARCHAR (20) CONSTRAINT UQ_SNAME UNIQUE, SAL NUMBER (10) CONSTRAINT
CHK_SAL CHECK(SAL>8000));

Table created.

SQL> SELECT COLUMN_NAME, CONSTRAINT_NAME FROM USER_CONS_COLUMNS
WHERE TABLE_NAME='TEST12';

COLUMN_NAME                    CONSTRAINT_NAME
------------------------------ ------------------------------
SAL                            CHK_SAL
SNO                            PK_NO
SNAME                          UQ_SNAME
```

- If we want to view all constraints information of a particular table then

we use "USER_CONSTRAINTS" data dictionary/ table.

```
SQL> DESC USER_CONSTRAINTS;

SQL> SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE FROM USER_CONSTRAINTS
WHERE TABLE_NAME='TEST12';

CONSTRAINT_NAME                C
---------------------------    -
CHK_SAL                        C
PK_NO                          P
UQ_SNAME                       U
```

- To view a logical condition of check constraint then we need to call "SEARCH_CONDITION" column from "USER_CONSTRAINTS" data dictionary/ table.

```
SQL> SELECT SEARCH_CONDITION FROM USER_CONSTRAINTS WHERE
TABLE_NAME='TEST12';

SEARCH_CONDITION
---------------------------------
SAL>8000
```

- To view all columns information of a particular table then we use "USER_TAB_COLUMNS" data dictionary/ table.

```
SQL> DESC USER_TAB_COLUMNS;

SQL> SELECT COLUMN_NAME FROM USER_TAB_COLUMNS WHERE
TABLE_NAME='EMP';

COLUMN_NAME
-------------------------------------------------
EMPNO
ENAME
JOB
MGR
HIREDATE
SAL
COMM
DEPTNO

8 rows selected.
```

- To view the information about privilege and also granter then we use

"USER_TAB_PRIVS_MADE" data dictionary.

```
SQL> DESC USER_TAB_PRIVS_MADE;

SELECT GRANTEE, TABLE_NAME, GRANTOR, PRIVILEGE FROM
USER_TAB_PRIVS_MADE;
```

- To view the information about privilege and also who received permission (grantee) then we have to use “USER_TAB_PRIVS_RECD” data dictionary " ".

```
SQL> DESC USER_TAB_PRIVS_RECD;

SQL> SELECT GRANTOR, PRIVILEGE, TABLE_NAME FROM USER_TAB_PRIVS_RECD;
```

- To view system privileges related to role then we are using the following data dictionary is "ROLE_SYS_PRIVS".

```
SQL> DESC ROLE_SYS_PRIVS;

SQL> SELECT ROLE, PRIVILEGE FROM ROLE_SYS_PRIVS WHERE ROLE='R1';

ROLE                      PRIVILEGE
--------------------      ------------------
R1                        CREATE TABLE
```

- To view object privileges related to role then we use the following data dictionary is "ROLE_TAB_PRIVS"

```
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
```

Q. How to find number of rows in a table?

Ans.

```
SQL> SELECT COUNT (*) FROM EMP;

  COUNT(*)
----------
        14
```

Q. How to find number of columns in a table?

Ans.

```
SQL> SELECT COUNT (*) FROM USER_TAB_COLUMNS WHERE TABLE_NAME='EMP';

  COUNT(*)
----------
         8
```

How to add constraints to an existing table:

Syntax:

SQL> ALTERE TABLE <table name> ADD CONSTRAINT <constraint key name> <constraint type> (<column name>);

First create the below table.

```
SQL> CREATE TABLE PARENT(EID INT, NAME VARCHAR2(10), SAL
NUMBER(10));
Table created.
```

Adding primary key:

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

Table altered.
```

Adding unique constraint:

```
SQL> ALTER TABLE PARENT ADD CONSTRAINT UQ_NAME UNIQUE(NAME);

Table altered.
```

Add check constraint:

```
SQL> ALTER TABLE PARENT ADD CONSTRAINT CHK_SAL CHECK(SAL>8000);

Table altered.
```

Adding "not null" constraint:

Syntax:

SQL> ALTER TABLE <table name> MODIFY <column name> CONSTRAINT <constraint key name> NOT NULL;

```
SQL> ALTER TABLE PARENT MODIFY NAME CONSTRAINT NN_NAME NOT NULL;

Table altered.
```

Adding foreign key constraint:

Syntax:

SQL> ALTER TABLE <table name> 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;

```
SQL> CREATE TABLE CHILD (NAME VARCHAR(10), EID INT);

Table created.

SQL> ALTER TABLE CHILD ADD CONSTRAINT FK_EID FOREIGN KEY(EID)
REFERENCES PARENT(EID) ON DELETE CASCADE;

Table altered.
```

How to drop constraint from an existing table:

Syntax:

SQL> ALTER TABLE <table name> DROP CONSTRAINT <constraint key name>;

Dropping primary key:

- When we drop primary key along with foreign key constraint from parent and child tables then we use "cascade" statement.

```
SQL> ALTER TABLE PARENT DROP CONSTRAINT PK_EID;
ALTER TABLE PARENT DROP CONSTRAINT PK_EID
                                   *
ERROR at line 1:
ORA-02273: this unique/primary key is referenced by some foreign
keys

SQL> ALTER TABLE PARENT DROP CONSTRAINT PK_EID CASCADE;

Table altered.
```

- Otherwise first delete the foreign key constraint for parent the primary key from parent table.

```
SQL> ALTER TABLE CHILD DROP CONSTRAINT FK_EID;

Table altered.

SQL> ALTER TABLE PARENT DROP CONSTRAINT PK_EID;

Table altered.
```

How to rename constraint name:

Syntax:

SQL> ALTER TABLE <table name> RENAME CONSTRAINT <old constraint name> TO <new constraint name>;

```
SQL> CREATE TABLE TEST(SNO INT PRIMARY KEY);

Table created.

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

COLUMN_NAME                    CONSTRAINT_NAME
------------------------------ ------------------------------
SNO                            SYS_C007541

SQL> ALTER TABLE TEST RENAME CONSTRAINT SYS_C007541 TO PK_SNO;

Table altered.

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

COLUMN_NAME                    CONSTRAINT_NAME
------------------------------ ------------------------------
SNO                            PK_SNO
```

How disable / enable a constraint:

- By default, constraints are enabling mode. If we want to disable constraint temporarily then we have to 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:

SQL> ALTER TABLE <table name> DISABLE / ENABLE CONSTRAINT <constraint key name>;

```
SQL> CREATE TABLE TEST1(SAL NUMBER(10) CONSTRAINT CHK_SAL
CHECK(SAL>10000));

Table created.
```

```
SQL> INSERT INTO TEST1 VALUES(5000);
INSERT INTO TEST1 VALUES(5000)
*
ERROR at line 1:
ORA-02290: check constraint (NIRMALA.CHK_SAL) violated


SQL> INSERT INTO TEST1 VALUES(12000);

1 row created.

SQL> SELECT * FROM TEST1;

       SAL
----------
     12000

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

COLUMN_NAME                      CONSTRAINT_NAME
-------------------------------  ------------------------------
SAL                              CHK_SAL

SQL> ALTER TABLE TEST1 DISABLE CONSTRAINT CHK_SAL;

Table altered.

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

COLUMN_NAME                      CONSTRAINT_NAME
-------------------------------  ------------------------------
SAL                              CHK_SAL

SQL> INSERT INTO TEST1 VALUES(5000);

1 row created.

SQL> ALTER TABLE TEST1 ENABLE CONSTRAINT CHK_SAL;
ALTER TABLE TEST1 ENABLE CONSTRAINT CHK_SAL
                                    *
ERROR at line 1:
ORA-02293: cannot validate (NIRMALA.CHK_SAL) - check constraint
violated
```

- If we try to enable that constraint, will get an error. To overcome the

above problem then we use "NOVALIDATE" keyword at the time of enable constraint.

NOVALIDATE:

- This constraint means not checking existing data in a column but newly inserted data will check.

```
SQL> ALTER TABLE TEST1 ENABLE NOVALIDATE CONSTRAINT CHK_SAL;

Table altered.

SQL> INSERT INTO TEST1 VALUES(5000);
INSERT INTO TEST1 VALUES(5000)
*
ERROR at line 1:
ORA-02290: check constraint (NIRMALA.CHK_SAL) violated

SQL> INSERT INTO TEST1 VALUES(15000);

1 row created.
```

How to add default value to an existing table:

Syntax:

SQL> ALTER TABLE <table name> MODIFY <column name> DEFAULT <value/ expression>;

```
SQL> CREATE TABLE TEST2(EID INT, SAL NUMBER (10));

Table created.

SQL> INSERT INTO TEST2 (EID) VALUES (1);

1 row created.

SQL> SELECT * FROM TEST2;

       EID        SAL
---------- ----------
         1

SQL> ALTER TABLE TEST2 MODIFY SAL DEFAULT 8000;

Table altered.

SQL> INSERT INTO TEST2 (EID) VALUES (2);
```

```
1 row created.

SQL> SELECT * FROM TEST2;

       EID        SAL
---------- ----------
         1
         2       8000
```

Note: If we want to view default value of a column then we use "USER_TAB_COLUMNS" data dictionary/ table.

```
SQL> DESC USER_TAB_COLUMNS;

SQL> SELECT COLUMN_NAME, DATA_DEFAULT FROM USER_TAB_COLUMNS WHERE
TABLE_NAME='TEST2';

COLUMN_NAME                    DATA_DEFAULT
------------------------------ ------------------------------
EID
SAL                            8000
```

How to remove default value of a column:

```
SQL> ALTER TABLE TEST2 MODIFY SAL DEFAULT NULL;

Table altered.

SQL> SELECT * FROM TEST2;

       EID        SAL
---------- ----------
         1
         2       8000

SQL> SELECT COLUMN_NAME, DATA_DEFAULT FROM USER_TAB_COLUMNS WHERE
TABLE_NAME='TEST2';

COLUMN_NAME                    DATA_DEFAULT
------------------------------ ------------------------------
EID
SAL                            NULL
```

Subquery/ Nested Query

- A query inside another query is called as subquery or nested query.
- A subquery is having two more queries those are,

- Inner/ child/ sub query
- Outer/ parent/ main query

Syntax:

SQL> SELECT * FROM <table name> WHERE <condition> (SELECT * FROM ……);

- As per the execution process of subquery it again classified into two categorised.
  - Non-corelated subqueries
  - Corelated subqueries

Non-Corelated Subqueries

- In non-corelated 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.
- This is default technique.
- There are different types of non-corelated subqueries,
  - Single row subquery
  - Multiple row subquery
  - Multiple column subquery
  - Inline view

Single row subquery:

- When a subquery returns a single value is called as Single row subquery.
- In single row subquery we can use operators are =, <, >, <=, >=, !=.

Ex: WAQ to display the first high salary from emp table.

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

  MAX(SAL)
----------
      5000

SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE SAL=(SELECT MAX(SAL) FROM
EMP);

ENAME      JOB              SAL
---------- --------- ----------
KING       PRESIDENT       5000
```

Ex: WAQ to display employee whose job is same as the job of 'SMITH'.

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

JOB
---------
CLERK

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

ENAME      JOB              SAL
---------- --------- ----------
SMITH      CLERK             800
ADAMS      CLERK            1100
JAMES      CLERK             950
MILLER     CLERK            1300
```

Ex: WAQ to display employee whose salary is more than maximum salary of a "SALESMAN"?

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

  MAX(SAL)
----------
      1600

SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE SAL>(SELECT MAX(SAL) FROM
EMP WHERE JOB='SALESMAN');

ENAME      JOB              SAL
---------- --------- ----------
JONES      MANAGER          2975
BLAKE      MANAGER          2850
CLARK      MANAGER          2450
SCOTT      ANALYST          3000
KING       PRESIDENT        5000
FORD       ANALYST          3000
```

Ex: WAQ to display the senior most employee detail from emp table.

```
SQL> SELECT MIN(HIREDATE) FROM EMP;

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

```
SQL> SELECT ENAME, JOB, HIREDATE FROM EMP WHERE HIREDATE=(SELECT
MIN(HIREDATE) FROM EMP);

ENAME      JOB       HIREDATE
---------- --------- ---------
SMITH      CLERK     17-DEC-80
```

Ex: WAQ to display whose employee job is same as the job of "BLAKE" and who are earning salary more than "BLAKE" salary.

```
SQL> SELECT JOB FROM EMP WHERE ENAME='BLAKE';

JOB
---------
MANAGER

SQL> SELECT SAL FROM EMP WHERE ENAME='BLAKE';

       SAL
----------
      2850

SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE JOB=(SELECT JOB FROM EMP
WHERE ENAME='BLAKE') AND SAL>(SELECT SAL FROM EMP WH
ERE ENAME='BLAKE');

ENAME      JOB              SAL
---------- --------- ----------
JONES      MANAGER         2975
```

Ex: WAQ To find the second high salary from emp table.

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

  MAX(SAL)
----------
      3000
```

Ex: WAQ to display second highest salary employee details from emp table.

```
SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE SAL=(SELECT MAX(SAL) FROM
EMP WHERE SAL<(SELECT MAX(SAL) FROM EMP));

ENAME      JOB              SAL
---------- --------- ----------
SCOTT      ANALYST         3000
FORD       ANALYST         3000
```

Ex: WAQ to display 3rd highest salary employee details from emp tab.

```
SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE SAL=(SELECT MAX(SAL) FROM
EMP WHERE SAL<(SELECT MAX(SAL) FROM EMP WHERE SAL<(
SELECT MAX(SAL) FROM EMP)));

ENAME      JOB              SAL
---------- --------- ----------
JONES      MANAGER         2975
```

Ex: WAQ to display number of employees of department numbers in which deptno number of employees is less than the number of employees of deptno is 20.

```
SQL> SELECT COUNT(*) FROM EMP WHERE DEPTNO=20;

  COUNT(*)
----------
         5

SQL> SELECT DEPTNO, COUNT(*) FROM EMP GROUP BY DEPTNO;

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

SQL> SELECT DEPTNO, COUNT(*) FROM EMP GROUP BY DEPTNO HAVING
COUNT(*)<(SELECT COUNT(*) FROM EMP WHERE DEPTNO=20);

    DEPTNO   COUNT(*)
---------- ----------
        10          3
```

Ex: WAQ to display sum of salary of jobs if sum of salary of jobs are more than sum of salary of the job is 'CLERK'?

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

  SUM(SAL)
----------
      4150

SQL> SELECT JOB, SUM(SAL) FROM EMP GROUP BY JOB;

JOB         SUM(SAL)
```

```
--------- ----------
CLERK           4150
SALESMAN        5600
ANALYST         6000
MANAGER         8275
PRESIDENT       5000

SQL> SELECT JOB, SUM(SAL) FROM EMP GROUP BY JOB HAVING
SUM(SAL)>(SELECT SUM(SAL) FROM EMP WHERE JOB='CLERK');

JOB         SUM(SAL)
--------- ----------
SALESMAN        5600
ANALYST         6000
MANAGER         8275
PRESIDENT       5000
```

Subquery with "update":

Ex: WAQ to update employee salary with maximum salary of Emp table whose EMPNO is 7900.

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

  MAX(SAL)
----------
      5000

SQL> UPDATE EMP SET SAL=(SELECT MAX(SAL) FROM EMP) WHERE EMPNO=7900;

1 row updated.
```

Subquery with "delete":

Ex: WAQ to delete employee details from emp table whose job is same as the job of 'SCOTT'?

```
SQL> SELECT JOB FROM EMP WHERE ENAME='SCOTT';

JOB
---------
ANALYST

SQL> DELETE FROM EMP WHERE JOB=(SELECT JOB FROM EMP WHERE
ENAME='SCOTT');

2 rows deleted.
```

Multiple row subquery:

- When a subquery returns more than one value is called as Multiple row subquery.
- In this subquery we can use the operators are in, any, all.

Ex: WAQ to display employee details whose employee job is same as the job of the employee "SMITH", "JONES”?

```
SQL> SELECT JOB FROM EMP WHERE ENAME='SMITH' OR ENAME='JONES';

JOB
---------
CLERK
MANAGER

SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE JOB IN(SELECT JOB FROM
EMP WHERE ENAME='SMITH' OR ENAME='JONES');

ENAME      JOB              SAL
---------- --------- ----------
SMITH      CLERK            800
ADAMS      CLERK           1100
JAMES      CLERK            950
MILLER     CLERK           1300
JONES      MANAGER         2975
BLAKE      MANAGER         2850
CLARK      MANAGER         2450

SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE JOB IN(SELECT JOB FROM
EMP WHERE ENAME IN ('SMITH', 'JONES'));

ENAME      JOB              SAL
---------- --------- ----------
SMITH      CLERK            800
ADAMS      CLERK           1100
JAMES      CLERK            950
MILLER     CLERK           1300
JONES      MANAGER         2975
BLAKE      MANAGER         2850
CLARK      MANAGER         2450
```

Ex: WAQ to display employees who are earning highest salary from each job wise.

```
SQL> SELECT JOB, MAX(SAL) FROM EMP GROUP BY JOB;
```

```
JOB         MAX(SAL)
--------- ----------
CLERK           1300
SALESMAN        1600
ANALYST         3000
MANAGER         2975
PRESIDENT       5000

SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE SAL IN(SELECT MAX(SAL)
FROM EMP GROUP BY JOB);

ENAME      JOB              SAL
---------- --------- ----------
ALLEN      SALESMAN        1600
JONES      MANAGER         2975
SCOTT      ANALYST         3000
KING       PRESIDENT       5000
FORD       ANALYST         3000
MILLER     CLERK           1300
```

Ex: WAQ to display employee details who are getting min, max salaries?

```
SQL> SELECT MAX(SAL), MIN(SAL) FROM EMP;

  MAX(SAL)   MIN(SAL)
---------- ----------
      5000        800

SQL> SELECT MAX(SAL) FROM EMP UNION SELECT MIN(SAL) FROM EMP;

  MAX(SAL)
----------
       800
      5000

SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE SAL IN(SELECT MAX(SAL)
FROM EMP UNION SELECT MIN(SAL) FROM EMP);

ENAME      JOB              SAL
---------- --------- ----------
SMITH      CLERK            800
KING       PRESIDENT       5000
```

Ex: WAQ to display the senior most employees from each deptno wise.

```
SQL> SELECT JOB, MIN(HIREDATE) FROM EMP GROUP BY JOB;

JOB       MIN(HIRED
```

```
--------- ---------
CLERK     17-DEC-80
SALESMAN  20-FEB-81
ANALYST   03-DEC-81
MANAGER   02-APR-81
PRESIDENT 17-NOV-81

SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE HIREDATE IN(SELECT
MIN(HIREDATE) FROM EMP GROUP BY JOB);

ENAME      JOB              SAL
---------- --------- ----------
SMITH      CLERK            800
ALLEN      SALESMAN        1600
JONES      MANAGER         2975
KING       PRESIDENT       5000
JAMES      CLERK            950
FORD       ANALYST         3000
```

Working with "any", "all" operators:

Any:

- It returns a value if anyone value is true from give group of values in the list.
- Ex: X > any (10,20,30)
  - If X = 40 – true
  - If X = 09 – false
  - If X = 25 – true

All:

- It returns a value if all values are true form give group of values in the list.
- Ex: X > all (10,20,30)
  - If X = 40 – true
  - If X = 09 – false
  - If X = 25 - false

Ex: WAQ to display employees whose salary is more than any "SALESMAN" salary.

```
SQL> SELECT SAL FROM EMP WHERE JOB='SALESMAN';

       SAL
----------
      1600
      1250
```

```
      1250
      1500

SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE SAL > ANY(SELECT SAL FROM
EMP WHERE JOB='SALESMAN');

ENAME      JOB              SAL
---------- --------- ----------
KING       PRESIDENT       5000
FORD       ANALYST         3000
SCOTT      ANALYST         3000
JONES      MANAGER         2975
BLAKE      MANAGER         2850
CLARK      MANAGER         2450
ALLEN      SALESMAN        1600
TURNER     SALESMAN        1500
MILLER     CLERK           1300
```

Ex: WAQ to display employees whose salary is more than of all "SALESMAN" salary?

```
SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE SAL > ALL(SELECT SAL FROM
EMP WHERE JOB='SALESMAN');

ENAME      JOB              SAL
---------- --------- ----------
CLARK      MANAGER         2450
BLAKE      MANAGER         2850
JONES      MANAGER         2975
SCOTT      ANALYST         3000
FORD       ANALYST         3000
KING       PRESIDENT       5000
```

Ex: WAQ to display employees who are earning highest salary from each job wise by using multiple row subquery?

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

ENAME      JOB              SAL
---------- --------- ----------
ALLEN      SALESMAN        1600
JONES      MANAGER         2975
SCOTT      ANALYST         3000
KING       PRESIDENT       5000
FORD       ANALYST         3000
MILLER     CLERK           1300
```

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 Multiple column subquery.

  Syntax:
  SQL> SELECT * FROM <table name> WHERE (<column 1>, <column 2>, ...) IN (SELECT <column 1>, <column 2>, ... FROM <table name> ....);

Ex: WAQ to display employees who are earning highest salary from each job.

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

ENAME      JOB              SAL
---------- --------- ----------
MILLER     CLERK           1300
ALLEN      SALESMAN        1600
SCOTT      ANALYST         3000
FORD       ANALYST         3000
JONES      MANAGER         2975
KING       PRESIDENT       5000
```

Ex: WAQ to display employee whose JOB, MGR are same as the JOB, MGR of the employee "scott".

```
SQL> SELECT JOB, MGR FROM EMP WHERE ENAME='SCOTT';

JOB              MGR
--------- ----------
ANALYST         7566

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

ENAME      JOB              MGR
---------- --------- ----------
SCOTT      ANALYST         7566
FORD       ANALYST         7566
```

Ex: WAQ to display employee whose JOB, SAL are same as the JOB, SAL of the employee 'WARD'.

```
SQL> SELECT JOB, SAL FROM EMP WHERE ENAME='WARD';

JOB              SAL
--------- ----------
SALESMAN        1250

SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE (JOB, SAL) IN (SELECT
JOB, SAL FROM EMP WHERE ENAME='WARD');

ENAME      JOB              SAL
---------- --------- ----------
WARD       SALESMAN        1250
MARTIN     SALESMAN        1250
```

Pseudo columns:

- Pseudo column is just like a table column.
- There are two pseudo columns
  - ROWID
  - ROWNUM

ROWID:

- Whenever we insert a new row into a table internally system is creating a unique id address/ number for each row wise automatically.
- These ROWIDs are stored in database so that these are permanent Id's.

```
SQL> SELECT ROWID, ENAME FROM EMP;

ROWID              ENAME
------------------ ----------
AAAR5hAAHAAAAHFAAA SMITH
AAAR5hAAHAAAAHFAAB ALLEN

SQL> SELECT ROWID, ENAME, DEPTNO FROM EMP WHERE DEPTNO=20;

ROWID              ENAME          DEPTNO
------------------ ---------- ----------
AAAR5hAAHAAAAHFAAA SMITH              20
AAAR5hAAHAAAAHFAAD JONES              20

SQL> SELECT MAX(ROWID) FROM EMP;

MAX(ROWID)
------------------
AAAR5hAAHAAAAHFAAN

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

```
MIN(ROWID)
------------------
AAAR5hAAHAAAAHFAAA
```

Q. How to delete multiple duplicate rows except one duplicate row from a table?

Ans. Whenever we want to delete multiple duplicate rows except one duplicate row from a table then we have to use "ROWID" pseudo column.

```
SQL> SELECT * FROM TEST;

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

SQL> SELECT MAX(ROWID), SNO, COUNT(*) FROM TEST GROUP BY SNO;

MAX(ROWID)                SNO   COUNT(*)
------------------ ---------- ----------
AAASEkAAHAAAAFzAAC          1          3
AAASEkAAHAAAAFzAAE          2          2
AAASEkAAHAAAAFzAAI          4          3
AAASEkAAHAAAAFzAAK          5          2
AAASEkAAHAAAAFzAAF          3          1

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

6 rows deleted.

SQL> SELECT MAX(ROWID), SNO, COUNT(*) FROM TEST GROUP BY SNO;

MAX(ROWID)                SNO   COUNT(*)
------------------ ---------- ----------
AAASEkAAHAAAAFzAAC          1          1
AAASEkAAHAAAAFzAAE          2          1
```

```
AAASEkAAHAAAAFzAAI          4          1
AAASEkAAHAAAAFzAAK          5          1
AAASEkAAHAAAAFzAAF          3          1
```

ROWNUM:

- To generate row numbers to each row wise/ group of rows wise automatically.
- These row numbers are not saved in DB, so that these are temporary numbers.
- This ROWNUM is used to perform “Nth / Top N” operation on table rows.

```
SQL> SELECT ROWNUM, ENAME FROM EMP;

    ROWNUM ENAME
---------- ----------
         1 SMITH
         2 ALLEN
         3 WARD

SQL> SELECT ROWNUM, ENAME, DEPTNO FROM EMP WHERE DEPTNO=10;

    ROWNUM ENAME          DEPTNO
---------- ---------- ----------
         1 CLARK              10
         2 KING               10
         3 MILLER             10
```

Ex: WAQ to fetch the first-row employee details from emp table by using ROWNUM?

```
SQL> SELECT ROWNUM, ENAME, JOB, SAL FROM EMP WHERE ROWNUM=1;

    ROWNUM ENAME      JOB              SAL
---------- ---------- --------- ----------
         1 SMITH      CLERK            800
```

Ex: WAQ to fetch the second-row employee details from emp Table by using ROWNUM?

```
SQL> SELECT ROWNUM, ENAME, JOB, SAL FROM EMP WHERE ROWNUM=2;

no rows selected
```

Note: Generally, ROWNUM is always start with 1 from every selected row in a

table. So, to avoid this problem then we use <, <= operators.

```
SQL> SELECT ROWNUM, ENAME, JOB, SAL FROM EMP WHERE ROWNUM<=2;

    ROWNUM ENAME      JOB              SAL
---------- ---------- --------- ----------
         1 SMITH      CLERK             800
         2 ALLEN      SALESMAN         1600

SQL> SELECT ROWNUM, ENAME, JOB, SAL FROM EMP WHERE ROWNUM<=2
  2  MINUS
  3  SELECT ROWNUM, ENAME, JOB, SAL FROM EMP WHERE ROWNUM=1;

    ROWNUM ENAME      JOB              SAL
---------- ---------- --------- ----------
         2 ALLEN      SALESMAN         1600
```

Ex: WAQ to fetch the first five rows from emp table by using ROWNUM.

```
SQL> SELECT ROWNUM, ENAME, JOB, SAL FROM EMP WHERE ROWNUM<=5;

    ROWNUM ENAME      JOB              SAL
---------- ---------- --------- ----------
         1 SMITH      CLERK             800
         2 ALLEN      SALESMAN         1600
         3 WARD       SALESMAN         1250
         4 JONES      MANAGER          2975
         5 MARTIN     SALESMAN         1250
```

Ex: WAQ to fetch the fifth-row employee details from emp table by using ROWNUM.

```
SQL> SELECT ROWNUM, ENAME, JOB, SAL FROM EMP WHERE ROWNUM<=5
  2  MINUS
  3  SELECT ROWNUM, ENAME, JOB, SAL FROM EMP WHERE ROWNUM<=4;

    ROWNUM ENAME      JOB              SAL
---------- ---------- --------- ----------
         5 MARTIN     SALESMAN         1250
```

Ex: WAQ to fetch from 3rd to 9th row from emp table by using ROWNUM.

```
SQL> SELECT ROWNUM, ENAME, JOB, SAL FROM EMP WHERE ROWNUM<=9
  2  MINUS
  3  SELECT ROWNUM, ENAME, JOB, SAL FROM EMP WHERE ROWNUM<=3;

    ROWNUM ENAME      JOB              SAL
```

```
---------- ---------- --------- ----------
         4 JONES      MANAGER         2975
         5 MARTIN     SALESMAN        1250
         6 BLAKE      MANAGER         2850
         7 CLARK      MANAGER         2450
         8 SCOTT      ANALYST         3000
         9 KING       PRESIDENT       5000
```

Ex: WAQ to fetch the last two rows from emp table by ROWNUM?

```
SQL> SELECT ROWNUM, ENAME, JOB, SAL FROM EMP WHERE ROWNUM<=14
  2  MINUS
  3  SELECT ROWNUM, ENAME, JOB, SAL FROM EMP WHERE ROWNUM<=12;

    ROWNUM ENAME      JOB              SAL
---------- ---------- --------- ----------
        13 FORD       ANALYST         3000
        14 MILLER     CLERK           1300

SQL> SELECT ROWNUM, ENAME, JOB, SAL FROM EMP
  2  MINUS
  3  SELECT ROWNUM, ENAME, JOB, SAL FROM EMP WHERE ROWNUM<=(SELECT
COUNT(*)-2 FROM EMP);

    ROWNUM ENAME      JOB              SAL
---------- ---------- --------- ----------
        13 FORD       ANALYST         3000
        14 MILLER     CLERK           1300
```

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:

SQL> SELECT * FROM (<select query>);

Note:

- ✓ Generally, subquery is not allowed to use “order by" clause. So that we have to use "inline view".
- ✓ Generally, column alias names are not allowed to use in "where" clause condition. So that we have to use "inline view ".

Using column alias names in where clause condition:

Ex: WAQ to display employee whose employee annual salary is more than 25000.

```
SQL> SELECT * FROM (SELECT ENAME, SAL, SAL*12 ANNUAL_SAL FROM EMP)
WHERE ANNUAL_SAL>25000;

ENAME             SAL ANNUAL_SAL
---------- ---------- ----------
JONES            2975      35700
BLAKE            2850      34200
CLARK            2450      29400
SCOTT            3000      36000
KING             5000      60000
FORD             3000      36000
```

Using "order by" clause in subquery:

Ex: WAQ to display first 5 highest salaries of employee from emp table by using ROWNUM along with inline view.

```
SQL> SELECT ENAME, JOB, SAL FROM (SELECT * FROM EMP ORDER BY SAL
DESC) WHERE ROWNUM<=5;

ENAME      JOB              SAL
---------- --------- ----------
KING       PRESIDENT       5000
SCOTT      ANALYST         3000
FORD       ANALYST         3000
JONES      MANAGER         2975
BLAKE      MANAGER         2850
```

Ex: WAQ to display 5th highest salary of employee from emp table by using ROWNUM along with inline view.

```
SQL> SELECT ENAME, JOB, SAL FROM (SELECT * FROM EMP ORDER BY SAL
DESC) WHERE ROWNUM<=5
  2  MINUS
  3  SELECT ENAME, JOB, SAL FROM (SELECT * FROM EMP ORDER BY SAL
DESC) WHERE ROWNUM<=4;

ENAME      JOB              SAL
---------- --------- ----------
BLAKE      MANAGER         2850
```

Using ROWNUM alias name:

Ex: WAQ to display 3rd position row from emp table by using ROWNUM alias

name along with inline view.

```
SQL> SELECT ENAME, JOB, SAL FROM (SELECT ROWNUM R, EMP.* FROM EMP
ORDER BY SAL DESC) WHERE R=3;

ENAME      JOB              SAL
---------- --------- ----------
WARD       SALESMAN        1250
```

Ex: WAQ to display 1st, 3rd, 5th, 7th, 9th rows from emp table by using ROWNUM alias name along with inline view.

```
SQL> SELECT ENAME, JOB, SAL FROM (SELECT ROWNUM R, EMP.* FROM EMP
ORDER BY SAL DESC) WHERE R IN (1,3,5,7,9);

ENAME      JOB              SAL
---------- --------- ----------
KING       PRESIDENT       5000
CLARK      MANAGER         2450
WARD       SALESMAN        1250
MARTIN     SALESMAN        1250
SMITH      CLERK            800
```

Ex: WAQ to display even position rows from emp table by using ROWNUM Alias name along with inline view.

```
SQL> SELECT ENAME, JOB, SAL FROM (SELECT ROWNUM R, EMP.* FROM EMP
ORDER BY SAL DESC) WHERE MOD(R,2)=0;

ENAME      JOB              SAL
---------- --------- ----------
SCOTT      ANALYST         3000
JONES      MANAGER         2975
BLAKE      MANAGER         2850
ALLEN      SALESMAN        1600
TURNER     SALESMAN        1500
MILLER     CLERK           1300
JAMES      CLERK            950
```

Ex: WAQ to display first row and last row from emp table by using ROWNUM alias name along with inline view?

```
SQL> SELECT ENAME, JOB, SAL FROM (SELECT ROWNUM R, EMP.* FROM EMP
ORDER BY SAL DESC) WHERE R=1 OR R=(SELECT COUNT(*) FRO
M EMP);
```

```
ENAME      JOB              SAL
---------- --------- ----------
MILLER     CLERK           1300
SMITH      CLERK            800
```

Analytical Function

- Oracle supporting the following three types of analytical functions those are,
  - ROW_NUMBER ()
  - RANK ()
  - DENSE_RANK ()
- These analytical functions are automatically generating ranking numbers to each row wise (or) group of rows wise except ROW_NUMBER ().

ROW_NUMBER ():

- This function will generate row numbers for each row wise/ for each group of rows wise.

RANK ():

- This function will assign rank numbers to each row wise/ to each group of rows wise.
- It will skip the next rank number in the order.

DENSE_RANK ()

- This function will assign rank numbers to each row wise/ to each group of rows wise.
- It will not skip the next rank number in the order.

Ex:

ENAME	SALARY	ROW_NUMBER ()	RANK ()	DENSE_RANK ()
A	85000	1	1	1
B	72000	2	2	2
C	72000	3	2	2
D	68000	4	4	3
E	55000	5	5	4
F	43000	6	6	5

- 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)
ROWNUMBERS FROM EMP;

ENAME             SAL ROWNUMBERS
---------- ---------- ----------
KING              5000          1
FORD              3000          2
SCOTT             3000          3
JONES             2975          4
BLAKE             2850          5
CLARK             2450          6
ALLEN             1600          7
TURNER            1500          8
MILLER            1300          9
WARD              1250         10
MARTIN            1250         11
ADAMS             1100         12
JAMES              950         13
SMITH              800         14

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

ENAME             SAL      RANKS
---------- ---------- ----------
KING              5000          1
FORD              3000          2
SCOTT             3000          2
JONES             2975          4
BLAKE             2850          5
CLARK             2450          6
ALLEN             1600          7
TURNER            1500          8
MILLER            1300          9
WARD              1250         10
MARTIN            1250         10
ADAMS             1100         12
JAMES              950         13
SMITH              800         14

SQL> SELECT ENAME, SAL, DENSE_RANK() OVER(ORDER BY SAL DESC)
DENSERANKS FROM EMP;
```

```
ENAME              SAL DENSERANKS
---------- ---------- ----------
KING             5000          1
FORD             3000          2
SCOTT            3000          2
JONES            2975          3
BLAKE            2850          4
CLARK            2450          5
ALLEN            1600          6
TURNER           1500          7
MILLER           1300          8
WARD             1250          9
MARTIN           1250          9
ADAMS            1100         10
JAMES             950         11
SMITH             800         12
```

With PARTITION BY clause:

Ex:

```
SQL> SELECT ENAME, SAL, DEPTNO, ROW_NUMBER() OVER(PARTITION BY
DEPTNO ORDER BY SAL DESC) ROWNUMBERS FROM EMP;

ENAME              SAL     DEPTNO ROWNUMBERS
---------- ---------- ---------- ----------
KING             5000         10          1
CLARK            2450         10          2
MILLER           1300         10          3
SCOTT            3000         20          1
FORD             3000         20          2
JONES            2975         20          3
ADAMS            1100         20          4
SMITH             800         20          5
BLAKE            2850         30          1
ALLEN            1600         30          2
TURNER           1500         30          3
MARTIN           1250         30          4
WARD             1250         30          5
JAMES             950         30          6

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

ENAME              SAL     DEPTNO      RANKS
---------- ---------- ---------- ----------
KING             5000         10          1
CLARK            2450         10          2
MILLER           1300         10          3
SCOTT            3000         20          1
```

```
FORD              3000         20          1
JONES             2975         20          3
ADAMS             1100         20          4
SMITH              800         20          5
BLAKE             2850         30          1
ALLEN             1600         30          2
TURNER            1500         30          3
MARTIN            1250         30          4
WARD              1250         30          4
JAMES              950         30          6

SQL> SELECT ENAME, SAL, DEPTNO, DENSE_RANK() OVER(PARTITION BY
DEPTNO ORDER BY SAL DESC) DENSERANKS FROM EMP;

ENAME             SAL     DEPTNO DENSERANKS
---------- ---------- ---------- ----------
KING              5000         10          1
CLARK             2450         10          2
MILLER            1300         10          3
SCOTT             3000         20          1
FORD              3000         20          1
JONES             2975         20          2
ADAMS             1100         20          3
SMITH              800         20          4
BLAKE             2850         30          1
ALLEN             1600         30          2
TURNER            1500         30          3
MARTIN            1250         30          4
WARD              1250         30          4
JAMES              950         30          5
```

Ex: WAQ to display 3rd highest salary employee details from emp table in each DEPTNO wise by using DENSE_RANK () along with inline view.

```
SQL> SELECT * FROM (SELECT ENAME, SAL, DEPTNO, DENSE_RANK() OVER
(PARTITION BY DEPTNO ORDER BY SAL DESC) R FROM EMP) WHE
RE R=3;

ENAME             SAL     DEPTNO          R
---------- ---------- ---------- ----------
MILLER            1300         10          3
ADAMS             1100         20          3
TURNER            1500         30          3
```

Ex: WAQ to display the 4th senior most employee from each JOB wise.

```
SQL> SELECT * FROM (SELECT ENAME, JOB, HIREDATE, DENSE_RANK() OVER
(PARTITION BY JOB ORDER BY HIREDATE) R FROM EMP) WHER
```

```
E R=4;

ENAME      JOB       HIREDATE           R
---------- --------- --------- ----------
ADAMS      CLERK     12-JAN-83          4
MARTIN     SALESMAN  28-SEP-81          4
```

Co-related Subquery

- In corelated 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.

  ### Syntax to find out "nth" high/ low salary:

  SQL> SELECT * FROM <table name> <table alias name1> WHERE N-1= (SELECT COUNT (DISTINCT <column name>) FROM <table name> <table alias name2> WHERE <table alias name2>.<column name> (< / >) <table alias name1>.<column name>);

First create the below table with following data.

```
SQL> SELECT * FROM EMPLOYEE;

ENAME             SAL
---------- ----------
SMITH           85000
WARD            37000
MILLER          55000
SCOTT           12000
JONES           85000
```

Ex: WAQ to find out employee who are getting first highest salary from employee table.

```
SQL> SELECT * FROM EMPLOYEE E1 WHERE 0=(SELECT COUNT(DISTINCT SAL)
FROM EMPLOYEE E2 WHERE E2.SAL>E1.SAL);

ENAME             SAL
---------- ----------
SMITH           85000
JONES           85000
```

Ex: WAQ to find out employee who are getting fourth highest salary from employee table.

```
SQL> SELECT * FROM EMPLOYEE E1 WHERE 3=(SELECT COUNT(DISTINCT SAL)
FROM EMPLOYEE E2 WHERE E2.SAL>E1.SAL);

ENAME             SAL
---------- ----------
SCOTT           12000
```

Syntax to display "top n" high/ low salaries:

SQL> SELECT * FROM <table name> <table alias name1> WHERE N > (SELECT COUNT (DISTINCT <column name>) FROM <table name> <table alias name2> where <table alias name2>.<column name> (< / >) <table alias name1>.<column name>);

Ex: WAQ to display top 3 highest salaries employee details from employee table.

```
SQL> SELECT * FROM EMPLOYEE E1 WHERE 3 > (SELECT COUNT(DISTINCT SAL)
FROM EMPLOYEE E2 WHERE E2.SAL>E1.SAL);

ENAME             SAL
---------- ----------
SMITH           85000
WARD            37000
MILLER          55000
JONES           85000
```

Note:

- ✓ To find out "nth" high / low salary --> N-1
- ✓ To display "top n" high / low salaries --> N >

Exists operator:

- It is a special operator which is used in co-related subquery only.
- This operator is used to check whether row/ rows exist in the table or not.
- It returns either true (or) false. If subquery returns at least one row then returns true or else if subquery not returns any row, then return false.
- Syntax: WHERE EXISTS (<select statement>)

Ex: WAQ to display department details in which department employee are working.

```
SQL> SELECT * FROM DEPT D WHERE EXISTS (SELECT DEPTNO FROM EMP E
WHERE E.DEPTNO=D.DEPTNO);
```

```
    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
```

Ex: WAQ to display department details in which department employee are not working.

```
SQL> SELECT * FROM DEPT D WHERE NOT EXISTS (SELECT DEPTNO FROM EMP E
WHERE E.DEPTNO=D.DEPTNO);

    DEPTNO DNAME          LOC
---------- -------------- -------------
        40 OPERATIONS     BOSTON
```

Scalar Subquery

- Subqueries in select clause is called as scalar subquery. Every subquery output will act as a column.

  Syntax:
  SQL> SELECT (subquery1), (subquery2), ... FROM <table name> [WHERE <condition>];

```
SQL> SELECT (SELECT COUNT(*) FROM EMP) AS EMPTOTAL, (SELECT COUNT(*)
FROM DEPT) AS DEPTTOTAL FROM DUAL;

  EMPTOTAL  DEPTTOTAL
---------- ----------
        14          4
```

```
SQL> SELECT (SELECT SUM(SAL) FROM EMP WHERE DEPTNO=10)
TOTAL_SAL_DEPT_10,
  2  (SELECT SUM(SAL) FROM EMP WHERE DEPTNO=20) TOTAL_SAL_DEPT_20,
  3  (SELECT SUM(SAL) FROM EMP WHERE DEPTNO=30) TOTAL_SAL_DEPT_30
  4  FROM DUAL;

TOTAL_SAL_DEPT_10     TOTAL_SAL_DEPT_20     TOTAL_SAL_DEPT_30
-----------------     -----------------     -----------------
             8750                 10875                  9400
```

Synonyms

- It a database object to create permanent alias names for DB objects like

table, view, procedure, etc.

- Synonym is nothing but alternative name for DB objects like table, view, procedure, etc.
- Synonyms are created to reduce lengthy table name.
- There are two types of synonyms,
  - Private synonym (default)
  - Public synonym

Ex:

Table name: COLLEGE_ENROLLMENT_DETAILS

Operations:

SELECT * FROM **COLLEGE_ENROLLMENT_DETAILS** WHERE ….;
INSERT INTO **COLLEGE_ENROLLMENT_DETAILS** VALUES (….);
UPDATE **COLLEGE_ENROLLMENT_DETAILS** SET SAL=3000 ….;
DELETE FROM **COLLEGE_ENROLLMENT_DETAILS** WHERE …;

After create a synonym, S1 for COLLEGE_ENROLLMENT_DETAILS

Operations:

SELECT * FROM **S1** WHERE ….;
INSERT INTO **S1** VALUES (….);
UPDATE **S1** SET SAL=3000 ….;
DELETE FROM **S1** WHERE …;

Ex: In SYSTEM user We have a table DEPT we want to give SELECT GRANT to any other user, so instead of giving directly grant we can create a synonym and we can give grant to that synonym then anther user will use that synonym by this our owner's name and table name hided.
Synonym create S2 for SYSTEM.DEPT and S2 GRANT to user U1.

SQL> CONN U1/U1
SQL> SELECT * FROM S2;

Private synonyms:

- These synonyms are created by users who are having permission.

Syntax:

SQL> CREATE SYNONYM <synonym name> FOR <DB object name>;

With out synonym:

```
SQL> CONN
Enter user-name: system/tiger
```

```
Connected.

SQL> CREATE USER U1 IDENTIFIED BY U1;

User created.

SQL> GRANT CONNECT, CREATE TABLE, UNLIMITED TABLESPACE TO U1;

Grant succeeded.

SQL> CONN
Enter user-name: U1/U1
Connected.

SQL> CREATE TABLE COLLEGE_ENROLLMENT_DETAILS(SID INT, SNAME
VARCHAR2(10));

Table created.

SQL> INSERT INTO COLLEGE_ENROLLMENT_DETAILS VALUES (1, 'SMITH');

1 row created.

SQL> INSERT INTO COLLEGE_ENROLLMENT_DETAILS VALUES (2, 'ALLEN');

1 row created.

SQL> SELECT * FROM COLLEGE_ENROLLMENT_DETAILS;
       SID SNAME
---------- ----------
         1 SMITH
         2 ALLEN

SQL> UPDATE COLLEGE_ENROLLMENT_DETAILS SET SID=101 WHERE SID=1;

1 row updated.

SQL> DELETE FROM COLLEGE_ENROLLMENT_DETAILS WHERE SID=2;

1 row deleted.
```

With synonym:

```
SQL> CREATE SYNONYM S1 FOR COLLEGE_ENROLLMENT_DETAILS;
CREATE SYNONYM S1 FOR COLLEGE_ENROLLMENT_DETAILS
*
ERROR at line 1:
ORA-01031: insufficient privileges

SQL> CONN
```

```
Enter user-name: system/tiger
Connected.

SQL> GRANT CREATE SYNONYM TO U1;

Grant succeeded.

SQL> CONN
Enter user-name: U1/U1
Connected.

SQL> CREATE SYNONYM S1 FOR COLLEGE_ENROLLMENT_DETAILS;

Synonym created.

SQL> SELECT * FROM S1;

       SID SNAME
---------- ----------
       101 SMITH

SQL> INSERT INTO S1 VALUES (102, 'ALLEN');

1 row 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.

Public synonyms:

- These synonyms are created by DBA.
- We should have "create public synonym" privilege and it can access by all users
- To use for hiding the information about owner name and object name (table).

  Syntax:
  SQL> CREATE PUBLIC SYNONYM <synonym name> FOR [user name]. <DB object name>;

```
SQL> CONN
Enter user-name: system/tiger
Connected.

SQL> CREATE USER U2 IDENTIFIED BY U2;

User created.
```

```
SQL> GRANT CONNECT, CREATE TABLE, UNLIMITED TABLESPACE TO U2;

Grant succeeded.

SQL> SELECT * FROM DEPT;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON

SQL> CREATE PUBLIC SYNONYM PS1 FOR SYSTEM.DEPT;

Synonym created.

SQL> GRANT SELECT ON PS1 TO U2;

Grant succeeded.

SQL> CONN
Enter user-name: U2/U2
Connected.

SQL> SELECT * FROM PS1;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

Note: Once we created public synonym then any user can access that public synonym without "username".

- To view all synonyms information in Oracle DB then we use "USER_SYNONYMS" data dictionary.

```
SQL> DESC USER_SYNONYMS;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------
 SYNONYM_NAME                              NOT NULL VARCHAR2(128)
 TABLE_OWNER                                        VARCHAR2(128)
 TABLE_NAME                                NOT NULL VARCHAR2(128)
```

```
 DB_LINK                                                   VARCHAR2(128)
 ORIGIN_CON_ID                                             NUMBER

SQL> SELECT SYNONYM_NAME, TABLE_NAME FROM USER_SYNONYMS;
```

- To view all private and public synonyms of a particular user in oracle database then we "ALL_SYNONYMS" data dictionary.

```
SQL> DESC ALL_SYNONYMS;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------
 OWNER                                               VARCHAR2(128)
 SYNONYM_NAME                                        VARCHAR2(128)
 TABLE_OWNER                                         VARCHAR2(128)
 TABLE_NAME                                          VARCHAR2(128)
 DB_LINK                                             VARCHAR2(128)
 ORIGIN_CON_ID                                       NUMBER

SQL> SELECT SYNONYM_NAME, TABLE_NAME FROM ALL_SYNONYMS WHERE
TABLE_NAME='DEPT';

SYNONYM_NAME                   TABLE_NAME
-------------------            ------------------------
PS1                            DEPT
```

Syntax to drop private synonyms:

In this case we dropping private synonyms only and dropping by user.
SQL> DROP SYNONYM <synonym name>;

```
SQL> DROP SYNONYM PVT_SYN;
```

Syntax to drop public synonyms:

In this case we dropping public synonyms only and dropping by DBA.
SQL> DROP PUBLIC SYNONYM <synonym name>;

```
SQL> DROP PUBLIC SYNONYM PUB_SYN;
```

Views

- View is DB object is called subset of a table.
- View is also called as logical/ virtual table of a base table (main table) because it doesn't store data and it doesn't occupy any memory.
- View is creating by using "select query" for getting/ retrieving the required data/ information from a base table and display to user.
- That means view will act as an “interface” between user & base table.

User <----------> <view> <-----------> base table

Purpose:

- Security:
  - Column level security: Hiding columns data from users.
  - Row level security: Hiding rows data from users.
- To save query statement.
- To check integrity rules.
- To convert complex query into simple query.

- A user can create the following two types of views on base tables those are,
  - Simple views
  - Complex views

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:

SQL> CREATE VIEW <view name> AS SELECT * FROM <table name> [WHERE <condition>];

```
SQL> CONN
Enter user-name: nirmala/nirmala
Connected.

SQL> CREATE VIEW SV1 AS SELECT * FROM DEPT;
CREATE VIEW SV1 AS SELECT * FROM DEPT
            *
ERROR at line 1:
ORA-01031: insufficient privileges

SQL> CONN
Enter user-name: system/tiger
Connected.

SQL> GRANT CREATE VIEW TO NIRMALA;
```

```
SQL> CONN
Enter user-name: nirmala/nirmala
Connected.

SQL> CREATE VIEW SV1 AS SELECT * FROM DEPT;

View created.

SQL> SELECT * FROM SV1;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON

SQL> INSERT INTO SV1 VALUES(50, 'DBA', 'HYD');

1 row created.

SQL> UPDATE SV1 SET LOC='PUNE' WHERE DEPTNO=50;

1 row updated.

SQL> DELETE FROM DEPT WHERE DEPTNO=50;

1 row deleted.
```

Note: Whenever we perform DML operations on view internally the view will perform those operations on base table.

Ex: Create view to access EMPNO, ENAME, SAL from emp table (column level security).

```
SQL> CREATE VIEW SV2 AS SELECT EMPNO, ENAME, SAL FROM EMP;

View created.

SQL> SELECT * FROM SV2;

     EMPNO ENAME             SAL
---------- ---------- ----------
      7369 SMITH             800
      7499 ALLEN            1600
      7521 WARD             1250

SQL> INSERT INTO SV2 VALUES (1122, 'YUVIN', 5500, 10);
```

```
INSERT INTO SV2 VALUES (1122, 'YUVIN', 5500, 10)
            *
ERROR at line 1:
ORA-00913: too many values


SQL> INSERT INTO SV2 VALUES (1122, 'YUVIN', 5500);

1 row created.

SQL> ALTER TABLE EMP MODIFY JOB CONSTRAINT NN_JOB NOT NULL;

Table altered.

SQL> INSERT INTO SV2 VALUES (1144, 'JUVIN', 5500);
INSERT INTO SV2 VALUES (1144, 'JUVIN', 5500)
*
ERROR at line 1:
ORA-01400: cannot insert NULL into ("NIRMALA"."EMP"."JOB")

SQL> ALTER TABLE EMP DROP CONSTRAINT NN_JOB;

Table altered.

SQL> INSERT INTO SV2 VALUES (1144, 'JUVIN', 5500);

1 row created.
```

Note: If hiding have any constraint the while performing DML operation we have to consider those things and we have to handle according to the constraint rules.

Ex: Create a view access employee detail who are working under DEPTNO is 20 (Row level security).

```
SQL> CREATE VIEW SV3 AS SELECT * FROM EMP WHERE DEPTNO=20;

View created.

SQL> SELECT * FROM SV3;

EMPNO ENAME      JOB         MGR HIREDATE       SAL      COMM     DEPTNO
----- --------   ------      --- ----------     ------   -------  ------
7369 SMITH       CLERK       7902 17-DEC-80     800                   20
7566 JONES       MANAGER     7839 02-APR-81     2975                  20
7788 SCOTT       ANALYST     7566 09-DEC-82     3000                  20
7876 ADAMS       CLERK       7788 12-JAN-83     1100                  20
7902 FORD        ANALYST     7566 03-DEC-81     3000                  20
```

```
SQL> INSERT INTO SV3 (EMPNO, ENAME, DEPTNO) VALUES (1155, 'YUVIN',
10);

1 row created.

SQL> SELECT * FROM SV3;

EMPNO ENAME      JOB         MGR HIREDATE      SAL      COMM     DEPTNO
----- --------   ------      --- ----------    ------   -------  ------
7369 SMITH       CLERK       7902 17-DEC-80    800                   20
7566 JONES       MANAGER     7839 02-APR-81    2975                  20
7788 SCOTT       ANALYST     7566 09-DEC-82    3000                  20
7876 ADAMS       CLERK       7788 12-JAN-83    1100                  20
7902 FORD        ANALYST     7566 03-DEC-81    3000                  20

SQL> INSERT INTO SV3 (EMPNO, ENAME, DEPTNO) VALUES (1156, 'NARESH',
20);

1 row created.

SQL> SELECT * FROM SV3;

EMPNO ENAME      JOB         MGR HIREDATE      SAL      COMM     DEPTNO
----- --------   ------      --- ----------    ------   -------  ------
1156 NARESH                                                        20
7369 SMITH       CLERK       7902 17-DEC-80    800                   20
7566 JONES       MANAGER     7839 02-APR-81    2975                  20
7788 SCOTT       ANALYST     7566 09-DEC-82    3000                  20
7876 ADAMS       CLERK       7788 12-JAN-83    1100                  20
7902 FORD        ANALYST     7566 03-DEC-81    3000                  20
```

View Option:

With check option:

- It is a constraint which is used to restrict rows on base table through a view while performing DML operations.

```
SQL> CREATE VIEW SV4 AS SELECT EMPNO, ENAME, SAL FROM EMP WHERE
SAL=3000 WITH CHECK OPTION;

View created.

SQL> SELECT * FROM SV4;

     EMPNO ENAME             SAL
---------- ---------- ----------
      7788 SCOTT            3000
      7902 FORD             3000
```

```
SQL> INSERT INTO SV4 VALUES (1125, 'SURESH', 2500);
INSERT INTO SV4 VALUES (1125, 'SURESH', 2500)
            *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation

SQL> INSERT INTO SV4 VALUES (1125, 'SURESH', 3500);
INSERT INTO SV4 VALUES (1125, 'SURESH', 3500)
            *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation

SQL> INSERT INTO SV4 VALUES (1125, 'SURESH', 3000);

1 row created.

SQL> SELECT * FROM SV4;

     EMPNO ENAME              SAL
---------- ---------- ----------
      1125 SURESH           3000
      7788 SCOTT            3000
      7902 FORD             3000
```

With read only:

- If we created a view using "with read only" clause then we restrict DML operations on base table through a view object.
- We can read data from base table, supporting "SELECT and DESC" commands only.

```
SQL> CREATE VIEW SV5 AS SELECT * FROM DEPT WITH READ ONLY;

View created.

SQL> SELECT * FROM SV5;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON

SQL> INSERT INTO SV5 VALUES (50, 'DBA', 'HYD');
INSERT INTO SV5 VALUES (50, 'DBA', 'HYD')
*
```

```
ERROR at line 1:
ORA-42399: cannot perform a DML operation on a read-only view
```

Complex views:

- A view is called as complex view,
  - When we create based on multiple base tables.
  - When we create a view with aggregative functions, group by, having clauses, set operators, sub-query, distinct key word.
- By default, complex view is not supporting DML operations.

First create the below tables with data

```
SQL> SELECT * FROM EMP_HYD;

       EID ENAME             SAL
---------- ---------- ----------
      1021 SMITH           85000
      1022 ALLEN           65000
      1023 WARD            48000

SQL> SELECT * FROM EMP_CHENNAI;

       EID ENAME             SAL
---------- ---------- ----------
      1021 SMITH           85000
      1024 JONES           15000

SQL> CREATE VIEW CV1 AS
  2  SELECT * FROM EMP_HYD
  3  UNION
  4  SELECT * FROM EMP_CHENNAI;

View created.

SQL> SELECT * FROM CV1;

       EID ENAME             SAL
---------- ---------- ----------
      1021 SMITH           85000
      1022 ALLEN           65000
      1023 WARD            48000
      1024 JONES           15000

SQL> INSERT INTO CV1 VALUES (1025, 'SCOTT', 63000);
INSERT INTO CV1 VALUES (1025, 'SCOTT', 63000)
            *
```

```
ERROR at line 1:
ORA-01732: data manipulation operation not legal on this view
```

Note:

- ✓ The above complex view CV1 is not allow DML operations.
- ✓ When we create a view with function then we must create alias name for aggregative functions otherwise oracle returns an error.

```
SQL> CREATE VIEW CV2 AS
  2  SELECT DEPTNO, SUM(SAL) FROM EMP GROUP BY DEPTNO;
SELECT DEPTNO, SUM(SAL) FROM EMP GROUP BY DEPTNO
               *
ERROR at line 2:
ORA-00998: must name this expression with a column alias

SQL> CREATE VIEW CV2 AS
  2  SELECT DEPTNO, SUM(SAL) SUMSAL FROM EMP GROUP BY DEPTNO;

View created.

SQL> SELECT * FROM CV2;

    DEPTNO     SUMSAL
---------- ----------
        30       9400
        10       8750
        20      10875
```

Note: When we create a view on base tables then we should not allow duplicate column names. To avoid this problem, we have to use "using" clause.

```
SQL> SELECT * FROM STUDENT;

       SID SNAME             CID
---------- ---------- ----------
         1 SMITH              10
         2 ALLEN              10
         3 WARD               20
         4 JONES              30

SQL> SELECT * FROM COURSE;

       CID CNAME           CFEE
---------- ---------- ----------
        10 ORACLE           2500
        20 JAVA             5000
        40 PYTHON          45000
```

```
SQL> CREATE VIEW CV3 AS SELECT * FROM STUDENT S INNER JOIN COURSE ON
S.CID=C.CID;
CREATE VIEW CV3 AS SELECT * FROM STUDENT S INNER JOIN COURSE ON
S.CID=C.CID

*
ERROR at line 1:
ORA-00904: "C"."CID": invalid identifier


SQL> CREATE VIEW CV3 AS SELECT * FROM STUDENT S INNER JOIN COURSE C
USING(CID);

View created.
```

Note: Generally complex view is 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.

```
SQL> CREATE VIEW CV4 AS SELECT EMPNO, ENAME, SAL, D. DEPTNO, DNAME,
LOC FROM EMP E INNER JOIN DEPT D ON E. DEPTNO=D.DEPT
NO;

View created.

SQL> UPDATE CV4 SET SAL=500 WHERE EMPNO=7788;

1 row updated.

SQL> DELETE FROM CV4 WHERE EMPNO=7782;

1 row deleted.

SQL> INSERT INTO CV4 VALUES (1122,'SAI',6000,10,'SAP','HYD');
INSERT INTO CV4 VALUES (1122,'SAI',6000,10,'SAP','HYD')
*
ERROR at line 1:
ORA-01776: cannot modify more than one base table through a join
view
```

Force views:

- Generally, views are created based on tables, but force views are created without tables.

  Syntax:
  SQL> CREATE FORCE VIEW <view name> AS <select query>;

```
SQL> CREATE FORCE VIEW FV1 AS SELECT * FROM TEST;

Warning: View created with compilation errors.

SQL> SELECT * FROM FV1;
SELECT * FROM FV1
              *
ERROR at line 1:
ORA-04063: view "NIRMALA.FV1" has errors
```

- To activate a force view then we should create a table with the name as "test".

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

Table created.

SQL> SELECT * FROM FV1;

no rows selected
```

Note: To view all views details in oracle DB then we use the following data dictionary is "USER_VIEWS".

```
SQL> DESC USER_VIEWS;

SQL> SELECT VIEW_NAME FROM USER_VIEWS;

VIEW_NAME
------------------------------------------------------------------------
FV1
```

Syntax to drop a view:

SQL> DROP VIEW <view name>;

```
SQL> DROP VIEW SV1;

View dropped.

SQL> DROP VIEW CV1;

View dropped.

SQL> DROP VIEW FV1;

View dropped.
```

Advantages of views:

- It is providing security. It means that to each user can be given permission to access specific columns & specific rows from a table.
- 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.
- Query simplify it means that to reduce complex query.

Q. What is the differences between synonym and view?

Ans.

Synonym	View
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 are not store any data whereas materialized views are storing data/ information.
- These views are used in data warehousing and handling by DBA.
- Materialized views are also created from base tables.
- These views are not allowed DML operations.
- These views are independent object whereas view is a dependent object.
- Once we drop a base table view is not working but materialized view is working.

Syntax:

SQL> CREATE MATERIALIZED VIEW <view name> AS <select query>;

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

Table created.

SQL> CREATE VIEW NV AS SELECT * FROM TEST;

View created.

SQL> CREATE MATERIALIZED VIEW MV AS SELECT * FROM TEST;
CREATE MATERIALIZED VIEW MV AS SELECT * FROM TEST
                                             *
```

```
ERROR at line 1:
ORA-01031: insufficient privileges

SQL> CONN
Enter user-name: system/tiger
Connected.

SQL> GRANT CREATE MATERIALIZED VIEW TO NIRMALA;

Grant succeeded.

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

SQL> CREATE MATERIALIZED VIEW MV AS SELECT * FROM TEST;

Materialized view created.

SQL> SELECT * FROM NV;

no rows selected

SQL> SELECT * FROM MV;

no rows selected

SQL> INSERT INTO TEST VALUES (101, 'SMITH');

1 row created.

SQL> SELECT * FROM TEST;

       SNO NAME
---------- ----------
       101 SMITH

SQL> SELECT * FROM NV;

       SNO NAME
---------- ----------
       101 SMITH

SQL> SELECT * FROM MV;

no rows selected
```

- Here, base table (TEST) and view (NV) table data is updated but materialized view (MV) table data is not updated. If we want to update

data in materialized view then we refresh materialized view by using Refreshing methods.

Refreshing methods:

- In oracle we are refreshing materialized view in two ways those are,
  - On demand: It is a default refreshing method. In this method we are refreshing materialized view by using "DBMS_MVIEW" procedure.

    Syntax:
    SQL> Execute DBMS_MVIEW.REFRESH ('mview name');

```
SQL> EXECUTE DBMS_MVIEW.REFRESH('MV');

PL/SQL procedure successfully completed.

SQL> SELECT * FROM MV;

       SNO NAME
---------- ----------
       101 SMITH

SQL> INSERT INTO TEST VALUES (102, 'ALLEN');

1 row created.

SQL> SELECT * FROM TEST;

       SNO NAME
---------- ----------
       101 SMITH
       102 ALLEN

SQL> SELECT * FROM NV;

       SNO NAME
---------- ----------
       101 SMITH
       102 ALLEN

SQL> SELECT * FROM MV;

       SNO NAME
---------- ----------
       101 SMITH
```

Note:

- ✓ Every time we add any new data in the base table, we have to execute the refresh command to update in Materialized view.
- ✓ After dropping the base table, we can't access the view but we can access the materialized view.

```
SQL> DROP TABLE TEST PURGE;

Table dropped.

SQL> SELECT * FROM NV;
SELECT * FROM NV
              *
ERROR at line 1:
ORA-04063: view "NIRMALA.NV" has errors

SQL> SELECT * FROM MV;

       SNO NAME
---------- ----------
       101 SMITH
```

- On commit: We can refresh a materialized view without using "DBMS_MVIEW" but using "REFRESH ON COMMIT" method.

  Syntax:
  SQL> CREATE MATERIALIZED VIEW <view name> REFRESH ON COMMIT <select query>;

```
SQL> CREATE TABLE TEST(SNO INT, SAL NUMBER(10));

Table created.

SQL> CREATE VIEW NV1 AS SELECT * FROM TEST;

View created.

SQL> CREATE MATERIALIZED VIEW MV2 REFRESH ON COMMIT AS SELECT * FROM
TEST;
CREATE MATERIALIZED VIEW MV2 REFRESH ON COMMIT AS SELECT * FROM TEST
                                                             *
ERROR at line 1:
ORA-12054: cannot set the ON COMMIT refresh attribute for the
materialized view
```

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.

```
SQL> ALTER TABLE TEST ADD CONSTRAINT PK_SNO PRIMARY KEY(SNO);

Table altered.

SQL> CREATE MATERIALIZED VIEW MV2 REFRESH ON COMMIT AS SELECT * FROM
TEST;

Materialized view created.

SQL> INSERT INTO TEST VALUES (101, 40000);

1 row created.

SQL> SELECT * FROM TEST;

       SNO        SAL
---------- ----------
       101      40000

SQL> SELECT * FROM NV1;

       SNO        SAL
---------- ----------
       101      40000

SQL> SELECT * FROM MV2;

no rows selected
```

Note: We can see the inserted data in materialized view until we not commit, So first we have to commit.

```
SQL> COMMIT;

Commit complete.

SQL> SELECT * FROM MV2;

       SNO        SAL
---------- ----------
       101      40000
```

Note: If we want to view materialized views then we are using the following data dictionary is “USER_MVIEWS”.

```
SQL> DESC USER_MVIEWS;

SQL> SELECT MVIEW_NAME FROM USER_MVIEWS;

MVIEW_NAME
------------------------------------------------
MV
```

Syntax to drop materialized view:

SQL> DROP MATERIALIZED VIEW <materialized view name>;

```
SQL> DROP MATERIALIZED VIEW MV;

Materialized view dropped.

SQL> DROP MATERIALIZED VIEW MV2;

Materialized view dropped.
```

Q. What is the differences between view and materialized view:

Ans.

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;

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.
- There are following types of partitions:
  - Range partition
  - List partition
  - Hash partition
- If we want to access a particular partition then we follow the following,

Syntax to call a particular partition:

SQL> SELECT * FROM <table name> PARTITION (<partition name>);

Range partition:

- In this method we are creating partitions table based on a particular range value.

Syntax:

SQL> CREATE TABLE <table name> (<column name1> <datatypes>[size], ...) PARTITION BY RANGE (<key column name>) (PARTITION <partition name1> VALUES LESS THAN (value), PARTITION <partition name2> VALUES LESS THAN (value), ...);

```
SQL> CREATE TABLE TEST(EID INT, ENAME VARCHAR2(10), SAL NUMBER(10))
  2  PARTITION BY RANGE (SAL)
  3  (PARTITION P1 VALUES LESS THAN (500),
  4  PARTITION P2 VALUES LESS THAN (1000),
  5  PARTITION P3 VALUES LESS THAN (2000));

Table created.

SQL> INSERT INTO TEST VALUES(1, 'SMITH', 1200);

1 row created.

SQL> INSERT INTO TEST VALUES(2, 'ALLEN', 450);

1 row created.

SQL> INSERT INTO TEST VALUES(3, 'WARD', 850);

1 row created.

SQL> INSERT INTO TEST VALUES(4, 'SCOTT', 1800);

1 row created.

SQL> SELECT * FROM TEST;

       EID ENAME             SAL
---------- ---------- ----------
         2 ALLEN              450
         3 WARD               850
         1 SMITH             1200
         4 SCOTT             1800

SQL> SELECT * FROM TEST PARTITION(P1);

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

```
         2 ALLEN             450

SQL> SELECT * FROM TEST PARTITION(P2);

       EID ENAME             SAL
---------- ---------- ----------
         3 WARD              850

SQL> SELECT * FROM TEST PARTITION(P3);

       EID ENAME             SAL
---------- ---------- ----------
         1 SMITH            1200
         4 SCOTT            1800

SQL> INSERT INTO TEST VALUES(5, 'MAX', 2500);
INSERT INTO TEST VALUES(5, 'MAX', 2500)
            *
ERROR at line 1:
ORA-14400: inserted partition key does not map to any partition
```

List partition:

- In this method we are creating partitions based on list of values.

Syntax:

SQL> CREATE TABLE <table name> (<column name1> <datatypes>[size], ...) PARTITION BY LIST (<key column name>) (PARTITION <partition name1> VALUES (value1, value2, ……), PARTITION <partition name2> VALUES (value1, valu2, ……), ……, PARTITION OTHERS VALUE (DEFAULT));

```
SQL> CREATE TABLE TEST2(CID INT, CNAME VARCHAR2(10))
  2  PARTITION BY LIST(CNAME)
  3  (PARTITION P1 VALUES('ORACLE', 'MYSQL'),
  4  PARTITION P2 VALUES('JAVA', '.NET', 'PYTHON'),
  5  PARTITION OTHERS VALUES(DEFAULT));

Table created.

SQL> INSERT INTO TEST2 VALUES(1, 'ORACLE');

1 row created.

SQL> INSERT INTO TEST2 VALUES(2, 'JAVA');

1 row created.

SQL> INSERT INTO TEST2 VALUES(3, 'C');
```

```
1 row created.

SQL> INSERT INTO TEST2 VALUES(4, 'C++');

1 row created.

SQL> SELECT * FROM TEST2;

       CID CNAME
---------- ----------
         1 ORACLE
         2 JAVA
         3 C
         4 C++

SQL> SELECT * FROM TEST2 PARTITION(OTHERS);

       CID CNAME
---------- ----------
         3 C
         4 C++

SQL> SELECT * FROM TEST2 PARTITION(P1);

       CID CNAME
---------- ----------
         1 ORACLE

SQL> SELECT * FROM TEST2 PARTITION(P2);

       CID CNAME
---------- ----------
         2 JAVA
```

Hash partition:

- In this method partitions are created by the system as per user requested number of partitions.

  Syntax:

  SQL> CREATE TABLE <table name> (<column name1> <datatype>[size], ...) PARTITION BY HASH (<key column name>) PARTITIONS <number>;

```
SQL> CREATE TABLE TEST3(ENAME VARCHAR2(10), SAL NUMBER(10))
  2  PARTITION BY HASH(SAL) PARTITIONS 5;
```

```
Table created.
```

Note: If we want to view all partitions information in oracle database then we use "USER_TAB_PARTITIONS" data dictionary.

```
SQL> DESC USER_TAB_PARTITIONS;

SQL> SELECT PARTITION_NAME FROM USER_TAB_PARTITIONS WHERE
TABLE_NAME='TEST3';

PARTITION_NAME
------------------------------------------------------
SYS_P367
SYS_P368
SYS_P369
SYS_P370
SYS_P371

SQL> SELECT PARTITION_NAME FROM USER_TAB_PARTITIONS WHERE
TABLE_NAME='TEST2';

PARTITION_NAME
------------------------------------------------------------
OTHERS
P1
P2

SQL> SELECT PARTITION_NAME FROM USER_TAB_PARTITIONS WHERE
TABLE_NAME='TEST';

PARTITION_NAME
------------------------------------------------------------------
P1
P2
P3
```

Adding a new partition:

Syntax for Rang partition:

SQL> ALTER TABLE <table name> ADD PARTITION <partition name> VALUES LESS THAN (value);

```
SQL> ALTER TABLE TEST ADD PARTITION P4 VALUES LESS THAN (3000);

Table altered.

SQL> SELECT PARTITION_NAME FROM USER_TAB_PARTITIONS WHERE
```

```
TABLE_NAME='TEST';

PARTITION_NAME
------------------------------------------------------------------------
P1
P2
P3
P4
```

Note: We can't add new partition for List partition because of default value and in hash partition also because of system created.

```
SQL> ALTER TABLE TEST2 ADD PARTITION P3 VALUES ('HTML', 'CSS');
ALTER TABLE TEST2 ADD PARTITION P3 VALUES ('HTML', 'CSS')
            *
ERROR at line 1:
ORA-14323: cannot add partition when DEFAULT partition exists
```

Dropping a partition:

Syntax:

SQL> ALTER TABLE <table name> DROP PARTITION <partition name>;

```
SQL> ALTER TABLE TEST DROP PARTITION P3;

Table altered.

SQL> SELECT * FROM TEST ;

       EID ENAME             SAL
---------- ---------- ----------
         2 ALLEN             450
         3 WARD              850
```

Note: We can use drop partition of Range and List partition by not in Hash partition. And when we drop the partition the partition data also got dropped.

```
SQL> ALTER TABLE TEST3 DROP PARTITION SYS_P367;
ALTER TABLE TEST3 DROP PARTITION SYS_P367
                                 *
ERROR at line 1:
ORA-14255: table is not partitioned by range, list, composite range,
or
composite list method
```

Note: If we want to know whether table is partitioned or not then we use "USER_TABLES" data dictionary.

```
SQL> DESC USER_TABLES;

SQL> SELECT PARTITIONED FROM USER_TABLES WHERE TABLE_NAME='EMP';

PAR
---
NO
```

Sequence

- Sequence is a DB object. Which is used to generate sequence numbers on a particular column automatically.

  Syntax:

  SQL> 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 represents the starting sequence number. Here "n" is represented with number.

- MINVALUE n:
  - It specifies the minimum value of the sequence. Here "n" is represented with number.

- INCREMENT BY n:
  - It specifies the incremental value in between sequence numbers. Here "n" is represented with number.

- MAXVALUE n:
  - It specifies the maximum value of the sequence. Here "n" is represented with number.

- NO CYCLE:
  - It is default parameter. If we created sequence with "NO CYCLE" then sequence starts from start with value and generate values up

to max value. After reaching max value then sequence is stop.
    - Means the sequence numbers are not repeated again and again.

- CYCLE:
    - If we created a sequence with "CYCLE" then sequence starts from start with value and generate values up to max value. After reaching max value then sequence will start with min value.

- 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 number of user requests are increasing then it will degrade the performance and burden on database.

- CACHE n:
    - When we created a sequence with "CACHE" parameter then system is allocating temporary 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 representing the size of cache file.
    - Minimum size of cache is 2KB and maximum size of cache is 20KB.

```
SQL> CONN
Enter user-name: system/tiger
Connected.

SQL> GRANT CREATE SEQUENCE TO NIRMALA;

Grant succeeded.

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

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

```
  5  MAXVALUE 3;

Sequence created.

SQL> CREATE TABLE TEST(SNO INT, NAME VARCHAR2(10));

Table created.
```

Note: To work with sequence object, we should use the following two pseudo columns are "NEXTVAL" and "CURRVAL".

NEXTVAL:

- It is used to generating next by next sequence numbers on a particular column.

  Syntax:
  SQL> SELECT <sequence name>. NEXTVAL FROM DUAL;

CURRVAL:

- It is used to show the current value of the sequence.

  Syntax:
  SQL> SELECT <sequence name>. CURRVAL FROM DUAL;

```
SQL> INSERT INTO TEST VALUES(SQ1.NEXTVAL, '&NAME');
Enter value for name: A
old   1: INSERT INTO TEST VALUES(SQ1.NEXTVAL, '&NAME')
new   1: INSERT INTO TEST VALUES(SQ1.NEXTVAL, 'A')

1 row created.

SQL> /
Enter value for name: B
old   1: INSERT INTO TEST VALUES(SQ1.NEXTVAL, '&NAME')
new   1: INSERT INTO TEST VALUES(SQ1.NEXTVAL, 'B')

1 row created.

SQL> /
Enter value for name: B
old   1: INSERT INTO TEST VALUES(SQ1.NEXTVAL, '&NAME')
new   1: INSERT INTO TEST VALUES(SQ1.NEXTVAL, 'B')

1 row created.
```

```
SQL> /
Enter value for name: D
old   1: INSERT INTO TEST VALUES(SQ1.NEXTVAL, '&NAME')
new   1: INSERT INTO TEST VALUES(SQ1.NEXTVAL, 'D')
INSERT INTO TEST VALUES(SQ1.NEXTVAL, 'D')
                        *
ERROR at line 1:
ORA-08004: sequence SQ1.NEXTVAL exceeds MAXVALUE and cannot be
instantiated
```

Altering a sequence:

Syntax:

SQL> ALTER SEQUENCE <sequence name> <parameter name> n;

```
SQL> ALTER SEQUENCE SQ1 MAXVALUE 5;

Sequence altered.

SQL> INSERT INTO TEST VALUES(SQ1.NEXTVAL, '&NAME');
Enter value for name: D
old   1: INSERT INTO TEST VALUES(SQ1.NEXTVAL, '&NAME')
new   1: INSERT INTO TEST VALUES(SQ1.NEXTVAL, 'D')

1 row created

SQL> /
Enter value for name: E
old   1: INSERT INTO TEST VALUES(SQ1.NEXTVAL, '&NAME')
new   1: INSERT INTO TEST VALUES(SQ1.NEXTVAL, 'E')

1 row created.

SQL> SELECT * FROM TEST;

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

Note: We can alter all parameters except "START WITH" parameter.

```
SQL> CREATE SEQUENCE SQ2
  2  START WITH 1
  3  MINVALUE 1
```

```
  4  INCREMENT BY 1
  5  MAXVALUE 3
  6  CYCLE
  7  CACHE 2;

Sequence created.

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

Table created.

SQL> INSERT INTO TEST1 VALUES(SQ2.NEXTVAL, '&NAME');
Enter value for name: A
old   1: INSERT INTO TEST1 VALUES(SQ2.NEXTVAL, '&NAME')
new   1: INSERT INTO TEST1 VALUES(SQ2.NEXTVAL, 'A')

1 row created.

SQL> /
Enter value for name: B
old   1: INSERT INTO TEST1 VALUES(SQ2.NEXTVAL, '&NAME')
new   1: INSERT INTO TEST1 VALUES(SQ2.NEXTVAL, 'B')

1 row created.

SQL> /
Enter value for name: B
old   1: INSERT INTO TEST1 VALUES(SQ2.NEXTVAL, '&NAME')
new   1: INSERT INTO TEST1 VALUES(SQ2.NEXTVAL, 'B')

1 row created.

SQL> /
Enter value for name: C
old   1: INSERT INTO TEST1 VALUES(SQ2.NEXTVAL, '&NAME')
new   1: INSERT INTO TEST1 VALUES(SQ2.NEXTVAL, 'C')

1 row created.

SQL> /
Enter value for name: D
old   1: INSERT INTO TEST1 VALUES(SQ2.NEXTVAL, '&NAME')
new   1: INSERT INTO TEST1 VALUES(SQ2.NEXTVAL, 'D')

1 row created.

SQL> SELECT * FROM TEST1;

       SNO NAME
```

```
---------- ----------
         1 A
         2 B
         3 B
         1 C
         2 D
```

Note: If we want to view all sequences in oracle database then we use "USER_SEQUENCES" data dictionary.

```
SQL> DESC USER_SEQUENCES;

SQL> SELECT SEQUENCE_NAME FROM USER_SEQUENCES;

SEQUENCE_NAME
--------------------------------------------------------
SQ1
SQ2
```

Syntax to drop a sequence:

SQL> DROP SEQUENCE <sequence name>;

```
SQL> DROP SEQUENCE SQ1;

Sequence dropped.

SQL> DROP SEQUENCE SQ2;

Sequence dropped.
```

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,
    - Row level locks
    - 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.

On row level locking on a single row:

User - 1 Terminal:

```
SQL> SHOW USER;
USER is "SYSTEM"

SQL> UPDATE NIRMALA.EMP SET SAL=1100 WHERE EMPNO=7369;

1 row updated.
```

User - 2 Terminal:

```
SQL> SHOW USER
USER is "NIRMALA"

SQL> UPDATE NIRMALA.EMP SET SAL=800 WHERE EMPNO=7369;
```

Note: we can't perform update operation because this row is locked by the user system, so until that release, we can't anything on that particular row so to release the lock we have to do either "COMMIT or ROLLBACK".

User - 1 Terminal:

```
SQL> COMMIT;

Commit complete.
```

When you execute the COMMIT command in user 1, you can see the second user update command will perform automatically and got success message.

User - 2 Terminal:

```
SQL> UPDATE NIRMALA.EMP SET SAL=1100 WHERE EMPNO=7369;

1 row updated
```

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 Terminal:

```
SQL> SHOW USER;
USER is "SYSTEM"

SQL> SELECT ENAME, JOB, SAL DEPTNO FROM NIRMALA.EMP WHERE DEPTNO=10
FOR UPDATE;

ENAME      JOB           DEPTNO
---------- --------- ----------
CLARK      MANAGER         2450
KING       PRESIDENT       5000
```

```
MILLER     CLERK           1300
```

User - 2 Terminal:

```
SQL> SHOW USER
USER is "NIRMALA"

SQL> UPDATE EMP SET SAL=3500 WHERE DEPTNO=10;
```

Note: we cannot perform update operation in terminal 2. Again, to release the lock we have to use either “COMMIT or ROLLBACK”.

User - 1 Terminal:

```
SQL> Rollback;

Rollback complete.
```

User - 2 Terminal:

```
SQL> UPDATE EMP SET SAL=3500 WHERE DEPTNO=10;

3 rows updated.
```

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.

User - 1 Terminal:

```
SQL> SHOW USER
USER is "SYSTEM"

SQL> UPDATE NIRMALA.EMP SET SAL=3300 WHERE EMPNO=7788;

1 row updated.
```

User - 2 Terminal:

```
SQL> SHOW USER
USER is "NIRMALA"

SQL> UPDATE EMP SET SAL=4400 WHERE EMPNO=7566;

1 row updated.
```

User - 1 Terminal:

```
SQL> UPDATE NIRMALA.EMP SET SAL=4400 WHERE EMPNO=7566;
```

User - 2 Terminal:

```
SQL> UPDATE EMP SET SAL=4400 WHERE EMPNO=7788;
```

User - 1 Terminal:

```
UPDATE NIRMALA.EMP SET SAL=4400 WHERE EMPNO=7566
                       *
ERROR at line 1:
ORA-00060: deadlock detected while waiting for resource


SQL> ROLLBACK;

Rollback complete.
```

User - 2 Terminal:

```
SQL> UPDATE EMP SET SAL=4400 WHERE EMPNO=7788;

1 row updated.
```

Table Level Locks:

- In this level we are locking a table (all rows). Oracle having two types of table level locking,
  - Share lock
  - 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 number of users are locks the resources.
- Updating data not allowed until the shared lock is released by using COMMIT/ ROLLBACK.

  Syntax:
  SQL> LOCK TABLE <table> IN SHARE MODE;

User - 1 Terminal:

```
SQL> SHOW USER
USER is "SYSTEM"

SQL> LOCK TABLE NIRMALA.EMP IN SHARE MODE;

Table(s) Locked.
```

```
SQL> ROLLBACK;

Rollback complete.
```

User - 2 Terminal:

```
SQL> SHOW USER
USER is "NIRMALA"

SQL> LOCK TABLE EMP IN SHARE MODE;

Table(s) Locked.

SQL> COMMIT;

Commit complete.
```

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 locks the resource.

  Syntax:
  SQL> LOCK TABLE <table name> IN EXCLUSIVE MODE;

User - 1 Terminal:

```
SQL> SHOW USER
USER is "SYSTEM"

SQL> LOCK TABLE NIRMALA.EMP IN EXCLUSIVE MODE;

Table(s) Locked.
```

User - 2 Terminal:

```
SQL> SHOW USER
USER is "NIRMALA"

SQL> LOCK TABLE EMP IN EXCLUSIVE MODE;
```

User - 1 Terminal:

```
SQL> COMMIT;

Commit complete.
```

User - 2 Terminal:

```
SQL> LOCK TABLE EMP IN EXCLUSIVE MODE;

Table(s) Locked.
```

Indexes

- Index is a database object which is used to retrieve the required row/ rows 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 very 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) column 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,
    - Table scan (default)
    - Index scan

Table scan:

- It is a default scanning mechanism for retrieving data from table.
- In this mechanism oracle server is scanning/ searching entire table (top - bottom).

Ex: WAQ to retrieve employee details whose salary is 3000.

```
SQL> SELECT COUNT(*) FROM EMP;

  COUNT(*)
----------
        14

SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE SAL=3000;

ENAME      JOB              SAL
---------- --------- ----------
SCOTT      ANALYST         3000
FORD       ANALYST         3000
```

Note: In this table scan we are comparing where condition 14 times.

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,

1. 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.

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

Table created.

SQL> CREATE TABLE TEST2(SNO INT PRIMARY KEY, NAME VARCHAR2(10));

Table created.
```

Note: If we want to view index name along with column name of a particular table then we use "USER_IND_COLUMNS" data dictionary.

```
SQL> DESC USER_IND_COLUMNS;

SQL> SELECT COLUMN_NAME, INDEX_NAME FROM USER_IND_COLUMNS WHERE
TABLE_NAME='TEST1';

COLUMN_NAME                    INDEX_NAME
-------------------            ---------------------------
SNO                            SYS_C007619

SQL> SELECT COLUMN_NAME, INDEX_NAME FROM USER_IND_COLUMNS WHERE
TABLE_NAME='TEST2';

COLUMN_NAME                    INDEX_NAME
-------------------            ---------------------------
SNO                             SYS_C007620
```

2. Manually/ explicitly
    - When user want to create an index object on a particular column/(s) then we follow the following syntax's,

      Syntax:
      SQL> CREATE INDEX <index name> ON <table name> (<column name>);

Types of indexes:

1. B - tree index (default index)
    a. Simple index
    b. Composite index
    c. Unique index
    d. Functional based index
2. Bitmap index

Simple index:

- When we created an index on a single column then we called as simple index.

Syntax:

SQL> CREATE INDEX <index name> ON <table name> (<column name>);

```
SQL> CREATE INDEX I1 ON EMP(SAL);

Index created.

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

COLUMN_NAME                    INDEX_NAME
-------------------            ---------------------------
SAL                            I1
EMPNO                          EMP_PK
SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE SAL=3000;

ENAME      JOB              SAL
---------- --------- ----------
SCOTT      ANALYST         3000
FORD       ANALYST         3000
```

B-tree (binary tree)

LP < |3000| >= RP (Root Level)

||

LP < |2975| >= RP

||

2850|*, 2450|*, 1600|*

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

1100|*, 950|*, 800|*

LP < |5000| >= RP (Parent Level)

||

3000|*, * -- (child level - stored)

Note: In index scan we are comparing 3 times. which is much faster than table

scan (14 times comparing). Here " * " is represent row id. Those many * those many rows.

Composite index:

- When we created an index on multiple columns then we called as composite index.

  Syntax:
  SQL> CREATE INDEX <index name> ON <table name> (<column name1>, <column name2>, ...);

```
SQL> CREATE INDEX I2 ON EMP(DEPTNO, JOB);

Index created.

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

COLUMN_NAME                    INDEX_NAME
-------------------            ---------------------------
SAL                            I1
DEPTNO                         I2
JOB                            I2
EMPNO                          EMP_PK
```

Note: Oracle server uses above index when "SELECT" query with where clause is based on leading column of index, i.e. DEPTNO.

- SELECT * FROM EMP WHERE DEPTNO=10; (index scan)
- SELECT * FROM EMP WHERE DEPTNO=10 AND JOB='CLERK'; (index scan)
- SELECT * FROM EMP WHERE JOB='CLERK'; (table scan)

```
SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE DEPTNO=10;

ENAME      JOB              SAL
---------- --------- ----------
MILLER     CLERK           1300
CLARK      MANAGER         2450
KING       PRESIDENT       5000

SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE DEPTNO=10 AND
JOB='CLERK';

ENAME      JOB              SAL
---------- --------- ----------
MILLER     CLERK           1300
```

Unique index:

- When we create an index based on "UNIQUE constraint" column is called unique index. Unique index does not allow duplicate values.

  Syntax:
  SQL> CREATE UNIQUE INDEX <index name> ON <table name> (<column name>);

```
SQL> SELECT * FROM DEPT;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON

SQL> CREATE UNIQUE INDEX UI ON DEPT(DNAME);

Index created.

SQL> SELECT COLUMN_NAME, INDEX_NAME FROM USER_IND_COLUMNS WHERE
TABLE_NAME='DEPT';

COLUMN_NAME                    INDEX_NAME
-------------------            --------------------------
DEPTNO                         DEPT_PK
DNAME                          UI

SQL> INSERT INTO DEPT VALUES(50, 'SALES', 'HYD');
INSERT INTO DEPT VALUES(50, 'SALES', 'HYD')
*
ERROR at line 1:
ORA-00001: unique constraint (NIRMALA.UI) violated
```

Functional based index:

- When we create an index based on function then we called as functional based index.

  Syntax:
  SQL> CREATE INDEX <index name> ON <table name>(<function name> (column name));

```
SQL> CREATE INDEX I3 ON EMP(UPPER(ENAME));
```

```
Index created.

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

COLUMN_NAME                    INDEX_NAME
-------------------            --------------------------
SAL                            I1
DEPTNO                         I2
JOB                            I2
SYS_NC00009$                   I3
EMPNO                          EMP_PK

SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE ENAME='scott';

no rows selected

SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE ENAME='SCOTT';

ENAME      JOB              SAL
---------- --------- ----------
SCOTT      ANALYST         3000
```

Bitmap index:

- Bitmap index is created on distinct values of a particular column. Generally, bitmap indexes are created on low cardinality of columns.
- Cardinality: It refers to the uniqueness of data values contained in particular column of table.
- 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 whereas 0 is represent condition is false.

Syntax:

SQL> CREATE BITMAP INDEX <index name> ON <table name>(<column name>);

How to find cardinality of a column:

$$\text{Cardinality of column} = \frac{\text{Number of distinct values of a Column}}{\text{Number of rows in a table}}$$

Cardinality of EMPNO = 14 / 14 = 1
Cardinality of JOB = 5 / 14 = 0.35 ------ (creating bit map index)

```
SQL> CREATE BITMAP INDEX BIT1 ON EMP(JOB);
```

```
Index created.

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


COLUMN_NAME                    INDEX_NAME
-------------------            ---------------------------
SAL                            I1
DEPTNO                         I2
JOB                            I2
SYS_NC00009$                   I3
JOB                            BIT1
EMPNO                          EMP_PK

SQL> SELECT ENAME, JOB, SAL FROM EMP WHERE JOB='CLERK';

ENAME      JOB              SAL
---------- --------- ----------
MILLER     CLERK           1300
SMITH      CLERK            800
ADAMS      CLERK           1100
JAMES      CLERK            950
```

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: If we want to view index name along with index type then we use "USER_INDEXES" data dictionary.

```
SQL> DESC USER_INDEXES;

SQL> SELECT INDEX_NAME, INDEX_TYPE FROM USER_INDEXES WHERE
TABLE_NAME='EMP';

INDEX_NAME                     INDEX_TYPE
-------------------------      -------------------------
EMP_PK                         NORMAL
I1                             NORMAL
I2                             NORMAL
```

```
I3                                FUNCTION-BASED NORMAL
BIT1                              BITMAP
```

How to drop an index:

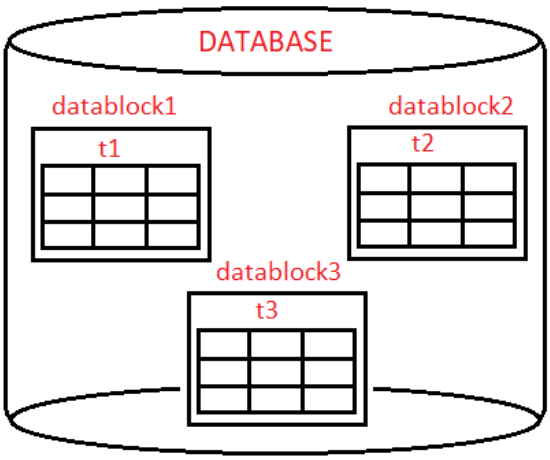
SQL> DROP INDEX <index name>;

```
SQL> DROP INDEX I1;

Index dropped.

SQL> DROP INDEX BIT1;

Index dropped.
```

Cluster

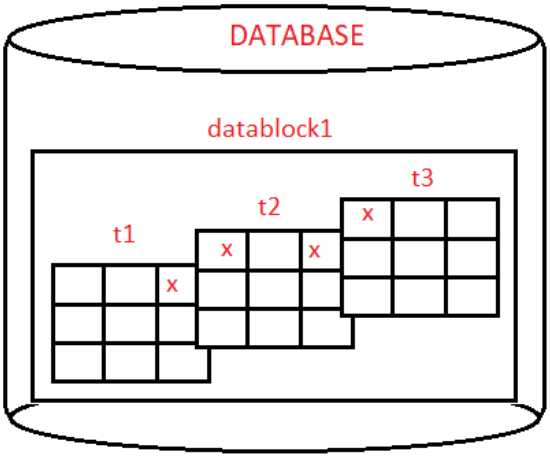
- Cluster is a DB object which contain group of tables together and also it shares same data block.
- Generally, cluster are used to improve performance of the joins and also clusters are created by DBA only.
- Cluster table must have a common column name. This common column is also called as cluster key. Generally, cluster are created at the time of table creation.

Non-Cluster Tables (Default)



To degrade joins performance

Cluster Tables



To improve joins performance

```
SELECT * FROM T1, T2, T3 WHERE
T1.CC=T2.CC AND T2.CC=T3.CC;
```

```
SELECT * FROM T1, T2, T3 WHERE
T1.CC=T2.CC AND T2.CC=T3.CC;
```

Steps to create cluster in oracle:

Step 1: Syntax to create a cluster:

SQL> CREATE CLUSTER <cluster name> (<common column name> <dt>[size]);

Step 2: Syntax to create an index cluster:

SQL> CREATE INDEX <index name> ON CLUSTER <cluster name>;

Step 3: Syntax to create cluster tables:

SQL> CREATE TABLE <table name>(<column1> <datatype>[size], <column2> <datatype>[size], ...) CLUSTER <cluster name> (common column name);

```
SQL> CONN
Enter user-name: system/tiger
Connected.

SQL> GRANT CREATE CLUSTER TO NIRMALA;

Grant succeeded.

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

SQL> CREATE CLUSTER EMP_DEPT(DEPTNO INT);

Cluster created.

SQL> CREATE INDEX EDI ON CLUSTER EMP_DEPT;

Index created.

SQL> CREATE TABLE EMP1(EID INT, ENAME VARCHAR2(10), DEPTNO INT)
CLUSTER EMP_DEPT(DEPTNO);

Table created.

SQL> INSERT INTO EMP1 VALUES(101, 'SMITH', 10);

1 row created.

SQL> INSERT INTO EMP1 VALUES(102, 'ALLEN', 20);

1 row created.

SQL> CREATE TABLE DEPT1(DEPTNO INT, DNAME VARCHAR2(10)) CLUSTER
EMP_DEPT(DEPTNO);

Table created.

SQL> INSERT INTO DEPT1 VALUES(10, 'SAP');

1 row created.
```

```
SQL> INSERT INTO DEPT1 VALUES(20, 'DBA');

1 row created.

SQL> SELECT * FROM EMP1;

       EID ENAME          DEPTNO
---------- ---------- ----------
       102 ALLEN              20
       101 SMITH              10

SQL> SELECT * FROM DEPT1;

    DEPTNO DNAME
---------- ----------
        20 DBA
        10 SAP
```

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
------------------
AAASTIAAHAAAAGDAAA
AAASTIAAHAAAAGHAAA

SQL> SELECT ROWID FROM DEPT1;

ROWID
------------------
AAASTIAAHAAAAGDAAA
AAASTIAAHAAAAGHAAA
```

Note: To view all cluster objects in oracle then we follow the following data dictionary 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 data dictionary is "USER_TABLES".

```
SQL> DESC USER_TABLES;

SQL> SELECT TABLE_NAME FROM USER_TABLES WHERE
CLUSTER_NAME='EMP_DEPT';

TABLE_NAME
----------------------------------------
EMP1
DEPT1
```

Dropping cluster with tables:

Syntax:

SQL> DROP CLUSTER <cluster name>;

```
SQL> DROP CLUSTER EMP_DEPT;
DROP CLUSTER EMP_DEPT
*
ERROR at line 1:
ORA-00951: cluster not empty
```

Note: 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.

SQL> SELECT TABLE_NAME FROM USER_TABLES WHERE
CLUSTER_NAME='EMP_DEPT';

no rows selected

SQL> SELECT * FROM EMP1;
SELECT * FROM EMP1
              *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL> SELECT * FROM DEPT1;
SELECT * FROM DEPT1
              *
ERROR at line 1:
ORA-00942: table or view does not exist
```

User-define Datatypes

- User define datatypes are introduced in oracle 8.0 version. When predefine 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.
    - Object type (or) Composite type
    - VARRAY
    - Nested table

Object type (or) Composite type:

- It allows group of values/ elements of different datatypes.

Syntax:

SQL> CREATE TYPE <type name> AS OBJECT (<column 1> datatype[size], <column 2> datatype[size], ...);
/

Note: We have to use "/" also, if you write the query and enter then it will ask for "/", so it is the part of syntax.

```
SQL> CONN
Enter user-name: system/tiger
Connected.

SQL> GRANT DBA TO NIRMALA;

Grant succeeded.

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

SQL> CREATE TYPE COURSE_TYPE AS OBJECT (CID NUMBER (4), CNAME
VARCHAR2(10), FEE NUMBER (10));
  2  /

Type created.

SQL> CREATE TABLE STUDENTS (SID NUMBER (4), SNAME VARCHAR2(10),
COURSE COURSE_TYPE);

Table created.
```

```
SQL> DESC STUDENTS;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------
 SID                                                NUMBER(4)
 SNAME                                              VARCHAR2(10)
 COURSE                                             COURSE_TYPE
```

To Insert:

```
SQL> INSERT INTO STUDENTS VALUES (101,'SAI', COURSE_TYPE
(1021,'ORACLE',1200));

1 row created.

SQL> INSERT INTO STUDENTS VALUES (102,'WARD', COURSE_TYPE
(1022,'C',500));

1 row created.
```

To Select:

```
SQL> SELECT * FROM STUDENTS;

       SID SNAME           COURSE(CID, CNAME, FEE)
---------- ----------      ---------------------------------------
       101 SAI             COURSE_TYPE(1021, 'ORACLE', 1200)
       102 WARD            COURSE_TYPE(1022, 'C', 500)

SQL> SELECT S.SID, S.SNAME, S.COURSE.CID, S.COURSE.CNAME,
S.COURSE.FEE FROM STUDENTS S;

       SID SNAME      COURSE.CID COURSE.CNA COURSE.FEE
---------- ---------- ---------- ---------- ----------
       101 SAI              1021 ORACLE           1200
       102 WARD             1022 C                 500

SQL> SELECT S.SID, S.SNAME, S.COURSE.CID CID, S.COURSE.CNAME CNAME,
S.COURSE.FEE FEE FROM STUDENTS S;

       SID SNAME             CID CNAME             FEE
---------- ---------- ---------- ---------- ----------
       101 SAI              1021 ORACLE           1200
       102 WARD             1022 C                 500
```

To update:

```
SQL> UPDATE STUDENTS S SET S.COURSE.FEE=2000 WHERE S.SID=101;

1 row updated.
```

To delete:

```
SQL> DELETE FROM STUDENTS S WHERE S.COURSE.CID=1022;

1 row deleted.
```

VARRAY:

- It allows group of values/ elements of same datatypes.
- It should declare with size.

Syntax:

SQL> CREATE TYPE <type name> IS VARRAY (size) OF <datatype>[size];
/

```
SQL> CREATE TYPE MBNO_ARRAY1 IS VARRAY (3) OF NUMBER (10);
  2  /

Type created.

SQL> CREATE TABLE EMPLOYEE (EMPNO NUMBER (4), MBNO MBNO_ARRAY1);

Table created.

SQL> INSERT INTO EMPLOYEE VALUES (1021, MBNO_ARRAY1(9703542749,
8502045789));

1 row created.

SQL> INSERT INTO EMPLOYEE VALUES (1022, MBNO_ARRAY1(9632587412,
8523691478, 7412356896));

1 row created.
```

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 declared with size.

Steps to create nested table:

Step1: Syntax to create an object type:

SQL> CREATE TYPE <type name> AS OBJECT (<column 1> datatype[size], <column2> datatype[size], ...);
/

Step2: Syntax to create nested table type:

SQL> CREATE TYPE <type name> AS TABLE OF <object type name>;

/

Step3: Syntax to create a table:

SQL> CREATE TABLE <table name>(<column 1> <datatype>[size], ..., <column n> <nested table type name>) NESTED TABLE <column n> STORE AS <any name>;

```
SQL> CREATE TYPE ADDR_TYPE AS OBJECT (HNO NUMBER (4), STREET
VARCHAR2(10), CITY VARCHAR2(10));
  2  /

Type created.

SQL> CREATE TYPE ADDR_ARRAY AS TABLE OF ADDR_TYPE;
  2  /

Type created.

SQL> CREATE TABLE CUSTOMER (CID NUMBER (4), CNAME VARCHAR2(10),
CADDRESS ADDR_ARRAY) NESTED TABLE CADDRESS STORE AS CUST_ADDR;

Table created.

SQL> INSERT INTO CUSTOMER VALUES (1,'SAI', ADDR_ARRAY (ADDR_TYPE
(1122,'GANDHI','HYD')));

1 row created.

SQL> INSERT INTO CUSTOMER VALUES (2,'WARD', ADDR_ARRAY (ADDR_TYPE
(1123,'ASHOK','CHE'), ADDR_TYPE (1124,'VASATI','MUM')));

1 row created.
```

Note: we can also select, update, delete, insert data within nested table by using the following syntax,

Syntax:

SQL> SELECT/ UPDATE/ DELETE/ INSERT (SELECT <nested table type column name> FROM <table name>);

```
SQL> SELECT * FROM TABLE (SELECT CADDRESS FROM CUSTOMER WHERE
CID=1);

       HNO STREET     CITY
---------- ---------- ----------
      1122 GANDHI     HYD
```

```
SQL> UPDATE TABLE (SELECT CADDRESS FROM CUSTOMER WHERE CID=2) SET
HNO=1024 WHERE HNO=1124;

1 row updated.

SQL> DELETE FROM TABLE (SELECT CADDRESS FROM CUSTOMER WHERE CID=2)
WHERE CITY='MUM';

1 row deleted.

SQL> INSERT INTO TABLE (SELECT CADDRESS FROM CUSTOMER WHERE CID=1)
VALUES (1124,'YUVIN','HYD');

1 row created.
```

Note: In oracle we want to view user types then follow the following data dictionary is "USER_TYPES".

```
SQL> DESC USER_TYPES;

SQL> SELECT TYPE_NAME FROM USER_TYPES;

TYPE_NAME
------------------------------------------------
MBNO_ARRAY1
ADDR_TYPE
ADDR_ARRAY
COURSE_TYPE
```

Syntax to drop type:

SQL> DROP TYPE <type name> FORCE;

```
SQL> DROP TYPE MBNO_ARRAY1;
DROP TYPE MBNO_ARRAY1
*
ERROR at line 1:
ORA-02303: cannot drop or replace a type with type or table
dependents

SQL> DROP TYPE MBNO_ARRAY1 FORCE;

Type dropped.

SQL> DESC EMPLOYEE;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------
 EMPNO                                              NUMBER(4)
```

Note: “FORCE” will delete the columns who are the type of that user define data type from the table.

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				
ROLLNO	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
105	WARNER	CSE	Mr. X	040-53337

- In the table above, we have data of 4 computer science 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.
- There are some other problems also are there like occupied more memory, data inconsistency, insertion, updating, deletion.

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

Updating 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 Updating problem.

Ex: Mr. X leaves and Mr. Y join as a new HOD for CSE. Then the table will be like below,

STUDENT DETAILS				
ROLLNO	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:

Problems:

STUDENT DETAILS				
ROLLNO	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		
ROLLNO	NAME	BRANCH (FK)
101	SAI	CSE
102	ALLEN	CSE
103	JAMES	CSE
104	MILLER	CSE
105	WARNER	CSE

BRANCH DETAILS		
ROLLNO	NAME	BRANCH (FK)
CSE	Mr. Y	040-53337

Note: By the above example we avoid insertion, deletion and updating problems.

Q. Where we want to use Normalization?
Ans. In DB design level by DB designer or DB architect.

Types of Normal Forms:

- Normalization can be achieved in multiple ways:
  - First Normal Form
  - Second Normal Form
  - Third Normal Form
  - BCNF
  - Fourth Normal Form
  - 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: ❌

Column 1	Column 2
A	X, Y
B	W, X
C	Y
D	Z

2. A column should contain values that are same datatype.
   Ex: ❌

NAME	DOB
SAI	19-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	19-JAN-92

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	ROY
3	JAMES	WARTON

Note: ❌ - means it not supporting that rule.

STUDENT		
ROLLNO	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>		
ROLLNO	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.

Q. What is dependency?

Ans. In a table if non-key columns (non-primary key) are depending on key column (primary key) then it is called as fully/ functional dependency.

Ex: (PK) A B C D

Here, “A” is key column → “B”, “C”, “D” are non-key columns.

STUDENT			
STUDENT_ID (PK)	NAME	BRANCH	ADDRESS
101	SAI	CSE	HYD
102	SAI	IT	MUM
103	JAMES	CSE	CHENNAI
104	MILLER	CSE	HYD

Note: A primary key column (STUDENT_ID) can be used to fetch data any column in the table.

Q. What is partial dependency?

Ans. In a table if non-key column depends on part of the key column, then it is called as partial dependency.

Ex: (Primary key (A, B)/ Composite primary key) A B C D

Here, “A and B” is a key column → “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.

SUBJECT	
SUBJECT_ID <PK>	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	ORACLE 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:

- ✓ In above score table," teacher column" is only depends on SUBJECT_ID but not on STUDENT_ID is called as "partial dependency".
- ✓ If there is no composite primary key on a table then there is no partial dependency.

Q. How to Remove partial dependency?

Ans. 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		
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.

SCORE		
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.

EX: (Composite Primary key) 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		
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_ID + SUBJECT_ID 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.

Q. How to remove Transitive Dependency?

Ans. Again, the solution is very simply taken 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			
STUDENT_ID	SUBJECT_ID	MARKS	EXAM_ID (FK)

EXAM		
EXAM_ID (PK)	EXAM_NAME	TOTAL_MARKS
1	Workshop	200
2	Mains	70
3	Practical's	30

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.
- 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 selecting candidate keys from those super keys.

STUDENT				
STUDENT_ID	NAME	BRANCH	MAILID	REG_NUMBER
101	SAI	CSE	sai@gmail.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:

- STUDENT_ID | STUDENT_ID + MAILID |

- MAILID | MAILID + REG_NUMBER | STUDENT_ID + MAILID + REG_NUMBER
- REG_NUMBER | REG_NUMBER + STUDENT_ID |

Ex. on candidate keys:

- STUDENT_ID
- MAILID
- REG_NUMBER

Note: Super key/ candidate key/ primary key all are same.

Boyce- Codd Normal Form (BCNF)

- For a table to satisfy the Boyce- Codd Normal Form, it should satisfy the following two conditions,
    1. It should be in the Third Normal Form.
    2. And, for any dependency A → B, A should be a super key.

(Composite Primary Key) COLLEGE_ENROLMENT		
STUDENT_ID (PK)	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.

Q. How to satisfy BCNF?

Ans. 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	
STUDENT_ID (PK)	PROFESSOR_ID
101	1
101	2

(Composite Primary Key) PROFESSOR		
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).

COLLEGE_ENROLMENT (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).

HOBBIE (4NF)	
STUDENT_ID	HOBBY
1	Cricket
1	Reading
1	Hockey

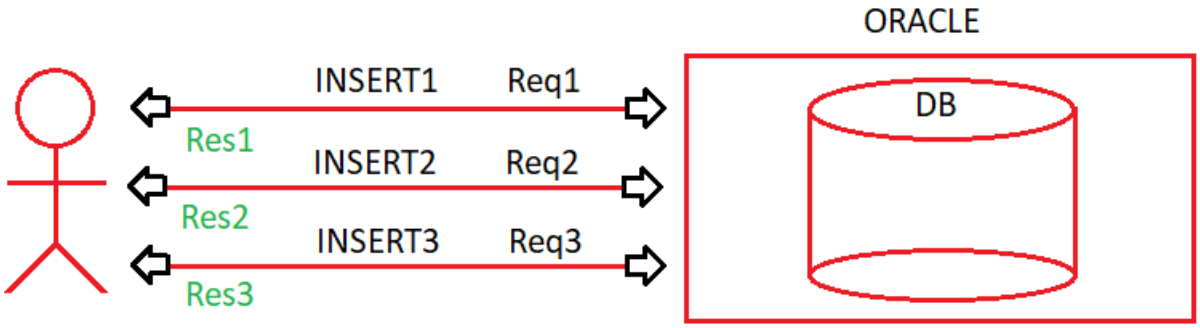
COURSE_OPTED (4NF)	
STUDENT_ID	COURSE
1	ORACLE
1	JAVA
1	C#

Fifth Normal Form (5NF)

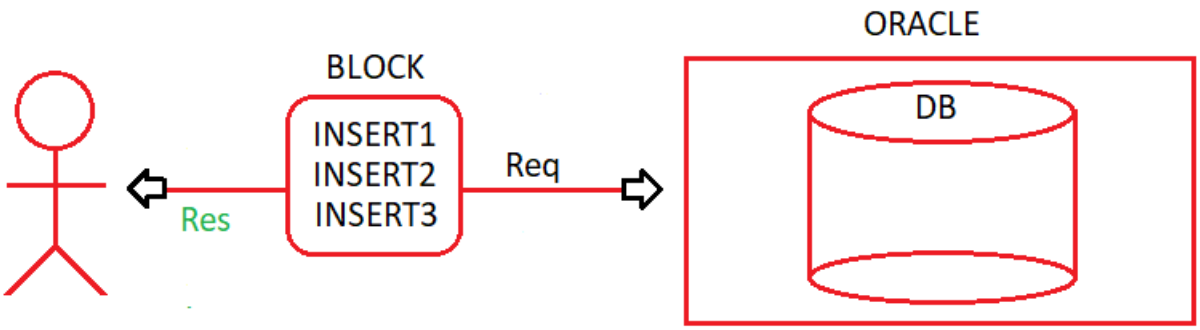
- If a table having multi valued attributes and also that table cannot decomposed into multiple tables is called as fifth normal form.
- Generally, in 4NF resource table some attributes are not logically related where as in 5NF resource table all attributes are related to one to another.
- Fifth normal 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/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 compiling and executing individually, so that number of compilations 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 number of compilations and improve performance of database.



[Dig: Single line statement execution process (SQL)]



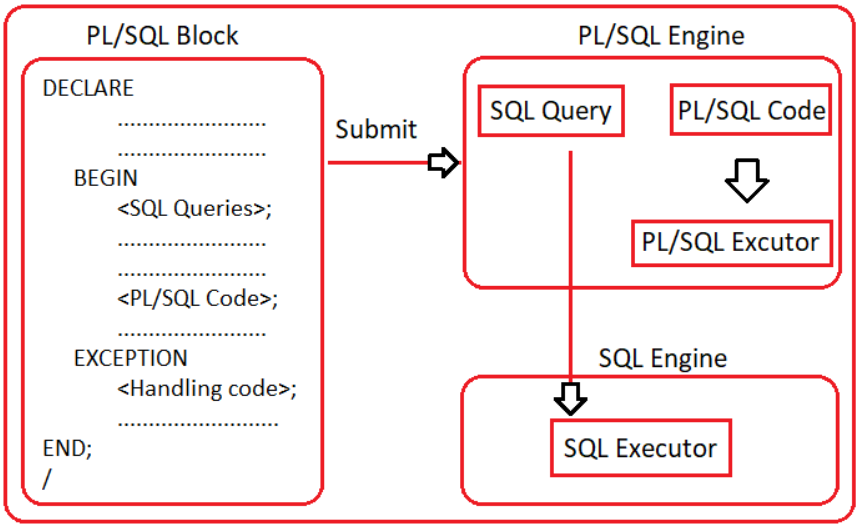
[Dig: Multi line statements execution process (PL/SQL)]

Features of PL/SQL:

- To improves performance.
- Supporting conditional & looping statements.
- Supporting reusability.
- Providing security because all programs are saved in database and authorized user can only access the programs.
- Supporting portability i.e. PL/SQL programs can be moved from one platform to another platform without any changes.
- Supporting exception handling mechanism.
- 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 blocking structure programming language. Which is having the following two engines those are
    - SQL engine
    - PL/SQL engine



[Dig: PL/SQL Architecture]

- Whenever we are submitting a PL/SQL block into oracle server then all SQL statements(queries) are separated and executing by SQL query executor with in sql engine. Whereas all PL/SQL statements (code) are separated and executing by PL/SQL code executor with in PL/SQL engine.

Q. What is a block?

Ans. 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

Difference B/W Anonymous & Sub block:

Anonymous Block	Sub Block
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 can't reusable.	3. It can reusable.
4. Every time compilation of code.	4. Precompiled code (First time compilation 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,
    a. Declaration block
    b. Execution block
    c. Exception block

Declaration block:

- This block starts with " DECLARE" statement.
- Declaring variables, cursors, user define exceptions.
- It is an optional block.

Execution block:

- This block starts with "BEGIN" statement & ends with "END" statement.
- Implementing SQL statements & logical code of a program (PL/SQL).
- It is a mandatory block.

Exception block:

- This block starts with "EXCEPTION" statement.
- Used for 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: Syntax to declaring a variable:

```
DECLARE
        <variable name> <dt>[size];
Ex:
        DECLARE
                A NUMBER (10) (or) A INT;
                B VARCHAR2(10);
```

Step2: Syntax to assigning/ storing a value into variable:

```
<variable name>: = <value>;
Ex:
        A:= 1021;
        B:= 'SAI';
```

Note: Here,

- ✓ ":=" means assignment operator in PL/SQL.
- ✓ "=" means comparison operator in PL/SQL.

Step3: Syntax to printing variables value

```
DBMS_OUTPUT.PUT_LINE (<variable name > (or) '<UD message>');
Ex:
        DBMS_OUTPUT.PUT_LINE(A);
        DBMS_OUTPUT.PUT_LINE(B);
```

Ex: WAPLSP to print "Welcome to PL/SQL" statement.

```
SQL> BEGIN
  2     DBMS_OUTPUT.PUT_LINE('Welcome to PL/SQL');
  3  END;
  4  /

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:
SQL> SET SERVEROUTPUT OFF/ ON;

Note: Here,

- ✓ OFF: it is default output is not display
- ✓ ON: output is display

```
SQL> SET SERVEROUTPUT ON;
SQL> /
WELCOME TO PL/SQL

PL/SQL procedure successfully completed.
```

Ex: WAPLSP to print variables values.

```
SQL> DECLARE
  2     X NUMBER(10);
  3     Y NUMBER(10);
  4  BEGIN
  5     X := 100;
  6     Y := 200;
  7     DBMS_OUTPUT.PUT_LINE('Variable values are : '||X||', '||Y);
  8  END;
  9  /
Variable values are : 100, 200

PL/SQL procedure successfully completed.
```

Ex: WAPLSP To print sum of two numbers at runtime.

```
SQL> DECLARE
  2     X NUMBER(2);
```

```
  3     Y NUMBER(2);
  4     Z NUMBER(10);
  5  BEGIN
  6     X := &X;
  7     Y := &Y;
  8     Z := X+Y;
  9     DBMS_OUTPUT.PUT_LINE(Z);
 10  END;
 11  /
Enter value for x: 10
old   6:        X := &X;
new   6:        X := 10;
Enter value for y: 20
old   7:        Y := &Y;
new   7:        Y := 20;
30

PL/SQL procedure successfully completed.
```

Verify:

- ON = display old, new bind variable statements.
- OFF = doesn't display old, new bind variables statements.

Syntax:

SQL> SET VERIFY ON / OFF;

```
SQL> SET VERIFY OFF;
SQL> /
Enter value for x: 40
Enter value for y: 50
90

PL/SQL procedure successfully completed.
```

SELECT ..... INTO statement:

- Storing a table column values into variables.
- Returns a single row (or) a single value
- We can use in execution block.

Syntax:

SELECT <column name1>, <column name2>, ... INTO <variable name1>, <variable name2>……. FROM <table name> [WHERE <condition>];

Ex: WAPLSP to display ENAME, SALARY details from EMP table as per the given EMPNO by using SELECT ..... INTO statement?

```
SQL> DECLARE
  2     V_NAME VARCHAR2(10);
  3     V_SAL NUMBER(10);
  4  BEGIN
  5     SELECT ENAME, SAL INTO V_NAME, V_SAL FROM EMP WHERE
EMPNO=&EMPNO;
  6     DBMS_OUTPUT.PUT_LINE(V_NAME||', '||V_SAL);
  7  END;
  8  /
Enter value for empno: 7902
FORD, 3000

PL/SQL procedure successfully completed.
```

Ex: WAPLSP to fetch maximum salary of emp table by using "SELECT …… INTO" statement?

```
SQL> DECLARE
  2     V_MAX_SAL NUMBER(10);
  3  BEGIN
  4     SELECT MAX(SAL) INTO V_MAX_SAL FROM EMP;
  5     DBMS_OUTPUT.PUT_LINE('Maximum salary - '||V_MAX_SAL);
  6  END;
  7  /
Maximum salary - 5000

PL/SQL procedure successfully completed.
```

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 allocating 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 is 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,
    - Column level attributes
    - Row level attributes

Column level attributes:

- In this level we are defining variables attributes for individual columns.
- It is representing with "%TYPE" statement.

Syntax:

<variable name> <table name>.<column name>%TYPE;

```
SQL> DECLARE
  2     V_ENAME EMP.ENAME%TYPE;
  3     V_SAL EMP.SAL%TYPE;
  4  BEGIN
  5     SELECT ENAME, SAL INTO V_ENAME, V_SAL FROM EMP WHERE
EMPNO=&EMPNO;
  6     DBMS_OUTPUT.PUT_LINE(V_ENAME||', '||V_SAL);
  7  END;
  8  /
Enter value for empno: 7902
FORD, 3000

PL/SQL procedure successfully completed.
```

Row level attributes:

- In this level we are declaring a single variable will represent all different datatypes of columns in a table.
- It represents with "%ROWTYPE".

Syntax:

<variable name> <table name>%ROWTYPE;

```
SQL> DECLARE
  2     I EMP%ROWTYPE;
  3  BEGIN
  4     SELECT ENAME, SAL INTO I.ENAME, I.SAL FROM EMP WHERE
EMPNO=&EMPNO;
  5     DBMS_OUTPUT.PUT_LINE(I.ENAME||', '||I.SAL);
  6  END;
  7  /
Enter value for empno: 7902
FORD, 3000

PL/SQL procedure successfully completed.

SQL> DECLARE
  2     I EMP%ROWTYPE;
  3  BEGIN
  4     SELECT * INTO I FROM EMP WHERE EMPNO=&EMPNO;
```

```
  5     DBMS_OUTPUT.PUT_LINE(I.ENAME||', '||I.SAL||', '||I.DEPTNO);
  6  END;
  7  /
Enter value for empno: 7902
FORD, 3000, 20

PL/SQL procedure successfully completed.
```

Control Structures

- It is used to control flow of the program.
- There are three types of control structures.
    1. Conditional control structures
    2. Branching control structures
    3. Iteration control structures

Conditional control structures:

- IF: It contains only true block.
  Syntax:
  IF <condition> THEN
  <exec-statements>; -- true block
  END IF;

- IF … ELSE: It contains both true block & false block.
  Syntax:
  IF <condition> THEN
  <exec-statements>; -- true block
  ELSE
  <exec-statements>; -- false block
  END IF;

- 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;
```

- IF … ELSE ladder:
  Syntax:

```
IF <condition> THEN
        <exec-statements>;
ELSEIF <condition> then
        <exec-statements>;
ELSEIF <condition> then
        <exec-statements>
.....................
ELSE
        <exec-statements>;
END IF;
```

Branching control structures:

- Case:
  Syntax:

```
CASE <variable/expression>
WHEN <condition> then
        <exec-statements>;
WHEN <condition> then
        <exec-statements>;
WHEN <condition> then
        <exec-statements>;
ELSE
        <exec-statement>;
END CASE;
```

Iteration control statements:

- 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;
```

- While loop:
  Syntax:
  WHILE <condition>
  LOOP
  <exec-statements>;
  <increment /decrement>;
  END LOOP;

- 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 temporary memory/ a private SQL area (PSA)/ a work space.
- Cursor are two types, those are:
  1. Explicit cursor (user define cursor)
  2. Implicit cursor (system define cursor)

Explicit Cursor:

- These cursors 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,
  - Declaring a cursor
  - Open the cursor
  - Fetch rows from the cursor
  - Close the cursor

Steps to create explicit cursor:

1. Declaring a cursor: In this process we define a cursor.

   Syntax:
   DECLARE CURSOR <cursor name> IS < select query>;

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.

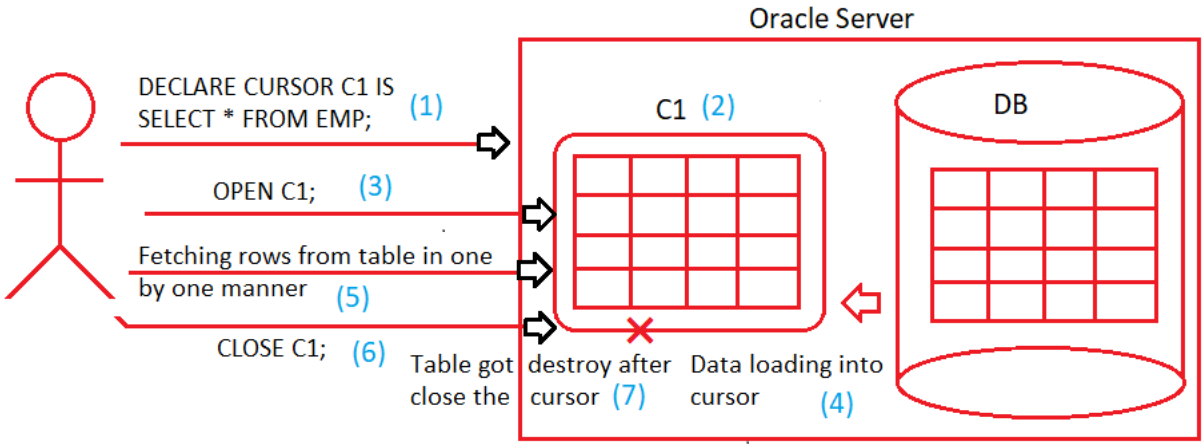
Syntax:
OPEN < cursor name>;

3. Fetching data from the cursor: In this process we access row by row from cursor.

   Syntax:
   FETCH <cursor name> INTO <variables>;

4. Closing a cursor: In this process, it releases the current result set of the cursor leaving the data structure available for reopening.

   Syntax:
   CLOSE <cursor name>;



[Dig: Working flow of cursor]

Attributes of explicit cursors:

- It shows status of the cursor and it returns boolean value

  Syntax:
  <cursor name>%<attribute>;

- %ISOPEN:
  - This is a default attribute.
  - It returns true, when the cursor connection is opens successfully.

- %FOUND:
  - It returns true, when the cursor contains data or data found in cursor memory.
  - It returns false, when data not found in cursor memory.

- %NOTFOUND:
  - It returns true, when data not found in cursor memory.
  - It returns false, when data found in cursor memory.

- %ROWCOUNT:
  - It returns number of fetch statements executed and the return type is number.

Ex: WAPLSP to fetch a single row from EMP table using cursor.

```
SQL> DECLARE
  2     CURSOR C1 IS SELECT ENAME, SAL FROM EMP;
  3     V_ENAME VARCHAR2(10);
  4     V_SAL NUMBER(10);
  5  BEGIN
  6     OPEN C1;
  7             FETCH C1 INTO V_ENAME, V_SAL;
  8             DBMS_OUTPUT.PUT_LINE(V_ENAME||', '||V_SAL);
  9     CLOSE C1;
 10  END;
 11  /
 SMITH,800

PL/SQL procedure successfully completed.
```

Ex: WAPLSP to fetch multiple rows from DEPT table by using cursor.

By using simple loop:

```
SQL> DECLARE
  2     CURSOR C1 IS SELECT * FROM DEPT;
  3     V_DEPTNO NUMBER(10);
  4     V_DNAME VARCHAR2(10);
  5     V_LOC VARCHAR2(10);
  6  BEGIN
  7     OPEN C1;
  8     LOOP
  9         FETCH C1 INTO V_DEPTNO, V_DNAME, V_LOC;
 10         EXIT WHEN C1%NOTFOUND;
 11         DBMS_OUTPUT.PUT_LINE(V_DEPTNO||', '||V_DNAME||',
'||V_LOC);
 12     END LOOP;
 13     CLOSE C1;
 14  END;
 15  /
10, ACCOUNTING, NEW YORK
```

```
20, RESEARCH, DALLAS
30, SALES, CHICAGO
40, OPERATIONS, BOSTON

PL/SQL procedure successfully completed.
```

By using while loop:

```
SQL> DECLARE
  2     CURSOR C1 IS SELECT * FROM DEPT;
  3     V_DEPTNO NUMBER(10);
  4     V_DNAME VARCHAR2(10);
  5     V_LOC VARCHAR2(10);
  6  BEGIN
  7     OPEN C1;
  8         FETCH C1 INTO V_DEPTNO, V_DNAME, V_LOC;
  9         WHILE(C1%FOUND)
 10         LOOP
 11             DBMS_OUTPUT.PUT_LINE(V_DEPTNO||', '||V_DNAME||',
'||V_LOC);
 12             FETCH C1 INTO V_DEPTNO, V_DNAME, V_LOC;
 13         END LOOP;
 14     CLOSE C1;
 15  END;
 16  /
10, ACCOUNTING, NEW YORK
20, RESEARCH, DALLAS
30, SALES, CHICAGO
40, OPERATIONS, BOSTON

PL/SQL procedure successfully completed.
```

By using for loop:

```
SQL> DECLARE
  2     CURSOR C1 IS SELECT * FROM DEPT;
  3  BEGIN
  4     FOR I IN C1
  5     LOOP
  6         DBMS_OUTPUT.PUT_LINE(I.DEPTNO||', '||I.DNAME||',
'||I.LOC);
  7     END LOOP;
  8  END;
  9  /
10, ACCOUNTING, NEW YORK
20, RESEARCH, DALLAS
30, SALES, CHICAGO
40, OPERATIONS, BOSTON

PL/SQL procedure successfully completed.
```

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 number of times depends on number 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.

Ex: WAPLSP to fetch top five highest salaries employee rows from EMP table using cursor.

```
SQL> DECLARE
  2     CURSOR C1 IS SELECT ENAME, SAL FROM EMP ORDER BY SAL DESC;
  3     V_ENAME VARCHAR2(10);
  4     V_SAL NUMBER(10);
  5  BEGIN
  6     OPEN C1;
  7     LOOP
  8         FETCH C1 INTO V_ENAME, V_SAL;
  9         EXIT WHEN C1%ROWCOUNT>5;
 10         DBMS_OUTPUT.PUT_LINE(V_ENAME||', '||V_SAL);
 11     END LOOP;
 12     CLOSE C1;
 13  END;
 14  /
KING, 5000
FORD, 3000
SCOTT, 3000
JONES, 2975
BLAKE, 2850

PL/SQL procedure successfully completed.
```

Ex: WAPLSP using cursor to fetch even position rows from EMP table.

```
SQL> DECLARE
  2     CURSOR C1 IS SELECT EMPNO, ENAME FROM EMP;
  3     V_ENAME VARCHAR2(10);
  4     V_EMPNO NUMBER(10);
  5  BEGIN
  6     OPEN C1;
  7     LOOP
  8         FETCH C1 INTO V_EMPNO, V_ENAME;
  9         EXIT WHEN C1%NOTFOUND;
```

```
 10         IF MOD(C1%ROWCOUNT, 2) = 0 THEN
 11             DBMS_OUTPUT.PUT_LINE(V_EMPNO||', '||V_ENAME);
 12         END IF;
 13     END LOOP;
 14     CLOSE C1;
 15  END;
 16  /
7499, ALLEN
7566, JONES
7698, BLAKE
7788, SCOTT
7844, TURNER
7900, JAMES
7934, MILLER

PL/SQL procedure successfully completed.
```

Ex: WAPLSP using cursor to fetch 9th position row from EMP table?

```
SQL> DECLARE
  2     CURSOR C1 IS SELECT EMPNO, ENAME FROM EMP;
  3     V_ENAME VARCHAR2(10);
  4     V_EMPNO NUMBER(10);
  5  BEGIN
  6     OPEN C1;
  7     LOOP
  8         FETCH C1 INTO V_EMPNO, V_ENAME;
  9         EXIT WHEN C1%NOTFOUND;
 10         IF C1%ROWCOUNT = 9 THEN
 11             DBMS_OUTPUT.PUT_LINE(V_EMPNO||', '||V_ENAME);
 12         END IF;
 13     END LOOP;
 14     CLOSE C1;
 15  END;
 16  /
7839, KING

PL/SQL procedure successfully completed.
```

Parameterized Cursors:

- Whenever we are passing parameters to the cursor at the time declaration is called as parameterized cursor. These parameterized cursors want to declare then we follow the following two steps are

Step1: Syntax to declare parameterized cursor:

DECLARE CURSOR <cursor name> (<parameter name> <datatype>, ...) IS <select query>;

Step2: Syntax to open parameterized cursor:

OPEN <cursor name> (<parameter name>/ <value>);

Ex: WAPLSP using cursor to accept deptno as a parameter and display the no. of employee working in the given DEPTNO from EMP table?

```
SQL> DECLARE
  2     CURSOR C1(P_DEPTNO NUMBER) IS SELECT ENAME, DEPTNO FROM EMP
WHERE DEPTNO=P_
DEPTNO;
  3     V_ENAME VARCHAR2(10);
  4     V_DEPTNO NUMBER(10);
  5  BEGIN
  6     OPEN C1(&P_DEPTNO);
  7     LOOP
  8         FETCH C1 INTO V_ENAME, V_DEPTNO;
  9         EXIT WHEN C1%NOTFOUND;
 10         DBMS_OUTPUT.PUT_LINE(V_ENAME||', '||V_DEPTNO);
 11     END LOOP;
 12     CLOSE C1;
 13  END;
 14  /
Enter value for p_deptno: 10
CLARK, 10
KING, 10
MILLER, 10

PL/SQL procedure successfully completed.
```

Ex: WAPLSP using cursor to accept EMPNO as a parameter and check that employee is exists or not exists in EMP table.

```
SQL> DECLARE
  2     CURSOR C1(P_EMPNO NUMBER) IS SELECT ENAME FROM EMP WHERE
EMPNO=P_EMPNO;
  3     V_ENAME VARCHAR2(10);
  4  BEGIN
  5     OPEN C1(&P_EMPNO);
  6     FETCH C1 INTO V_ENAME;
  7     IF C1%FOUND THEN
  8        DBMS_OUTPUT.PUT_LINE('Employee exists, name is :
'||V_ENAME);
  9     ELSE
 10        DBMS_OUTPUT.PUT_LINE('Employee not exists');
 11     END IF;
 12     CLOSE C1;
```

```
 13  END;
 14  /
Enter value for p_empno: 7788
Employee exists, name is : SCOTT

PL/SQL procedure successfully completed.
```

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 successful or not.

Attributes of implicit cursors:

- It shows status of the cursor and it returns boolean value

  Syntax:
  SQL%<attribute>;

- %ISOPEN:
  - It returns true when cursor is successfully open otherwise returns false.

- %FOUND:
  - It returns true when last DML operation is executed successfully otherwise returns false.

- %NOTFOUND:
  - It returns true when last DML operation is not executed otherwise returns false.

- %ROWCOUNT:
  - It returns number of rows affected by last DML command.

```
SQL> DECLARE
  2     V_EMPNO NUMBER(10);
  3  BEGIN
  4     V_EMPNO := &V_EMPNO;
  5     DELETE FROM EMP WHERE EMPNO=V_EMPNO;
  6     IF SQL%FOUND THEN
  7        DBMS_OUTPUT.PUT_LINE('Record is deleted');
  8     ELSE
  9        DBMS_OUTPUT.PUT_LINE('Record is not exist');
```

```
 10     END IF;
 11  END;
 12  /
Enter value for v_empno: 7788
Record is deleted

PL/SQL procedure successfully completed.
```

Ref. Cursors/ Dynamic Cursor:

- When we assign "select statement" at the time of opening cursor is called as "Ref. cursor".
- Ref. cursors are two types. Those are,
    - Weak ref. cursor
    - Strong ref. cursor

Weak Ref. Cursor:

- When we declare ref. cursor without return types is called as weak ref. cursor.

  Syntax:
  <cursor variable name> SYS_REFCURSOR;

On single table:

```
SQL> DCLARE
SP2-0042: unknown command "DCLARE" - rest of line ignored.
SQL>
SQL> DECLARE
  2     C1 SYS_REFCURSOR;
  3     I EMP%ROWTYPE;
  4  BEGIN
  5     OPEN C1 FOR SELECT * FROM EMP WHERE DEPTNO=10;
  6     LOOP
  7        FETCH C1 INTO I;
  8             EXIT WHEN C1%NOTFOUND;
  9             DBMS_OUTPUT.PUT_LINE(I.EMPNO||', '||I.ENAME||',
'||I.SAL||', '||I.D
EPTNO);
 10     END LOOP;
 11     CLOSE C1;
 12  END;
 13  /
7782, CLARK, 2450, 10
7839, KING, 5000, 10
7934, MILLER, 1300, 10

PL/SQL procedure successfully completed.
```

On multiple table:

```
SQL> DECLARE
  2     C1 SYS_REFCURSOR;
  3     I EMP%ROWTYPE;
  4     J DEPT%ROWTYPE;
  5     V_DEPTNO NUMBER(10) := &V_DEPTNO;
  6  BEGIN
  7     IF V_DEPTNO = 10 THEN
  8          OPEN C1 FOR SELECT * FROM EMP WHERE DEPTNO=10;
  9          LOOP
 10             FETCH C1 INTO I;
 11             EXIT WHEN C1%NOTFOUND;
 12             DBMS_OUTPUT.PUT_LINE(I.EMPNO||', '||I.ENAME||',
'||I.SAL||', '||I.D
EPTNO);
 13          END LOOP;
 14     ELSIF V_DEPTNO = 20 THEN
 15          OPEN C1 FOR SELECT * FROM DEPT WHERE DEPTNO=20;
 16          LOOP
 17             FETCH C1 INTO J;
 18             EXIT WHEN C1%NOTFOUND;
 19             DBMS_OUTPUT.PUT_LINE(J.DEPTNO||', '||J.DNAME||',
'||J.LOC);
 20          END LOOP;
 21     END IF;
 22  END;
 23  /
Enter value for v_deptno: 20
20, RESEARCH, DALLAS

PL/SQL procedure successfully completed.
```

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>;

On single table:

```
SQL> DECLARE
  2     TYPE UD_REFCURSOR IS REF CURSOR RETURN EMP%ROWTYPE;
  3     C1 UD_REFCURSOR;
  4     I EMP%ROWTYPE;
  5  BEGIN
  6     OPEN C1 FOR SELECT * FROM EMP WHERE DEPTNO=10;
```

```
  7     LOOP
  8         FETCH C1 INTO I;
  9         EXIT WHEN C1%NOTFOUND;
 10             DBMS_OUTPUT.PUT_LINE(I.EMPNO||', '||I.ENAME||',
'||I.SAL||', '||I.DEPTNO);
 11     END LOOP;
 12     CLOSE C1;
 13  END;
 14  /
7782, CLARK, 2450, 10
7839, KING, 5000, 10
7934, MILLER, 1300, 10

PL/SQL procedure successfully completed.
```

On multiple table:

```
SQL> DECLARE
  2     TYPE UD_REFCURSOR IS REF CURSOR RETURN EMP%ROWTYPE;
  3     C1 UD_REFCURSOR;
  4     I EMP%ROWTYPE;
  5     J DEPT%ROWTYPE;
  6     V_DEPTNO NUMBER(10) := &V_DEPTNO;
  7  BEGIN
  8     IF V_DEPTNO = 10 THEN
  9          OPEN C1 FOR SELECT * FROM EMP WHERE DEPTNO=10;
 10          LOOP
 11             FETCH C1 INTO I;
 12             EXIT WHEN C1%NOTFOUND;
 13             DBMS_OUTPUT.PUT_LINE(I.EMPNO||', '||I.ENAME||',
'||I.SAL||', '||I.DEPTNO);
 14          END LOOP;
 15     ELSIF V_DEPTNO = 20 THEN
 16          OPEN C1 FOR SELECT * FROM DEPT WHERE DEPTNO=20;
 17          LOOP
 18             FETCH C1 INTO J;
 19             DBMS_OUTPUT.PUT_LINE(J.DEPTNO||', '||J.DNAME||',
'||J.LOC);
 20          END LOOP;
 21     END IF;
 22  END;
 23  /
Enter value for v_deptno: 20
          OPEN C1 FOR SELECT * FROM DEPT WHERE DEPTNO=20;
                      *
ERROR at line 16:
ORA-06550: line 16, column 19:
PLS-00382: expression is of wrong type
ORA-06550: line 16, column 7:
PL/SQL: SQL Statement ignored
```

```
ORA-06550: line 18, column 10:
PLS-00394: wrong number of values in the INTO list of a FETCH
statement
ORA-06550: line 18, column 10:
PL/SQL: SQL Statement ignored
```

Q. What are the differences b/w weak and strong ref. cursor?

Ans.

Weak ref cursor	Strong ref cursor
1. It is not declared with "return"	1. Declaring with "return" type.
2. Pre-define type is available	2. Pre-define type is not available that's why we are creating "used define type".
3. It can access rows from any of the table (more than one table)	3. It can access rows of a specific table only.

If we create 2 strong ref. cursor then we can use 2 table:

```
SQL> DECLARE
  2     TYPE UD_REFCURSOR IS REF CURSOR RETURN EMP%ROWTYPE;
  3     TYPE UD_REFCURSOR2 IS REF CURSOR RETURN DEPT%ROWTYPE;
  4     C1 UD_REFCURSOR;
  5     C2 UD_REFCURSOR2;
  6     I EMP%ROWTYPE;
  7     J DEPT%ROWTYPE;
  8     V_DEPTNO NUMBER(10) := &V_DEPTNO;
  9  BEGIN
 10     IF V_DEPTNO = 10 THEN
 11          OPEN C1 FOR SELECT * FROM EMP WHERE DEPTNO=10;
 12          LOOP
 13             FETCH C1 INTO I;
 14             EXIT WHEN C1%NOTFOUND;
 15             DBMS_OUTPUT.PUT_LINE(I.EMPNO||', '||I.ENAME||',
'||I.SAL||', '||I.DEPTNO);
 16          END LOOP;
 17     ELSIF V_DEPTNO = 20 THEN
 18          OPEN C2 FOR SELECT * FROM DEPT WHERE DEPTNO=20;
 19          LOOP
 20             FETCH C2 INTO J;
 21             EXIT WHEN C2%NOTFOUND;
 22             DBMS_OUTPUT.PUT_LINE(J.DEPTNO||', '||J.DNAME||',
'||J.LOC);
 23          END LOOP;
 24     END IF;
 25  END;
 26  /
```

```
Enter value for v_deptno: 20
20, RESEARCH, DALLAS

PL/SQL procedure successfully completed.
```

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
  - System/ pre-defined exception
  - User defined exception

Syntax:

```
Declare
        < variables, cursor, user define exception>;
Begin
        <statements>;
        Exception
                When <exception name> then
                <error statements>;
End;
```

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 are,
  - NO_DATA_FOUND
  - TOO_MANY_ROWS
  - ZERO_DIVIDE
  - INVALID _CURSOR
  - CURSOR_ALREADY_OPEN

    Etc.

NO_DATA_FOUND:

- Whenever PL/SQL block carry the SELECT …. INTO clause and if required

data not available in a table then oracle server returns an exception like ORA-1403: no data found.

```
SQL> DECLARE
  2     V_ENAME VARCHAR2(10);
  3     V_SAL NUMBER(10);
  4  BEGIN
  5     SELECT ENAME, SAL INTO V_ENAME, V_SAL FROM EMP WHERE
EMPNO=&EMPNO;
  6     DBMS_OUTPUT.PUT_LINE(V_ENAME||', '||V_SAL);
  7  END;
  8  /
Enter value for empno: 3524254
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 5
```

- To handle this exception oracle provided “NO_DATA_FOUND” exception.

```
SQL> DECLARE
  2     V_ENAME VARCHAR2(10);
  3     V_SAL NUMBER(10);
  4  BEGIN
  5     SELECT ENAME, SAL INTO V_ENAME, V_SAL FROM EMP WHERE
EMPNO=&EMPNO;
  6     DBMS_OUTPUT.PUT_LINE(V_ENAME||', '||V_SAL);
  7     EXCEPTION
  8         WHEN NO_DATA_FOUND THEN
  9         DBMS_OUTPUT.PUT_LINE('Record is not found');
 10  END;
 11  /
Enter value for empno: 9746454
Record is not found

PL/SQL procedure successfully completed.
```

TOO_MANY_ROWS:

- When SELECT …. INTO clause try to return more than one value or one row then oracle server returns an error like ORA-1422: exact fetch returns more than requested number of rows.

```
SQL> DECLARE
  2     V_SAL NUMBER(10);
  3  BEGIN
  4     SELECT SAL INTO V_SAL FROM EMP;
```

```
  5     DBMS_OUTPUT.PUT_LINE(V_SAL);
  6  END;
  7  /
DECLARE
*
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 4
```

- To handle for this error oracle, provide “TOO_MANY_ROWS” exception.

```
SQL> DECLARE
  2     V_SAL NUMBER(10);
  3  BEGIN
  4     SELECT SAL INTO V_SAL FROM EMP;
  5     DBMS_OUTPUT.PUT_LINE(V_SAL);
  6     EXCEPTION
  7         WHEN TOO_MANY_ROWS THEN
  8         DBMS_OUTPUT.PUT_LINE('Fetching more than one');
  9  END;
 10  /
Fetching more than one

PL/SQL procedure successfully completed.
```

ZERO_DIVIDE:

- In oracle when we are tried to perform division with zero then oracle return an error like ORA-1476: divisor is equal to zero.

```
SQL> DECLARE
  2     X NUMBER(10);
  3     Y NUMBER(10);
  4     Z NUMBER(10);
  5  BEGIN
  6     X:=&X;
  7     Y:=&Y;
  8     Z:=X/Y;
  9     DBMS_OUTPUT.PUT_LINE('Result - '||Z);
 10  END;
 11  /
Enter value for x: 25
Enter value for y: 0
DECLARE
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at line 8
```

- To handle for this error oracle, provide “ZERO_DIVIDE” exception

```
SQL> DECLARE
  2     X NUMBER(10);
  3     Y NUMBER(10);
  4     Z NUMBER(10);
  5  BEGIN
  6     X:=&X;
  7     Y:=&Y;
  8     Z:=X/Y;
  9     DBMS_OUTPUT.PUT_LINE('Result - '||Z);
 10     EXCEPTION
 11         WHEN ZERO_DIVIDE THEN
 12         DBMS_OUTPUT.PUT_LINE('Second number should not be 0');
 13  END;
 14  /
Enter value for x: 10
Enter value for y: 0
Second number should not be zero

PL/SQL procedure successfully completed.
```

INVALID_CURSOR:

- When we are not opening the cursor but we are trying to perform operations on cursor then oracle returns an error like ORA-1001: invalid cursor.

```
SQL> DECLARE
  2     CURSOR C1 IS SELECT EMPNO, ENAME, SAL FROM EMP;
  3     V_EMPNO NUMBER(10);
  4     V_ENAME VARCHAR2(10);
  5     V_SAL NUMBER(10);
  6  BEGIN
  7     FETCH C1 INTO V_EMPNO, V_ENAME, V_SAL;
  8     DBMS_OUTPUT.PUT_LINE(V_EMPNO||', '||V_ENAME||', '||V_SAL);
  9     CLOSE C1;
 10  END;
 11  /
DECLARE
*
ERROR at line 1:
ORA-01001: invalid cursor
ORA-06512: at line
```

- To handle this error oracle, provide “INVALID_CURSOR” exception.

```
SQL> DECLARE
```

```
  2     CURSOR C1 IS SELECT EMPNO, ENAME, SAL FROM EMP;
  3     V_EMPNO NUMBER(10);
  4     V_ENAME VARCHAR2(10);
  5     V_SAL NUMBER(10);
  6  BEGIN
  7     FETCH C1 INTO V_EMPNO, V_ENAME, V_SAL;
  8     DBMS_OUTPUT.PUT_LINE(V_EMPNO||', '||V_ENAME||', '||V_SAL);
  9     CLOSE C1;
 10     EXCEPTION
 11         WHEN INVALID_CURSOR THEN
 12         DBMS_OUTPUT.PUT_LINE('First you must open the cursor');
 13  END;
 14  /
First you must open the cursor

PL/SQL procedure successfully completed.
```

CURSOR_ALREADY_OPEN:

- Before reopening the cursor, we must close the cursor properly otherwise oracle returns an error like ORA-6511: cursor already open.

```
SQL> DECLARE
  2     CURSOR C1 IS SELECT EMPNO, ENAME, SAL FROM EMP WHERE
DEPTNO=10;
  3     V_EMPNO NUMBER(10);
  4     V_ENAME VARCHAR2(10);
  5     V_SAL NUMBER(10);
  6  BEGIN
  7     OPEN C1;
  8     LOOP
  9     FETCH C1 INTO V_EMPNO, V_ENAME, V_SAL;
 10     EXIT WHEN C1%NOTFOUND;
 11         DBMS_OUTPUT.PUT_LINE(V_EMPNO||', '||V_ENAME||',
'||V_SAL);
 12     END LOOP;
 13     OPEN C1;
 14  END;
 15  /

7782, CLARK, 2450
7839, KING, 5000
7934, MILLER, 1300
DECLARE
*
ERROR at line 1:
ORA-06511: PL/SQL: cursor already open
ORA-06512: at line 2
ORA-06512: at line 13
```

- To handle this error oracle, provide “CURSOR_ALREADY_OPEN” exception.

```
SQL> DECLARE
  2     CURSOR C1 IS SELECT EMPNO, ENAME, SAL FROM EMP WHERE
DEPTNO=10;
  3     V_EMPNO NUMBER(10);
  4     V_ENAME VARCHAR2(10);
  5     V_SAL NUMBER(10);
  6  BEGIN
  7     OPEN C1;
  8     LOOP
  9     FETCH C1 INTO V_EMPNO, V_ENAME, V_SAL;
 10     EXIT WHEN C1%NOTFOUND;
 11         DBMS_OUTPUT.PUT_LINE(V_EMPNO||', '||V_ENAME||',
'||V_SAL);
 12     END LOOP;
 13     OPEN C1;
 14     EXCEPTION
 15         WHEN CURSOR_ALREADY_OPEN THEN
 16         DBMS_OUTPUT.PUT_LINE('We must clsoe the cursor before
reopen');
 17  END;
 18  /
7782, CLARK, 2450
7839, KING, 5000
7934, MILLER, 1300
We must clsoe the cursor before reopen

PL/SQL procedure successfully completed.
```

SQLCODE & SQLERRM:

- PL/SQL provides following built-in/ predefine properties which are used to handle an exception which was raised in PL/SQL block.
- SQLCODE returns error code/ exception number.
- SQLERRM returns error message.
- These two properties are used along with “OTHERS” exception name.

```
SQL> DECLARE
  2     X NUMBER(10);
  3     Y NUMBER(10);
  4     Z NUMBER(10);
  5  BEGIN
  6     X:=&X;
  7     Y:=&Y;
  8     Z:=X/Y;
  9     DBMS_OUTPUT.PUT_LINE(Z);
```

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

PL/SQL procedure successfully completed.

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

PL/SQL procedure successfully completed.
```

User define Exception:

- When we create our own exception name and raise explicitly whenever is required. These types of exceptions are called as user define exceptions.
- Generally, if we want to return message as per client business rules then we must use user define exceptions.
- To create a user, define exception name then we follow the following three steps are,

Step1: Syntax to declare user define exception name:

Declare
<user define exception name> EXCEPTION;

Step2: Syntax to raise user define exception:

Method 1: RAISE statement

- It can raise an exception & handling an exception.
  RAISE < user define exception name>;

Method 2: RAISE_APPLICATION_ERROR (NUMBER, MESSAGE)

- It can raise an exception but not handling an exception.
- It is a pre-define method which is used to display a user define exception information in form of oracle format.
- This method is having two arguments are number and message.
  - Number - number should be -20001 to -20999

- Message - user define exception message.

Step3: Syntax to handling user define exception:

```
EXCEPTION
WHEN < user define exception name> THEN
        <statements>;
```

RAISE statement:

```
SQL> DECLARE
  2     X INT;
  3     Y INT;
  4     Z INT;
  5     EX EXCEPTION;
  6  BEGIN
  7     X:=&X;
  8     Y:=&Y;
  9     IF Y=0 THEN
 10         RAISE EX;
 11     ELSE
 12         Z:=X/Y;
 13         DBMS_OUTPUT.PUT_LINE(Z);
 14     END IF;
 15     EXCEPTION
 16         WHEN EX THEN
 17             DBMS_OUTPUT.PUT_LINE('Second number not be zero');
 18  END;
 19  /
Enter value for x: 10
Enter value for y: 2
5

PL/SQL procedure successfully completed.

SQL> /
Enter value for x: 10
Enter value for y: 0
Second number not be zero

PL/SQL procedure successfully completed.
```

RAISE_APPLICATION_ERROR (NUMBER, MESSAGE):

```
SQL> DECLARE

  2     X INT;
  3     Y INT;
  4     Z INT;
  5     EX EXCEPTION;
```

```
  6  BEGIN
  7     X:=&X;
  8     Y:=&Y;
  9     IF Y=0 THEN
 10         RAISE EX;
 11     ELSE
 12         Z:=X/Y;
 13         DBMS_OUTPUT.PUT_LINE(Z);
 14     END IF;
 15     EXCEPTION
 16         WHEN EX THEN
 17             RAISE_APPLICATION_ERROR(-20457, 'Second number not
be zero');
 18  END;
 19  /
Enter value for x: 10
Enter value for y: 0
DECLARE
*
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.

Syntax:

PRAGMA EXCEPTION_INIT (<user define exception name>, error number);

```
SQL> DECLARE
  2     X EXCEPTION;
  3     PRAGMA EXCEPTION_INIT(X, -2291);
  4  BEGIN
  5     INSERT INTO EMP(EMPNO, ENAME, DEPTNO) VALUES (1122, 'SAI',
50);
  6     EXCEPTION
  7         WHEN X THEN
  8             DBMS_OUTPUT.PUT_LINE('Not allowed into emp table
because parent key is not found');
  9  END;
```

```
 10  /
Not allowed into emp table because parent key is not found

PL/SQL procedure successfully completed.
```

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.

```
SQL> DECLARE
  2     X VARCHAR2(3):='PQRS';
  3  BEGIN
  4     DBMS_OUTPUT.PUT_LINE(X);
  5     EXCEPTION
  6         WHEN VALUE_ERROR THEN
  7         DBMS_OUTPUT.PUT_LINE('Invalid string length');
  8  END;
  9  /
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character string buffer
too small
ORA-06512: at line 2
```

- 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.
- To implement exception propagation to handle exception which was raised in declaration block then we nested PL/SQL block.

```
SQL> BEGIN
  2  DECLARE
  3     X VARCHAR2(3):='PQRS';
  4  BEGIN
  5     DBMS_OUTPUT.PUT_LINE(X);
  6     EXCEPTION
  7         WHEN VALUE_ERROR THEN
  8         DBMS_OUTPUT.PUT_LINE('Invalid string length');
  9  END;
 10     EXCEPTION
 11         WHEN VALUE_ERROR THEN
 12         DBMS_OUTPUT.PUT_LINE('String length is greater than size
```

```
of variable x');
 13  END;
 14  /
String length is greater than size of variable x

PL/SQL procedure successfully completed.
```

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 whereas when exceptions are occurred in declaration block those exceptions are handled in outer block only. This mechanism is called as Exception propagation.

Syntax to see the error while creating block if any issue:
SQL> SHOW ERROR;

```
SQL> SHOW ERROR;
No errors.
```

Sub Blocks

- A sub block is a named block of code that is directly saved on the DB server permanently and it can be executed when and where it is required.
- These are also called pre-compiled blocks.
- We have four types of sub blocks in oracle.
    - Stored procedures
    - Stored functions
    - Packages
    - 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.
- But when we use “OUT” parameters then stored procedures must be return a value otherwise no return value.
- Whenever we want to execute a SQL query from an application the SQL query will be first parsed (i.e. complied) 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-complied block of code without parsing the statements gets executed whenever the procedures are called which can increase the performance of an application.

Syntax to create a Stored procedure:

```
CREATE OR REPLACE PROCEDURE <procedure name> (
        parameter name [mode type] datatype,
        ….
)
IS / AS
<variable declaration>;
BEGIN
        <exec statements>;
        <exception statements>
        <handling statements>
END / END <procedure name>;
/
```

Advantages of Stored Procedure:

- As there is no unnecessary compilation of queries, this will reduce burden on database.
- Application performance will be improved.
- User will get quick response
- Code reusability & security.

To execute/ call the procedure:

Syntax1:

```
SQL> EXECUTE / EXEC <procedure name> (values);
```

Syntax2: (anonymous block)

```
BEGIN
        <procedure name>(values);
END;
```

Examples on procedure without parameters:

```
SQL> CREATE OR REPLACE PROCEDURE MY_PROC
  2  IS
  3  BEGIN
  4     DBMS_OUTPUT.PUT_LINE('Welcome to procedures');
  5  END MY_PROC;
```

```
  7  /

Procedure created.
SQL> EXEC MY_PROC;
Welcome to procedures

PL/SQL procedure successfully completed.
```

Ex: WASP to display sum of two numbers.

```
SQL> CREATE OR REPLACE PROCEDURE ADD_PROC
  2  IS
  3  A NUMBER:=10;
  4  B NUMBER:=10;
  5  BEGIN
  6     DBMS_OUTPUT.PUT_LINE('Sum of two number : '||(A+B));
  7  END;
  8  /

Procedure created.

SQL> EXEC ADD_PROC;
Sum of two number : 20

PL/SQL procedure successfully completed.
```

Examples on procedures with parameters:

```
SQL> CREATE OR REPLACE PROCEDURE ADD_PROC(A NUMBER, B NUMBER)
  2  IS
  3  BEGIN
  4     DBMS_OUTPUT.PUT_LINE('Sum of two number : '||(A+B));
  5  END;
  6  /

Procedure created.

SQL> EXEC ADD_PROC(10, 30);
Sum of two number : 40

PL/SQL procedure successfully completed.
```

Types parameter modes:

- There are three types of parameters modes.
    - IN:
        - It accepts/ store input values from user into stored procedure.
        - These are default parameters.

- OUT:
  - It returns output values from stored procedure to user.
- IN OUT:
  - Both accepting and also return.

IN parameters:

```
SQL> CREATE OR REPLACE PROCEDURE ADD_PROC(A IN NUMBER, B IN NUMBER)
  2  IS
  3  BEGIN
  4     DBMS_OUTPUT.PUT_LINE('Sum of two number : '||(A+B));
  5  END;
  6  /

Procedure created.
```

Ex: WASP to take EMPNO and display that employee NAME, SAL from EMP table.

```
SQL> CREATE OR REPLACE PROCEDURE EMP_PROC1(P_EMPNO IN NUMBER)
  2  IS
  3  V_ENAME VARCHAR2(10);
  4  V_SAL NUMBER(10);
  5  BEGIN
  6     SELECT ENAME, SAL INTO V_ENAME, V_SAL FROM EMP WHERE
EMPNO=P_EMPNO;
  7     DBMS_OUTPUT.PUT_LINE(V_ENAME||', '||V_SAL);
  8  END;
  9  /

Procedure created.

SQL> EXECUTE EMP_PROC1(7788);
SCOTT, 3000

PL/SQL procedure successfully completed.
```

OUT parameters:

```
SQL> CREATE OR REPLACE PROCEDURE SP1(X IN NUMBER, Y OUT NUMBER)
  2  IS
  3  BEGIN
  4     Y:=X*X*X;
  5  END;
  6  /

Procedure created.

SQL> EXECUTE SP1(5);
```

```
BEGIN SP1(5); END;

      *
ERROR at line 1:
ORA-06550: line 1, column 7:
PLS-00306: wrong number or types of arguments in call to 'SP1'
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored
```

Note: To overcome the above problem then we follow the following 3 steps,

Step1: Syntax to Declare referenced/ bind variable for "OUT" parameters in stored procedure

SQL> VAR/ VARIABLE <bind/ ref. variable name> <dt>[size];

Step2: Syntax to add a referenced/ bind variable to a stored procedure

SQL> EXECUTE <parameter name> (value1, value2, ..., :<ref variable name>....);

Step3: Syntax to print referenced variables

SQL> PRINT <ref. variable name>;

```
SQL> VAR POWER NUMBER;
SQL> EXECUTE SP1(5, :POWER);

PL/SQL procedure successfully completed.

SQL> PRINT POWER;

     POWER
----------
       125
```

Ex: WASP to input EMPNO as an "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 SP2(P_EMPNO IN NUMBER, PF OUT
NUMBER, PT OUT NUMBE
R)
  2  IS
  3     V_SAL NUMBER(10);
  4  BEGIN
  5     SELECT SAL INTO V_SAL FROM EMP WHERE EMPNO=P_EMPNO;
  6     PF:=V_SAL*0.1;
  7     PT:=V_SAL*0.2;
```

```
  8  END;
  9  /

Procedure created.

SQL> VAR RPF NUMBER;
SQL> VAR RPT NUMBER;
SQL> EXECUTE SP2(7788, :RPF, :RPT);

PL/SQL procedure successfully completed.

SQL> PRINT RPF RPT;

       RPF
----------
       300

       RPT
----------
       600
```

IN OUT parameters:

```
SQL> CREATE OR REPLACE PROCEDURE SP4(X IN OUT NUMBER)
  2  AS
  3  BEGIN
  4     X:=X*X;
  5  END;
  6  /

Procedure created.

SQL> EXECUTE SP4(5);
BEGIN SP4(5); END;

          *
ERROR at line 1:
ORA-06550: line 1, column 11:
PLS-00363: expression '5' cannot be used as an assignment target
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored
```

Note: To overcome the above problem then we follow the following 4 steps,

Step1: Syntax to declare referenced variable for "out" parameters in stored procedure

SQL> VAR/ VARIABLE <ref. variable name> <dt>[size];

Step2: Syntax to assign a value to referenced variable:

SQL> EXECUTE <ref. variable name> := <value>;

Step3: Syntax to add a referenced variable to a stored procedure

SQL> EXECUTE <procedure name> (:<ref. variable name>.....);

Step4: Syntax to print referenced variables

SQL> PRINT <ref. variable name>;

```
SQL> VAR RX NUMBER;
SQL> EXECUTE :RX:=10;

PL/SQL procedure successfully completed.

SQL> EXECUTE SP4(:RX);

PL/SQL procedure successfully completed.

SQL> PRINT RX;

        RX
----------
       100
```

Note: To view all stored procedures in oracle DB the we have to use "USER_OBJECTS" data dictionary.

```
SQL> DESC USER_OBJECTS;

SQL> SELECT OBJECT_NAME FROM USER_OBJECTS WHERE
OBJECT_TYPE='PROCEDURE';

OBJECT_NAME
-----------------------------------
MY_PROC
ADD_PROC
EMP_PROC1
SP1
```

Note: To view procedure bodies/ source code of a particular sub block object then we have to user "USER_SOURCE" data dictionary.

```
SQL> DESC USER_OBJECTS;

SQL> SELECT TEXT FROM USER_SOURCE WHERE NAME='EMP_PROC1';
```

```
TEXT
--------------------------------------------------
PROCEDURE EMP_PROC1(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;

8 rows selected.
```

Syntax to drop a procedure:

SQL> DROP PROCEDURE <procedure name>;

```
SQL> DROP PROCEDURE MY_PROC;

Procedure dropped.
```

Stored Functions

- A function is block of code to perform some tasks and must return a value. These functions are created by user explicitly. So that we can also called as “User defined function”

  Syntax:

  CREATE OR REPLACE FUNCTION <function name>(<parameter name> datatype, ……)
  RETURN <return variables datatype>
  IS/ AS
  BEGIN
  <exec-statements>;
  RETURN (value/ variable name);
  END/ END <function name>;
  /

How to call a stored function:

SQL> SELECT <function name>(values) from dual;

Ex: WASF to accept EMPNO and return ENAME from EMP table?

```
SQL> CREATE OR REPLACE FUNCTION SF1(P_EMPNO NUMBER)
  2     RETURN VARCHAR2
  3  AS
  4     V_ENAME VARCHAR2(10);
  5  BEGIN
  6     SELECT ENAME INTO V_ENAME FROM EMP WHERE EMPNO=P_EMPNO;
  7     RETURN V_ENAME;
  8  END;
  9  /

Function created.

SQL> SELECT SF1(7566) FROM DUAL;

SF1(7566)
----------------------------------------
JONES
```

Ex: WASF to take input as department name and return sum of salary of the department.

```
SQL> CREATE OR REPLACE FUNCTION SF2(P_DNAME VARCHAR2)
  2     RETURN NUMBER
  3  AS
  4     V_TOTALSAL NUMBER(10);
  5  BEGIN
  6     SELECT SUM(SAL) INTO V_TOTALSAL FROM EMP E INNER JOIN DEPT D
  7     ON E.DEPTNO = D.DEPTNO WHERE D.DNAME = P_DNAME;
  8     RETURN V_TOTALSAL;
  9  END;
 10  /

Function created.

SQL> SELECT SF2('ACCOUNTING') FROM DUAL;

SF2('ACCOUNTING')
-----------------
             8750
```

Ex: WASF to return number of employees in between given dates?

```
SQL> CREATE OR REPLACE FUNCTION SF3(SD DATE, ED DATE)
  2     RETURN NUMBER
  3  AS
  4     V_COUNT NUMBER(10);
  5  BEGIN
  6     SELECT COUNT(*) INTO V_COUNT FROM EMP WHERE HIREDATE BETWEEN
```

```
SD AND ED;
  7     RETURN V_COUNT;
  8  END;
  9  /

Function created.

SQL> SELECT SF3('01-JAN-81', '31-DEC-81') FROM DUAL;

SF3('01-JAN-81','31-DEC-81')
----------------------------
                          10
```

Ex: WASF to take input EMPNO and return that employee gross salary as per given conditions are

- HRA - 10%
- DA - 20%
- PF - 10%

```
SQL> CREATE OR REPLACE FUNCTION SF4(P_EMPNO NUMBER)
  2     RETURN NUMBER
  3  AS
  4     V_BSAL NUMBER(10);
  5     V_HRA NUMBER(10);
  6     V_DA NUMBER(10);
  7     V_PF NUMBER(10);
  8     V_GROSS NUMBER(10);
  9  BEGIN
 10     SELECT SAL INTO V_BSAL FROM EMP WHERE EMPNO=P_EMPNO;
 11     V_HRA:=V_BSAL*0.1;
 12     V_DA:=V_BSAL*0.2;
 13     V_PF:=V_BSAL*0.1;
 14     V_GROSS:=V_BSAL+V_HRA+V_DA+V_PF;
 15     RETURN V_GROSS;
 16  END;
 17  /

Function created.

SQL> SELECT SF4(7788) FROM DUAL;

 SF4(7788)
----------
      4200
```

Ex: WASF to find simple interest.

```
SQL> CREATE OR REPLACE FUNCTION SI(P NUMBER, T NUMBER, R NUMBER)
  2     RETURN NUMBER
  3  IS
  4     SIMPLE_INT NUMBER;
  5  BEGIN
  6     SIMPLE_INT:=(P*T*R)/100;
  7     RETURN SIMPLE_INT;
  8  END SI;
  9  /

Function created.

SQL> SELECT SI(1000, 2, 10) FROM DUAL;

SI(1000,2,10)
-------------
          200
```

Ex: WASF to find experience of given employee.

```
SQL> CREATE OR REPLACE FUNCTION EMP_EXP(TEMPNO EMP.EMPNO%TYPE)
  2     RETURN VARCHAR2
  3  IS
  4     TDATE EMP.HIREDATE%TYPE;
  5     TEXP NUMBER;
  6  BEGIN
  7     SELECT HIREDATE INTO TDATE FROM EMP WHERE EMPNO=TEMPNO;
  8     TEXP:=ROUND((SYSDATE-TDATE)/365);
  9     RETURN (TEMPNO||' employee experience is '||texp||'
years.');
 10     EXCEPTION
 11         WHEN NO_DATA_FOUND THEN
 12         RETURN ('Given employee round not found.');
 13  END;
 14  /

Function created.

SQL>

EMP_EXP(7788)
--------------------------------------------------
7788 employee experience is 41 years.
```

Ex: WASF to calculate employee experience.

```
SQL> CREATE OR REPLACE FUNCTION EMP_EXPE(TEMPNO EMP.EMPNO%TYPE)
  2     RETURN NUMBER
```

```
  3  IS
  4     TEXP NUMBER;
  5  BEGIN
  6     SELECT ROUND((SYSDATE-HIREDATE)/365) INTO TEXP FROM EMP
WHERE EMPNO=TEMPNO;
  7     RETURN TEXP;
  8  END;
  9  /

Function created.

SQL> SELECT EMP_EXPE(7788) FROM DUAL;

EMP_EXPE(7788)
--------------
            41
```

Note: To view all stored functions in oracle DB the we have to use "USER_OBJECTS" data dictionary.

```
SQL> DESC USER_OBJECTS;

SQL> SELECT OBJECT_NAME FROM USER_OBJECTS WHERE
OBJECT_TYPE='FUNCTION';

OBJECT_NAME
------------------------------------------------
SF1
SF2
SF3
```

Note: To view function bodies/ source code of a particular sub block object then we have to user "USER_SOURCE" data dictionary.

```
SQL> DESC USER_SOURCE;

SQL> SELECT TEXT FROM USER_SOURCE WHERE NAME='EMP_EXP';

TEXT
------------------------------------------------------------------
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 round not found.');
END;

13 rows selected.
```

Syntax to dropping functions:

SQL> DROP FUNCTION <function name>;

```
SQL> DROP FUNCTION EMP_EXPE;

Function dropped.
```

Packages

- It is a collection of variables, cursors, procedures & functions are stored in one location.
- To bind stored procedure & stored functions within the same unit of memory.
- 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 support function overloading, encapsulation & databinding
- To create a package, we need two blocks
    - Package specification block
    - Package implementation block (body)

Package specification block:

- It holds the declaration of variables, cursors, stored procedure and stored functions.

Syntax:

CREATE OR REPLACE PACKAGE <package name>
IS / AS
        <declare variables, cursors, sub blocks>;
END;
/

Package implementation block (body):

- It holds the implementation logical code of procedures & functions which we declared in package specification.

  Syntax:

  CREATE OR REPLACE PACKAGE BODY <package name>
  IS / AS
  <implementing sub blocks>;
  END;
  /

Syntax to call a stored procedure from a package:

SQL> EXECUTE <package name>.<procedure name>(values);

Syntax to call a stored function from a package:

SQL> SELECT <package name>.<function name>(values) FROM DUAL;

Ex: WAP to encapsulate/ bind multiple procedures in a single unit of memory.

```
SQL> CREATE OR REPLACE PACKAGE PK1
  2  IS
  3     PROCEDURE P1;
  4     PROCEDURE P2;
  5  END;
  6  /

Package created.

SQL> CREATE OR REPLACE PACKAGE BODY PK1
  2  IS
  3     PROCEDURE P1
  4  AS
  5  BEGIN
  6     DBMS_OUTPUT.PUT_LINE('My name is procedure');
  7  END P1;
  8     PROCEDURE P2
  9  AS
 10  BEGIN
 11     DBMS_OUTPUT.PUT_LINE('My name is second procedure');
 12  END P2;
 13  END;
 14  /

Package body created.
```

```
SQL> EXECUTE PK1.P1;
My name is procedure

PL/SQL procedure successfully completed.

SQL> EXECUTE PK1.P2;
My name is second procedure

PL/SQL procedure successfully completed.
```

Ex: WAP to bind variable, procedure & function.

```
SQL> CREATE OR REPLACE PACKAGE PK2
  2  IS
  3     X NUMBER(10):=2000;
  4     PROCEDURE SP1;
  5     FUNCTION SF1(A NUMBER) RETURN NUMBER;
  6  END;
  7  /

Package created.

SQL> CREATE OR REPLACE PACKAGE BODY PK2
  2  IS
  3  PROCEDURE SP1
  4  AS
  5     A NUMBER(10);
  6  BEGIN
  7     A:=X/2;
  8     DBMS_OUTPUT.PUT_LINE(A);
  9  END SP1;
 10  FUNCTION SF1(A NUMBER)
 11     RETURN NUMBER
 12  AS
 13  BEGIN
 14     RETURN A*X;
 15  END SF1;
 16  END;
 17  /

Package body created.

SQL> EXECUTE PK2.SP1;
1000

PL/SQL procedure successfully completed.

SQL> SELECT PK2.SF1(10) FROM DUAL;
```

```
PK2.SF1(10)
-----------
      20000
```

```
SQL> CREATE OR REPLACE PACKAGE MY_PACK
  2  IS
  3     RESULT VARCHAR2(50);
  4     PROCEDURE EMP_EXP(TEMPNO EMP.EMPNO%TYPE);
  5     FUNCTION EMP_NETSAL(TEMPNO EMP.EMPNO%TYPE) RETURN VARCHAR2;
  6  END MY_PACK;
  7  /

SQL> CREATE OR REPLACE PACKAGE BODY MY_PACK
  2  IS
  3  PROCEDURE EMP_EXP(TEMPNO EMP.EMPNO%TYPE)
  4  IS
  5     TDATE EMP.HIREDATE%TYPE;
  6     TEXP NUMBER;
  7  BEGIN
  8     SELECT HIREDATE INTO TDATE FROM EMP WHERE EMPNO=TEMPNO;
  9     TEXP:=ROUND((SYSDATE-TDATE)/365);
 10     DBMS_OUTPUT.PUT_LINE(TEMPNO||' employee experience is
'||TEXP||' years.');
 11  END EMP_EXP;
 12  FUNCTION EMP_NETSAL(TEMPNO EMP.EMPNO%TYPE)
 13     RETURN VARCHAR2
 14  IS
 15     TSAL EMP.SAL%TYPE;
 16     TCOMM EMP.COMM%TYPE;
 17  BEGIN
 18     SELECT SAL+NVL(COMM,0) INTO RESULT FROM EMP WHERE
EMPNO=TEMPNO;
 19     RETURN (TEMPNO||' employee net salary Rs. '||RESULT);
 20  END EMP_NETSAL;
 21  END;
 22  /

Package body created.

SQL> EXEC MY_PACK.EMP_EXP(7788);
7788 employee experience is 41 years.

PL/SQL procedure successfully completed.

SQL> SELECT MY_PACK.EMP_NETSAL(7788) FROM DUAL;

MY_PACK.EMP_NETSAL(7788)
------------------------------------------------------------
7788 employee net salary Rs. 3000
```

Function overloading using package:

```
SQL> CREATE OR REPLACE PACKAGE PK3
  2  IS
  3     FUNCTION SF1(X NUMBER, Y NUMBER) RETURN NUMBER;
  4     FUNCTION SF1(X NUMBER, Y NUMBER, Z NUMBER) RETURN NUMBER;
  5  END;
  6  /

Package created.

SQL> CREATE OR REPLACE PACKAGE BODY PK3
  2  IS
  3  FUNCTION SF1(X NUMBER, Y NUMBER)
  4     RETURN NUMBER
  5  AS
  6  BEGIN
  7     RETURN (X*Y);
  8  END SF1;
  9  FUNCTION SF1(X NUMBER, Y NUMBER, Z NUMBER)
 10     RETURN NUMBER
 11  AS
 12  BEGIN
 13     RETURN (X*Y*Z);
 14  END SF1;
 15  END;
 16  /

Package body created.

SQL> SELECT PK3.SF1(10, 20) FROM DUAL;

PK3.SF1(10,20)
--------------
           200

SQL> SELECT PK3.SF1(10, 20, 30) FROM DUAL;

PK3.SF1(10,20,30)
-----------------
             6000
```

```
SQL> CREATE OR REPLACE PACKAGE PK4
  2  IS
  3     FUNCTION ADDVAL(A NUMBER, B NUMBER) RETURN NUMBER;
  4     FUNCTION ADDVAL(A NUMBER, B NUMBER, C NUMBER) RETURN NUMBER;
  5     FUNCTION ADDVAL(STR1 VARCHAR2, STR2 VARCHAR2) RETURN
VARCHAR2;
  6     FUNCTION ADDVAL(STR1 VARCHAR2, STR2 VARCHAR2, STR3 VARCHAR2)
```

```
RETURN VARCHAR
2;
  7  END PK4;
  8  /

Package created.

SQL> CREATE OR REPLACE PACKAGE BODY PK4
  2  IS
  3  FUNCTION ADDVAL(A NUMBER, B NUMBER) RETURN NUMBER
  4  IS
  5  BEGIN
  6     RETURN(A+B);
  7  END ADDVAL;
  8  FUNCTION ADDVAL(A NUMBER, B NUMBER, C NUMBER) RETURN NUMBER
  9  IS
 10  BEGIN
 11     RETURN(A+B+C);
 12  END ADDVAL;
 13  FUNCTION ADDVAL(STR1 VARCHAR2, STR2 VARCHAR2) RETURN VARCHAR2
 14  IS
 15  BEGIN
 16     RETURN(STR1||' '||STR2);
 17  END ADDVAL;
 18  FUNCTION ADDVAL(STR1 VARCHAR2, STR2 VARCHAR2, STR3 VARCHAR2)
RETURN VARCHAR2
 19  IS
 20  BEGIN
 21     RETURN(STR1||' '||STR2||' '||STR3);
 22  END ADDVAL;
 23  END;
 24  /

Package body created.

SQL> SELECT PK4.ADDVAL(10, 20) FROM DUAL;

PK4.ADDVAL(10,20)
-----------------
               30

SQL> SELECT PK4.ADDVAL(10, 20, 50) FROM DUAL;

PK4.ADDVAL(10,20,50)
--------------------
                  80
```

```
SQL> SELECT PK4.ADDVAL('GOOD', 'MORNING') FROM DUAL;

PK4.ADDVAL('GOOD','MORNING')
------------------------------------------------------------
GOOD MORNING

SQL> SELECT PK4.ADDVAL('HII ALL,', 'GOOD', 'MORNING') FROM DUAL;

PK4.ADDVAL('HIIALL,','GOOD','MORNING')
------------------------------------------------------------
HII ALL, GOOD MORNING
```

Note: To view all packages in oracle DB the we have to use “USER_OBJECTS” data dictionary.

```
SQL> DESC USER_OBJECTS;

SQL> SELECT OBJECT_NAME FROM USER_OBJECTS WHERE
OBJECT_TYPE='PACKAGE';

OBJECT_NAME
------------------------------------------------------------
PK1
PK2
```

Note: To view package bodies/ source code of a particular sub block object then we have to user “USER_SOURCE” data dictionary.

```
SQL> DESC USER_SOURCE;

SQL> SELECT TEXT FROM USER_SOURCE WHERE NAME='PK1';

TEXT
------------------------------------------------------------
PACKAGE PK1
IS
        PROCEDURE P1;
        PROCEDURE P2;
END;
PACKAGE BODY PK1
IS
        PROCEDURE P1
AS
BEGIN
        DBMS_OUTPUT.PUT_LINE('My name is procedure');
```

```
END P1;
        PROCEDURE P2
AS
BEGIN
        DBMS_OUTPUT.PUT_LINE('My name is second procedure');
END P2;
END;

18 rows selected.
```

Syntax to dropping package body:

SQL> DROP PACKAGE BODY <package name>;

```
SQL> DROP PACKAGE BODY PK1;

Package body dropped.
```

Syntax to dropping package body:

SQL> DROP PACKAGE <package name>;

```
SQL> DROP PACKAGE PK1;

Package dropped.
```

Note: The entire package (specification & implantation block) will be dropped.

Triggers

- A set of PL/SQL statements stored permanently in database and "automatically" activated/ executed whenever an event raising statement (DML/ DDL) is performed.
- It is also a named blocked which is similar to stored procedure but executed implicitly (by system automatically).
- 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_TRIGGERS" system table.
- There are 2 types of triggers
    - DML triggers
    - DDL trigger/ DB triggers

Purpose of triggers:

- To invoking user defined message (or) alerts at event raise.
- To control/ restrict DML/ DDL operations.

- To implementing business logical conditions.
- To perform validations.
- For auditing.

DML Triggers:

- When we created a trigger objected based on DML commands (INSERT/ UPDATE/ DELETE) then those triggers called as DML triggers.
- These triggers are executed by system automatically when we modify data in a table by using DML commands.

Syntax:

```
CREATE OR REPLACE TRIGGER <trigger name>
        BEFORE/ AFTER <INSERT OR UPDATE OR DELETE>
                                [OF <columns>] ON <table name>
        [FOR EACH ROW]
        [WHEN <condition>]
[DECLARE <variable declaration>;]
BEGIN
        <exec statements>;
        [EXCEPTION]
        <exec statements>;]
END;
/
```

Trigger event:

- Indicates when to activate the trigger.
  - Before trigger:
    - First trigger body is executed
    - Later DML command executed
  - After trigger:
    - First DML command executed
    - Later trigger body executed

Note: These two events will provide same result finally.

Trigger levels:

- Triggers are can create at two levels.
  - Row level trigger
  - Statement level trigger

Row level trigger:

- In this level, trigger body (logic) is executing for each row wise for a DML operation.

First create the below table with data

```
SQL> SELECT * FROM EMPLOYEE;

       EID ENAME              SAL
---------- ---------- ----------
         1 SMITH           15000
         2 JONES           25000
         3 WARD            15000
         4 ADAMS           32000
```

```
SQL> CREATE OR REPLACE TRIGGER TR1
  2     AFTER UPDATE ON EMPLOYEE
  3     FOR EACH ROW
  4  BEGIN
  5     DBMS_OUTPUT.PUT_LINE('HELLO');
  6  END;
  7  /

Trigger created.

SQL> UPDATE EMPLOYEE SET SAL=10000 WHERE SAL=15000;
HELLO
HELLO

2 rows updated.
```

Statement level trigger:

- In this level, trigger body is executing only one time for a DML operation.

```
SQL> CREATE OR REPLACE TRIGGER TR1
  2     AFTER UPDATE ON EMPLOYEE
  3  BEGIN
  4     DBMS_OUTPUT.PUT_LINE('HELLO');
  5  END;
  6  /

Trigger created.

SQL> UPDATE EMPLOYEE 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,
  - : NEW
    - This bind variable will store new values when we insert.

      Syntax:
      : NEW. <column name>= <value>;

      Ex:
      : NEW.sal = 15000;

  - : 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".

To invoking user defined message (or) alerts at event raise:

INSERT:

```
SQL> CREATE OR REPLACE TRIGGER TINSERT
  2     AFTER INSERT ON EMPLOYEE
  3  BEGIN
  4     DBMS_OUTPUT.PUT_LINE('Someone inserted data into your
table');
  5  END;
  6  /

Trigger created.

SQL> INSERT INTO EMPLOYEE VALUES (5, 'SCOTT', 36000);
Someone inserted data into your table

1 row created.
```

UPDATE:

```
SQL> CREATE OR REPLACE TRIGGER TUPDATE
  2     AFTER UPDATE ON EMPLOYEE
```

```
  3  BEGIN
  4     DBMS_OUTPUT.PUT_LINE('Someone updated data into your
table');
  5  END;
  6  /

Trigger created.

SQL> UPDATE EMPLOYEE SET SAL=22000 WHERE EID=5;
Someone updated data into your table

1 row updated.
```

DELETE:

```
SQL> CREATE OR REPLACE TRIGGER TDELETE
  2     AFTER DELETE ON EMPLOYEE
  3  BEGIN
  4     DBMS_OUTPUT.PUT_LINE('Someone deleted data into your
table');
  5  END;
  6  /

Trigger created.

SQL> DELETE FROM EMPLOYEE WHERE EID=5;
Someone deleted data into your table

1 row deleted.
```

All DML operations:

```
SQL> CREATE OR REPLACE TRIGGER TDML
  2     AFTER INSERT OR UPDATE OR DELETE ON EMPLOYEE
  3  BEGIN
  4     DBMS_OUTPUT.PUT_LINE('Someone performing DML operations your
table');
  5  END;
  6  /

Trigger created.

SQL> INSERT INTO EMPLOYEE VALUES(5, 'SCOTT', 25000);
Someone performing DML operations your table

1 row created.

SQL> UPDATE EMPLOYEE SET SAL=30000 WHERE EID=5;
Someone performing DML operations your table
```

```
1 row updated.

SQL> DELETE FROM EMPLOYEE WHERE EID=5;
Someone performing DML operations your table

1 row deleted.
```

To control/ restrict DML/ DDL operations:

INSERT:

```
SQL> CREATE OR REPLACE TRIGGER TRIN
  2     AFTER INSERT ON EMPLOYEE
  3  BEGIN
  4     RAISE_APPLICATION_ERROR(-20487, 'Someone inserting data into
your table');
  5  END;
  6  /

Trigger created.

SQL> INSERT INTO EMPLOYEE VALUES(6, 'MILLER', 52000);
INSERT INTO EMPLOYEE VALUES(6, 'MILLER', 52000)
            *
ERROR at line 1:
ORA-20487: Someone inserting data into your table
ORA-06512: at "NIRMALA.TRIN", line 2
ORA-04088: error during execution of trigger 'NIRMALA.TRIN'
```

UPDATE:

```
SQL> CREATE OR REPLACE TRIGGER TRUP
  2     AFTER UPDATE ON EMPLOYEE
  3  BEGIN
  4     RAISE_APPLICATION_ERROR(-20487, 'Someone updating data into
your table');
  5  END;
  6  /

Trigger created.

SQL> UPDATE EMPLOYEE SET SAL=30000 WHERE EID=4;
UPDATE EMPLOYEE SET SAL=30000 WHERE EID=4
       *
ERROR at line 1:
ORA-20487: Someone updating data into your table
ORA-06512: at "NIRMALA.TRUP", line 2
ORA-04088: error during execution of trigger 'NIRMALA.TRUP'
```

DELETE:

```
SQL> CREATE OR REPLACE TRIGGER TRDEL
```

```
  2     AFTER DELETE ON EMPLOYEE
  3  BEGIN
  4     RAISE_APPLICATION_ERROR(-20487, 'Someone deleting data into
your table');
  5  END;
  6  /

Trigger created.

SQL> DELETE FROM EMPLOYEE WHERE EID=4;
DELETE FROM EMPLOYEE WHERE EID=4
            *
ERROR at line 1:
ORA-20487: Someone deleting data into your table
ORA-06512: at "NIRMALA.TRDEL", line 2
ORA-04088: error during execution of trigger 'NIRMALA.TRDEL'
```

All DML operations:

```
SQL> CREATE OR REPLACE TRIGGER TRDML
  2     AFTER INSERT OR UPDATE OR DELETE ON EMPLOYEE
  3  BEGIN
  4     RAISE_APPLICATION_ERROR(-20487, 'Someone performing DML
operations on your table');
  5  END;
  6  /

Trigger created.

SQL> INSERT INTO EMPLOYEE VALUES(6, 'MILLER', 52000);
INSERT INTO EMPLOYEE VALUES(6, 'MILLER', 52000)
            *
ERROR at line 1:
ORA-20487: Someone performing DML operations on your table
ORA-06512: at "NIRMALA.TRDML", line 2
ORA-04088: error during execution of trigger 'NIRMALA.TRDML'

SQL> UPDATE EMPLOYEE SET SAL=30000 WHERE EID=5;
UPDATE EMPLOYEE SET SAL=30000 WHERE EID=5
       *
ERROR at line 1:
ORA-20487: Someone performing DML operations on your table
ORA-06512: at "NIRMALA.TRDML", line 2
ORA-04088: error during execution of trigger 'NIRMALA.TRDML'

SQL> DELETE FROM EMPLOYEE WHERE EID=5;
DELETE FROM EMPLOYEE WHERE EID=5
            *
ERROR at line 1:
ORA-20487: Someone performing DML operations on your table
```

```
ORA-06512: at "NIRMALA.TRDML", line 2
ORA-04088: error during execution of trigger 'NIRMALA.TRDML'
```

To implementing business logical conditions:

Ex: WAT to restricted DML operations on every Tuesday.

```
SQL> CREATE OR REPLACE TRIGGER TRDAY
  2     AFTER INSERT OR UPDATE OR DELETE ON EMPLOYEE
  3  BEGIN
  4     IF TO_CHAR(SYSDATE, 'DY') = 'TUE' THEN
  5         RAISE_APPLICATION_ERROR(-20456, 'You cannot perform DML
operations on Tuesday');
  6     END IF;
  7  END;
  8  /

Trigger created.

SQL> INSERT INTO EMPLOYEE VALUES(6, 'EVE', 50000);
INSERT INTO EMPLOYEE VALUES(6, 'EVE', 50000)
            *
ERROR at line 1:
ORA-20456: You cannot perform DML operations on tuesday
ORA-06512: at "NIRMALA.TRDAY", line 3
ORA-04088: error during execution of trigger 'NIRMALA.TRDAY'
```

Ex: WAT to restricted DML operations on EMPLOYEE table in between 1 am to 9 am.

```
SQL> CREATE OR REPLACE TRIGGER TRTIME
  2     AFTER INSERT OR UPDATE OR DELETE ON EMPLOYEE
  3  BEGIN
  4     IF TO_CHAR(SYSDATE, 'HH24') BETWEEN 1 AND 9 THEN
  5         RAISE_APPLICATION_ERROR(-20456, 'You cannot perform DML
operations between 1 am to 9 am');
  6     END IF;
  7  END;
  8  /

Trigger created.

SQL> UPDATE EMPLOYEE SET SAL=30000 WHERE EID=4;
UPDATE EMPLOYEE SET SAL=30000 WHERE EID=4
       *
ERROR at line 1:
ORA-20456: You cannot perform DML operations between 1 am to 9 am
ORA-06512: at "NIRMALA.TRTIME", line 3
```

```
ORA-04088: error during execution of trigger 'NIRMALA.TRTIME'
```

To validating data:

Ex: WAT to validate insert operation on EMPLOYEE if new salary is less than 8000?

```
SQL> CREATE OR REPLACE TRIGGER TRSAL
  2     BEFORE INSERT ON EMPLOYEE
  3     FOR EACH ROW
  4  BEGIN
  5     IF :NEW.SAL<8000 THEN
  6         RAISE_APPLICATION_ERROR(-20348, 'New SAL should not be
less than to 8000');
  7     END IF;
  8  END;
  9  /

Trigger created.

SQL> INSERT INTO EMPLOYEE VALUES (5, 'ROBBOT', 7500);
INSERT INTO EMPLOYEE VALUES (5, 'ROBBOT', 7500)
            *
ERROR at line 1:
ORA-20348: New SAL should not be less than to 8000
ORA-06512: at "NIRMALA.TRSAL", line 3
ORA-04088: error during execution of trigger 'NIRMALA.TRSAL'
```

Ex: WAT to validate data and inserted as upper case.

```
SQL> CREATE OR REPLACE TRIGGER DEPT_TRI
  2     BEFORE INSERT ON DEPT
  3     FOR EACH ROW
  4  BEGIN
  5     :NEW.DNAME:=UPPER(:NEW.DNAME);
  6     :NEW.LOC:=UPPER(:NEW.LOC);
  7  END;
  8  /

Trigger created.

SQL> INSERT INTO DEPT VALUES (50, 'economics', 'hyd');

1 row created.

SQL> SELECT * FROM DEPT;

    DEPTNO DNAME          LOC
```

```
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
        50 ECONOMICS      HYD
```

Ex: WAT to validate update operation on EMPLOYEE table if new salary is less than to old salary.

```
SQL> CREATE OR REPLACE TRIGGER TRSAL2
  2     BEFORE UPDATE ON EMPLOYEE
  3     FOR EACH ROW
  4  BEGIN
  5     IF :NEW.SAL<:OLD.SAL THEN
  6         RAISE_APPLICATION_ERROR(-20748, 'New SAL should not less
than to old SAL');
  7     END IF;
  8  END;
  9  /

SQL> UPDATE EMPLOYEE SET SAL=3000 WHERE EID=4;
UPDATE EMPLOYEE SET SAL=3000 WHERE EID=4
       *
ERROR at line 1:
ORA-20748: New SAL should not less than to old SAL
ORA-06512: at "NIRMALA.TRSAL2", line 3
ORA-04088: error during execution of trigger 'NIRMALA.TRSAL2'
```

Ex: WAT to validate delete operation on EMPLOYEE table if we try to delete the employee "smith" details.

```
SQL> CREATE OR REPLACE TRIGGER TRDEL
  2     BEFORE DELETE ON EMPLOYEE
  3     FOR EACH ROW
  4  BEGIN
  5     IF :OLD.ENAME='SMITH' THEN
  6         RAISE_APPLICATION_ERROR(-20648, 'We cannot delete SMITH
employee details');
  7     END IF;
  8  END;
  9  /

Trigger created.

SQL> DELETE FROM EMPLOYEE WHERE ENAME='SMITH';
DELETE FROM EMPLOYEE WHERE ENAME='SMITH'
```

```
            *
ERROR at line 1:
ORA-20648: We cannot delete SMITH employee details
ORA-06512: at "NIRMALA.TRDEL", line 3
ORA-04088: error during execution of trigger 'NIRMALA.TRDEL'
```

For auditing:

- When we manipulate data in a table those transactional values are stored in another table is called as auditing table.

```
SQL> CREATE TABLE EMP1(EID INT, ENAME VARCHAR2(10), SAL NUMBER(10));

Table created.

SQL> CREATE TABLE EMP1_AUDIT(EID INT, AUDIT_INFOR VARCHAR2(100));

Table created.

SQL> SELECT * FROM EMP1;

no rows selected

SQL> SELECT * FROM EMP1_AUDIT;

no rows selected

SQL> CREATE OR REPLACE TRIGGER TRA_AUDIT
  2     AFTER INSERT ON EMP1
  3     FOR EACH ROW
  4  BEGIN
  5     INSERT INTO EMP1_AUDIT VALUES(:NEW.EID, 'Someone inserted a
new row into EMP1 table on'||' '||TO_CHAR(SYSDATE, 'DD-MON-YYYY
HH:MI:SS AM'));
  6  END;
  7  /

Trigger created.

SQL> INSERT INTO EMP1 VALUES (1001, 'SMITH', 80000);

1 row created.

SQL> SELECT * FROM EMP1;

       EID ENAME             SAL
---------- ---------- ----------
      1001 SMITH           80000
```

```
SQL> SELECT * FROM EMP1_AUDIT;

       EID       AUDIT_INFOR
------------   ----------------------------------------------------
      1001       Someone inserted a new row into EMP1 table on 26-
                 MAR-2024 07:27:32 AM
```

```
SQL> CREATE OR REPLACE TRIGGER TRA_AUDIT_DML
  2     AFTER UPDATE ON EMP1
  3     FOR EACH ROW
  4  BEGIN
  5     INSERT INTO EMP1_AUDIT VALUES(:NEW.EID, 'Someone updated a
row on EMP1 table on'||' '||TO_CHAR(SYSDATE, 'DD-MON-YYYY HH:MI:SS
AM'));
  6  END;
  7  /

Trigger created
```

```
SQL> CREATE OR REPLACE TRIGGER TRA_AUDIT_DML
  2     AFTER DELETE ON EMP1
  3     FOR EACH ROW
  4  BEGIN
  5     INSERT INTO EMP1_AUDIT VALUES(:NEW.EID, 'Someone deleted a
row from EMP1 table on'||' '||TO_CHAR(SYSDATE, 'DD-MON-YYYY HH:MI:SS
AM'));
  6  END;
  7  /

Trigger created
```

```
SQL> CREATE OR REPLACE TRIGGER TRA_AUDIT_DML
  2     AFTER INSERT OR UPDATE OR DELETE ON EMP1
  3     FOR EACH ROW
  4  BEGIN
  5     INSERT INTO EMP1_AUDIT VALUES(:NEW.EID, 'Someone peforming
DML operations on EMP1 table on'||' '||TO_CHAR(SYSDATE, 'DD-MON-YYYY
HH:MI:SS AM'));
  6  END;
  7  /

Trigger created
```

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 handling by DBA only.

Syntax:

Create or replace trigger <trigger name>
BEFORE/AFTER CREATE OR ALTER OR DROP OR RENAME
ON <user name.SCHEMA>
[FOR EACH ROW]
[DECLARE <variable declaration>;]
BEGIN
<exec statements>;
END;
/

```
SQL> CREATE OR REPLACE TRIGGER TRDDL
  2     AFTER CREATE ON NIRMALA.SCHEMA
  3  BEGIN
  4     RAISE_APPLICATION_ERROR(-20456, 'Someone creating a new
table in you NIRMALA database, so please check it!!!');
  5  END;
  6  /

Trigger created.

SQL> CREATE TABLE T1(SNO INT);
CREATE TABLE T1(SNO INT)
*
ERROR at line 1:
ORA-04088: error during execution of trigger 'NIRMALA.TRDDL'
ORA-00604: error occurred at recursive SQL level 1
ORA-20456: Someone creating a new table in you NIRMALA database, so
please
check it!!!
ORA-06512: at line 2
```

```
SQL> CREATE OR REPLACE TRIGGER TRDDL
  2     AFTER ALTER ON NIRMALA.SCHEMA
  3  BEGIN
  4     RAISE_APPLICATION_ERROR(-20456, 'Someone try to alter the
table in you NIRMALA database, so please check it!!!');
  5  END;
  6  /

Trigger created.

SQL> ALTER TABLE EMP1 ADD EADD VARCHAR2(10);
ALTER TABLE EMP1 ADD EADD VARCHAR2(10)
```

```
*
ERROR at line 1:
ORA-04088: error during execution of trigger 'NIRMALA.TRDDL'
ORA-00604: error occurred at recursive SQL level 1
ORA-20456: Someone try to alter the table in you NIRMALA database,
so please
check it!!!
ORA-06512: at line 2
```

```
SQL> CREATE OR REPLACE TRIGGER TRDDL
  2     AFTER DROP ON NIRMALA.SCHEMA
  3  BEGIN
  4     RAISE_APPLICATION_ERROR(-20456, 'Someone try to DELETE the
table in you NIRMALA database, so please check it!!!');
  5  END;
  6  /

Trigger created.

SQL> DROP TABLE EMP1;
DROP TABLE EMP1
           *
ERROR at line 1:
ORA-04088: error during execution of trigger 'NIRMALA.TRDDL'
ORA-00604: error occurred at recursive SQL level 1
ORA-20456: Someone try to DELETE the table in you NIRMALA database,
so please
check it!!!
ORA-06512: at line 2


SQL> DROP TABLE EMP1 PURGE;
DROP TABLE EMP1 PURGE
           *
ERROR at line 1:
ORA-04088: error during execution of trigger 'NIRMALA.TRDDL'
ORA-00604: error occurred at recursive SQL level 1
ORA-20456: Someone try to DELETE the table in you NIRMALA database,
so please
check it!!!
ORA-06512: at line 2
```

```
SQL> CREATE OR REPLACE TRIGGER TRDDL
  2     AFTER RENAME ON NIRMALA.SCHEMA
  3  BEGIN
  4     RAISE_APPLICATION_ERROR(-20456, 'Someone try to rename
operation on table in you NIRMALA database, so please check it!!!');
  5  END;
  6  /
```

```
Trigger created.

SQL> RENAME EMP1 TO EMPDETAILS;
RENAME EMP1 TO EMPDETAILS
*
ERROR at line 1:
ORA-04088: error during execution of trigger 'NIRMALA.TRDDL'
ORA-00604: error occurred at recursive SQL level 1
ORA-20456: Someone try to rename operation on table in you NIRMALA
database, so
please check it!!!
ORA-06512: at line 2
```

```
SQL> CREATE OR REPLACE TRIGGER TRDDL
  2     AFTER CREATE OR ALTER OR DROP OR RENAME ON NIRMALA.SCHEMA
  3  BEGIN
  4     RAISE_APPLICATION_ERROR(-20456, 'Someone try to peform DDL
operations on NI
RMALA database, so please check it!!!!');
  5  END;
  6  /

Trigger created.
```

Note: To view all trigger in oracle DB the we have to use “USER_TRIGGERS” data dictionary.

```
SQL> DESC USER_TRIGGERS;

SQL> SELECT TRIGGER_NAME FROM USER_TRIGGERS;

TRIGGER_NAME
--------------------------------------------------
TDELETE
TR1
TINSERT
TUPDATE
```

Syntax to drop a trigger:

SQL> DROP TRIGGER <trigger name>;

```
SQL> DROP TRIGGER TDELETE;
DROP TRIGGER TDELETE
*
ERROR at line 1:
ORA-04088: error during execution of trigger 'NIRMALA.TRDDL'
```

```
ORA-00604: error occurred at recursive SQL level 1
ORA-20456: Someone try to peform DDL operations on NIRMALA database,
so please
check it!!!!
ORA-06512: at line 2
```

Note: Now if we are trying to delete the trigger as then it will be preventing because of last trigger, to solve this issue fast create a dummy trigger again on that name only and now you can try to delete.

```
SQL> CREATE OR REPLACE TRIGGER TRDDL
  2     AFTER CREATE OR ALTER OR DROP OR RENAME ON NIRMALA.SCHEMA
  3  BEGIN
  4     NULL;
  5  END;
  6  /

Trigger created.

SQL> DROP TRIGGER TRDDL;

Trigger dropped.
```

UTL_FILES Package

- With UTL file package we can write data into and read data from files.

Members of UTL file package:

FILE_TYPE:

- It is a type used to declare file variable.
- Syntax:
  <file variable name> UTL_FILE.FILE_TYPE

FOPEN ():

- It a function used to open file.
- Syntax:
  <file variable name>: = UTL_FILE.FOPEN(directory name, <file name>, mode);
- Here, mode is written (W), read (R), append(A).

PUT_LINE:

- It is a procedure used to write data into file.
- Syntax:
  UTL_FILE.PUT_LINE (<file variable name>, <type data>);

GET_LINE:

- It is a procedure used to read data from file.
- Syntax:
  UTL_FILE.GET_LINE (<file variable name>, <string variable>);

FCLOSE:

- It is procedure used to close file.
- Syntax:
  UTL_FILE.FCLOSE (<file variable name>);

```
SQL> CONN
Enter user-name: system/tiger
Connected.

SQL> GRANT CREATE ANY DIRECTORY TO NIRMALA;

Grant succeeded.

SQL> CONN
Enter user-name: nirmala/nirmala
Connected.
```

Now you have to create a folder in any directory, so created ‘NIRMALA’ folder in E directory.

```
SQL> CREATE DIRECTORY XYZ AS 'E:\NIRMALA';

Directory created.

SQL> CONN
Enter user-name: system/tiger
Connected.

SQL> GRANT READ, WRITE ON DIRECTORY XYZ TO NIRMALA;

Grant succeeded.

SQL> CONN
Enter user-name: nirmala/nirmala
Connected.
```

Ex: WAP to write data into text file.

```
SQL> DECLARE
  2     FV UTL_FILE.FILE_TYPE;
```

```
  3  BEGIN
  4     FV:=UTL_FILE.FOPEN('XYZ', 'FILE1.TXT', 'W');
  5     UTL_FILE.PUT_LINE(FV, 'HELLO');
  6     UTL_FILE.FCLOSE(FV);
  7  END;
  8  /

PL/SQL procedure successfully completed.
```

Note: If you now go to that directory folder, you can see the FILE.txt is created with “HELLO” content.

Ex: WAP to read data from text file.

```
SQL> DECLARE
  2     FV UTL_FILE.FILE_TYPE;
  3     S VARCHAR2(100);
  4  BEGIN
  5     FV:=UTL_FILE.FOPEN('XYZ', 'FILE1.txt', 'R');
  6     LOOP
  7        UTL_FILE.GET_LINE(FV, S);
  8        DBMS_OUTPUT.PUT_LINE(S);
  9     END LOOP;
 10     EXCEPTION
 11        WHEN NO_DATA_FOUND THEN
 12        UTL_FILE.FCLOSE(FV);
 13  END;
 14  /

PL/SQL procedure successfully completed.

SQL> SET SERVEROUTPUT ON;
SQL> /
HELLO

PL/SQL procedure successfully completed.
```

Ex: WAP to write employee data into text file.

```
SQL> DECLARE
  2     F1 UTL_FILE.FILE_TYPE;
  3     TYPE EMP_ARRAY IS TABLE OF EMPLOYEE%ROWTYPE;
  4     E EMP_ARRAY;
  5  BEGIN
  6     F1:=UTL_FILE.FOPEN('XYZ', 'EMP.txt', 'W');
  7     SELECT * BULK COLLECT INTO E FROM EMPLOYEE;
  8     FOR I IN E.FIRST..E.LAST
```

```
  9     LOOP
 10        UTL_FILE.PUT_LINE(F1, E(I).EID||', '||E(I).ENAME||',
'||E(I).SAL);
 11     END LOOP;
 12     UTL_FILE.FCLOSE(F1);
 13  END;
 14  /

PL/SQL procedure successfully completed.
```

Ex: WAP to read employee data from text file.

```
SQL> DECLARE
  2     F1 UTL_FILE.FILE_TYPE;
  3     S VARCHAR2(1000);
  4  BEGIN
  5     F1:=UTL_FILE.FOPEN('XYZ', 'EMP.txt', 'R');
  6     LOOP
  7        UTL_FILE.GET_LINE(F1, S);
  8        DBMS_OUTPUT.PUT_LINE(S);
  9     END LOOP;
 10     EXCEPTION
 11        WHEN NO_DATA_FOUND THEN
 12        UTL_FILE.FCLOSE(F1);
 13  END;
 14  /

1, SMITH, 10000
2, JONES, 25000
3, WARD, 10000
4, ADAMS, 32000

PL/SQL procedure successfully completed.
```

Note: To view directory related to physical path then use the following data dictionary is "ALL_DIRECTORIES".

```
SQL> DESC ALL_DIRECTORIES;

SQL> SELECT DIRECTORY_NAME, DIRECTORY_PATH FROM ALL_DIRECTORIES;

DIRECTORY_NAME                 DIRECTORY_PATH
------------------------       ----------------------------------------
XYZ                            E:\NIRMALA
```

Syntax to drop directory:

SQL> DROP DIRECTORY <directory name>;

```
SQL> DROP DIRECTORY XYZ;

Directory dropped.
```

Data Pump

- It is a tool which is used to transfer database from one location to another location. Data pump comes with two tools those are,
  - EXPDP: export data pump
  - IMPDP: import data pump

EXPDP:

- This tool is used to copy data from database to dump file.

  Syntax:
  EXPDP username / password DIRECTORY=<directory name>
  DUMPFILE=<dump file name> SCHEMAS=<schema name>

Step 1: Go to Oracle SQL Plus tool and create directory before that make sure the folder will there on that directory:

```
SQL> CONN
Enter user-name: system/tiger
Connected.

SQL> CREATE DIRECTORY XYZ AS 'E:\DUMP';

Directory created.
```

Step 2: Now go to open command prompt

```
C:\Users\Nirmala>EXPDP system/tiger DIRECTORY=XYZ
DUMPFILE=NIRMALA.DMP SCHEMAS=NIRMALA

Export: Release 19.0.0.0.0 - Production on Tue Mar 26 09:30:28 2024
Version 19.3.0.0.0
……
……
Master table "SYSTEM"."SYS_EXPORT_SCHEMA_01" successfully
loaded/unloaded
******************************************************************
Dump file set for SYSTEM.SYS_EXPORT_SCHEMA_01 is:
  E:\DUMP\NIRMALA.DMP
Job "SYSTEM"."SYS_EXPORT_SCHEMA_01" successfully completed at Tue
Mar 26 09:31:26 2024 elapsed 0 00:00:54
```

Note: After some time while processing done, go to the directory location where we created the dump file in system and check it.

IMPDP:

- This tool is used to import data from dump file to oracle DB.

  Syntax:
  IMPDP username/password DIRECTORY=<directory name>
  DUMPFILE=<dump file name>

Note: Before importing dump file data into oracle server first we drop scott user schema from oracle server.

```
SQL> CONN
Enter user-name: system/tiger
Connected.
SQL> DROP USER NIRMALA CASCADE;

User dropped.

SQL> CONN
Enter user-name: nirmala/nirmala
ERROR:
ORA-01017: invalid username/password; logon denied

Warning: You are no longer connected to ORACLE.
```

Afte tha go to CMD

```
C:\Users\Nirmala>IMPDP system/tiger DIRECTORY=XYZ
DUMPFILE=NIRMALA.DMP

Import: Release 19.0.0.0.0 - Production on Tue Mar 26 10:02:14 2024
Version 19.3.0.0.0
……
……
Job "SYSTEM"."SYS_IMPORT_FULL_01" successfully completed at Tue Mar
26 10:02:41 2024 elapsed 0 00:00:27
```

Now if you back to SQL plus and try to connect the user, you can connect

```
SQL> CONN
Enter user-name: nirmala/nirmala
Connected.
```

Dynamic SQL

- Dynamic SQL is a programming technique to build SQL statements at runtime.

  Ex:
  DROP TABLE EMP; ---------->static

  Ex:
  TNAME = '&TNAME ';
  DROP TABLE TNAME; -------> dynamic

- Dynamic SQL is useful when we don't know table name, column name until runtime.
- Dynamic SQL commands (DDL, DML, DQL, DCL) can be executed by using "EXECUTE IMMEDIATE" statement.
- EXECUTE IMMEDIATE statement is used to execute DDL commands (or) dynamic SQL command.

  Syntax:
  EXECUTE IMMEDIATE 'dynamic SQL command';

Ex: WASP to drop a table at runtime.

```
SQL> CREATE OR REPLACE PROCEDURE DROP_TABLE(N IN VARCHAR2)
  2  IS
  3  BEGIN
  4     EXECUTE IMMEDIATE 'DROP TABLE '||N;
  5  END;
  6  /

Procedure created.

SQL> EXECUTE DROP_TABLE('TEST');

PL/SQL procedure successfully completed.

SQL> CREATE OR REPLACE PROCEDURE DROP_TABLE(N IN VARCHAR2)
  2  IS
  3  BEGIN
  4     EXECUTE IMMEDIATE 'DROP TABLE '||N||' PURGE';
  5  END;
  6  /

Procedure created.
```

```
SQL> EXECUTE DROP_TABLE('TEST1');

PL/SQL procedure successfully completed.
```

Ex: WASP to drop any object at runtime.
Means:

DROP TABLE <table name>;
DROP VIEW <view name>;
DROP SEQUENCE <sequence name>;
DROP INDEX <index name>;
DROP SYNONYM <synonym name>;

```
SQL> CREATE OR REPLACE PROCEDURE DROP_OBJ(T IN VARCHAR2, N IN
VARCHAR2)
  2  IS
  3  BEGIN
  4     EXECUTE IMMEDIATE 'DROP '||T||' '||N;
  5  END;
  6  /

Procedure created.
```

Collections

- Collection is a group of elements of particular data type and elements are accessed by using index.pl/sql supporting three types of collections those are,
  - PL/SQL table (or) associated array (or) index by table
  - VARRAY
  - Nested table

PL/SQL table:

- PL/SQL table is a user defined type which is used to store number of data items either integers or characters.
- When we use PL/SQL table then we follow the following two steps mechanism.

Step 1: Syntax to declare a type
TYPE <type name> IS TABLE OF <datatype(size)> INDEX BY BINARY_INTEGER;

Step 2: Syntax to declare variable
<Variable name> <type name>;

Ex: WAPLSP to print integer elements by using collection.

```
SQL> DECLARE
  2     TYPE NUM_ARRAY IS TABLE OF NUMBER(4) INDEX BY
BINARY_INTEGER;
  3     X NUM_ARRAY;
  4  BEGIN
  5     FOR I IN 1..10
  6     LOOP
  7        X(I):=I*10;
  8        DBMS_OUTPUT.PUT_LINE(X(I));
  9     END LOOP;
 10  END;
 11  /
10
20
30
40
50
60
70
80
90
100

PL/SQL procedure successfully completed.
```

Ex: WAPLSP to print all departments names by using collection.

```
SQL> DECLARE
  2     TYPE DNAME_ARRAY IS TABLE OF VARCHAR2(10) INDEX BY
BINARY_INTEGER;
  3     D DNAME_ARRAY;
  4  BEGIN
  5     FOR I IN 1..4
  6     LOOP
  7        SELECT DNAME INTO D(I) FROM DEPT WHERE DEPTNO=I*10;
  8        DBMS_OUTPUT.PUT_LINE(D(I));
  9     END LOOP;
 10  END;
 11  /
ACCOUNTING
RESEARCH
SALES
OPERATIONS

PL/SQL procedure successfully completed.
```

Note: In the above example select statement is inside a loop so that no. Of request is going to increase burden on database and reduce performance.to overcome this problem we use "BULK COLLECT" clause.

BULK COLLECT:

- By using BULK COLLECT in a single request we can get all elements from database server and store those elements in a collection. So that bulk collect reduce number of trips to database server and improves performance.

```
SQL> DECLARE
  2     TYPE DNAME_ARRAY IS TABLE OF VARCHAR2(10) INDEX BY
BINARY_INTEGER;
  3     D DNAME_ARRAY;
  4  BEGIN
  5     SELECT DNAME BULK COLLECT INTO D FROM DEPT;
  6     FOR I IN 1..4
  7     LOOP
  8        DBMS_OUTPUT.PUT_LINE(D(I));
  9     END LOOP;
 10  END;
 11  /
ACCOUNTING
RESEARCH
SALES
OPERATIONS

PL/SQL procedure successfully completed.
```

Collection Methods:

- First: return index value of the first element.
- Last: return index value of the last element.
- Next: return index value of the next element.
- Prior: return index value of previous element.

Note: All these methods are used by the collection name.

Syntax:

<collection name>. <method>

```
SQL> DECLARE
  2     TYPE DNAME_ARRAY IS TABLE OF VARCHAR2(10) INDEX BY
BINARY_INTEGER;
  3     D DNAME_ARRAY;
  4  BEGIN
  5     SELECT DNAME BULK COLLECT INTO D FROM DEPT;
```

```
  6     FOR I IN D.FIRST..D.LAST
  7     LOOP
  8        DBMS_OUTPUT.PUT_LINE(D(I));
  9     END LOOP;
 10  END;
 11  /
ACCOUNTING
RESEARCH
SALES
OPERATIONS

PL/SQL procedure successfully completed.
```

Ex: WAPLSP on bulk collecting with forward navigation using "for" loop with records type.

```
SQL> DECLARE
  2     TYPE DNAME_ARRAY IS TABLE OF DEPT%ROWTYPE INDEX BY
BINARY_INTEGER;
  3     D DNAME_ARRAY;
  4  BEGIN
  5     SELECT * BULK COLLECT INTO D FROM DEPT;
  6     FOR I IN D.FIRST..D.LAST
  7     LOOP
  8        DBMS_OUTPUT.PUT_LINE(D(I).DEPTNO||', '||D(I).DNAME||',
'||D(I).LOC);
  9     END LOOP;
 10  END;
 11  /
10, ACCOUNTING, NEW YORK
20, RESEARCH, DALLAS
30, SALES, CHICAGO
40, OPERATIONS, BOSTON

PL/SQL procedure successfully completed.
```

Ex: WAPLSP on bulk collecting with backward navigation using "for" loop with record type.

```
SQL> DECLARE
  2     TYPE DNAME_ARRAY IS TABLE OF DEPT%ROWTYPE INDEX BY
BINARY_INTEGER;
  3     D DNAME_ARRAY;
  4  BEGIN
  5     SELECT * BULK COLLECT INTO D FROM DEPT;
  6     FOR I IN REVERSE D.FIRST..D.LAST
  7     LOOP
```

```
  8         DBMS_OUTPUT.PUT_LINE(D(I).DEPTNO||', '||D(I).DNAME||',
'||D(I).LOC);
  9      END LOOP;
 10   END;
 11   /
40, OPERATIONS, BOSTON
30, SALES, CHICAGO
20, RESEARCH, DALLAS
10, ACCOUNTING, NEW YORK

PL/SQL procedure successfully completed.
```

Ex on bulk collecting with forward navigation using "while loop" with record type.

```
SQL> DECLARE
  2     TYPE DNAME_ARRAY IS TABLE OF DEPT%ROWTYPE INDEX BY
BINARY_INTEGER;
  3     D DNAME_ARRAY;
  4     X NUMBER(10);
  5   BEGIN
  6     SELECT * BULK COLLECT INTO D FROM DEPT;
  7     X:=D.FIRST;
  8     WHILE(X<=D.LAST)
  9     LOOP
 10         DBMS_OUTPUT.PUT_LINE(D(X).DEPTNO||', '||D(X).DNAME||',
'||D(X).LOC);
 11         X:=D.NEXT(X);
 12      END LOOP;
 13   END;
 14   /
10, ACCOUNTING, NEW YORK
20, RESEARCH, DALLAS
30, SALES, CHICAGO
40, OPERATIONS, BOSTON

PL/SQL procedure successfully completed.
```

Ex on bulk collecting with backward navigation using "while loop" with record type.

```
SQL> DECLARE
  2     TYPE DNAME_ARRAY IS TABLE OF DEPT%ROWTYPE INDEX BY
BINARY_INTEGER;
  3     D DNAME_ARRAY;
  4     X NUMBER(10);
  5   BEGIN
```

```
  6     SELECT * BULK COLLECT INTO D FROM DEPT;
  7     X:=D.LAST;
  8     WHILE(X>=D.FIRST)
  9     LOOP
 10         DBMS_OUTPUT.PUT_LINE(D(X).DEPTNO||', '||D(X).DNAME||',
'||D(X).LOC);
 11         X:=D.PRIOR(X);
 12     END LOOP;
 13  END;
 14  /
40, OPERATIONS, BOSTON
30, SALES, CHICAGO
20, RESEARCH, DALLAS
10, ACCOUNTING, NEW YORK

PL/SQL procedure successfully completed.
```

VARRAY:

- VARRAY also user defined types which is used to store number of data items in a single unit and declare with size.
- Here number of elements are limited as per array size.

```
SQL> DECLARE
  2     TYPE T1 IS VARRAY(10) OF VARCHAR2(10);
  3     V_T T1;
  4  BEGIN
  5     SELECT ENAME BULK COLLECT INTO V_T FROM EMP WHERE
ROWNUM<=10;
  6     FOR I IN V_T.FIRST..V_T.LAST
  7     LOOP
  8        DBMS_OUTPUT.PUT_LINE(V_T(I));
  9     END LOOP;
 10  END;
 11  /
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK
SCOTT
KING
TURNER

PL/SQL procedure successfully completed.
```

Nested table:

- Nested table also user defined types which is used to store number of data items in a single unit and not declare with size.
- Here number of elements are unlimited.

```
SQL> DECLARE
  2     TYPE T1 IS TABLE OF VARCHAR2(10);
  3     V_T T1;
  4  BEGIN
  5     SELECT ENAME BULK COLLECT INTO V_T FROM EMP WHERE ROWNUM<=5;
  6     FOR I IN V_T.FIRST..V_T.LAST
  7     LOOP
  8        DBMS_OUTPUT.PUT_LINE(V_T(I));
  9     END LOOP;
 10  END;
 11  /
SMITH
ALLEN
WARD
JONES
MARTIN

PL/SQL procedure successfully completed.
```

---------- The END ----------