ORACLE 19 C
(SQL/PLSQL)

NEW CLASS NOTES

BY

Mr. Murali Sir

DURGA SOFTWARE SOLUTIONS
202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500 038.
Phone : 72 07 21 24 27, 80 96 96 96 96, 92 42 21 21 43 | www.durgasoftonline.com

ORACLE 12C

There are mainly two different types of database language in ORACLE. They are:

1) ***SQL*** (Structured Query Language)
2) ***PL/SQL*** (Procedural Language Extension of SQL)
3) ***Dynamic SQL*** (Optional)

ORACLE is a relational database product which is used to store data permanently in secondary storage devices.

If you want to operate ORACLE then we are using following languages:

1) ***SQL:*** It is non-procedural language.
2) ***PL/SQL:*** It is a procedural language.

All organizations store same type of data.

Data: It is a collection of raw facts.

Example:

101	*Dinesh*	*2000*
102	*Mahesh*	*3000*

In above example, there no meaningful data such data is known as raw facts.

Information: It is a collection of meaningful data or processed data.

Example:

EMPID	*ENAME*	*SALARY*
101	*Dinesh*	*2000*
102	*Mahesh*	*3000*

In the above example, there is meaningful data which is in table format which consist of three different fields.

Data Store: It is a place where we can store data or information.

1) Books & Papers
2) Flat files
3) Database

Flat Files:

This is a traditional mechanism which is used to store data or information in individual unrelated files. These files are also called as Flat Files.

Drawbacks of Flat files:

1) **Data Retrieval**
2) **Data Redu :dancy**
3) **Data Integrity**
4) **Data Security**
5) **Data Indexing**

ORACLE 12C

There are mainly two different types of database language in ORACLE. They are:

1) ***SQL*** (Structured Query Language)
2) ***PL/SQL*** (Procedural Language Extension of SQL)
3) ***Dynamic SQL*** (Optional)

ORACLE is a relational database product which is used to store data permanently in secondary storage devices.

If you want to operate ORACLE then we are using following languages:

1) ***SQL:*** It is non-procedural language.
2) ***PL/SQL:*** It is a procedural language.

All organizations store same type of data.

Data: It is a collection of raw facts.

Example:

101	*Dinesh*	*2000*
102	*Mahesh*	*3000*

In above example, there no meaningful data such data is known as raw facts.

Information: It is a collection of meaningful data or processed data.

Example:

EMPID	*ENAME*	*SALARY*
101	*Dinesh*	*2000*
102	*Mahesh*	*3000*

In the above example, there is meaningful data which is in table format which consist of three different fields.

Data Store: It is a place where we can store data or information.

1) Books & Papers
2) Flat files
3) Database

Flat Files:

This is a traditional mechanism which is used to store data or information in individual unrelated files. These files are also called as Flat Files.

Drawbacks of Flat files:

1) **Data Retrieval**
2) **Data Redu dancy**
3) **Data Integrity**
4) **Data Security**
5) **Data Indexing**

1) **Data Retrieval:**
If we want to retrieve data from flat files then we must develop application program in high level languages, where as if we want to retrieve data from databases then we are using SEQUEL Language.

SEQUEL (Structured English Query Language)

2) **Data Redundancy:**
Sometimes we are maintaining multiple copies of the same data in different locations this data is also called as duplicate data or redundant data.

In Flat files mechanism when we are modifying data in one location it is not effected in another location. This is called INCONSISTENCY.

In databases, every transaction internally has 4 properties. These properties are known as ACID properties.

A: Atomicity (ROLLBACK)

C: Consistency

I: Isolation

D: Durability (COMMIT)

These properties only automatically maintain consistent data in databases.

3) **Data Integrity:**
Integrity means to maintain proper data. if we want to maintain proper data then we are defining set of rules, these rules are also called as "Business rules".

In databases, we are maintaining proper data using constraints, triggers. If we want to maintain proper data in flat files we must develop application programs in high level languages like COBOL, JAVA etc.

4) **Data Security:**
Data stored in flat files cannot be secured because a flat file doesn't provide security mechanism. Where as databases provides "ROLE based security".

5) **Data Indexing:**
If we want to retrieve data very quickly from database then we are using indexing mechanism. Whereas flat files doesn't provide indexing mechanism.

To overcome from all the above problems new software is used by all organizations to store data or information in secondary storage devices. This is called DBMS software.

DBMS (Database Management System)

A database management system (DBMS) is system software for creating and managing databases. The DBMS provides users and programmers with a systematic way to create, retrieve, update and manage data.

Example

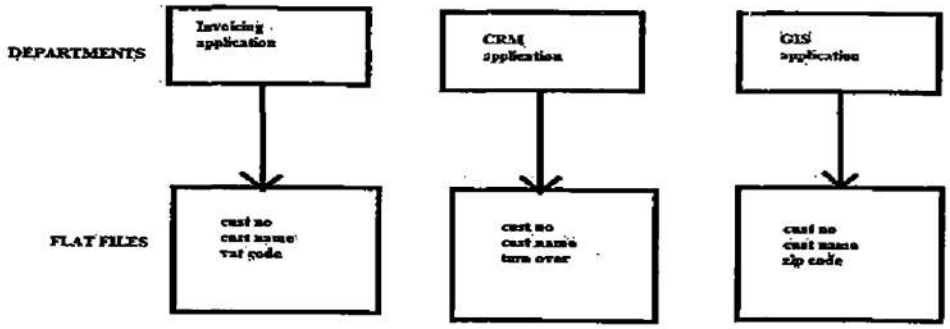
ORACLE, FOXPRO, DB2, TERADATA, SQLSERVER, SYBASE, MYSQL, INGRESS, INFORMIX, SQLLITE etc

Flat File Approach to Data Management

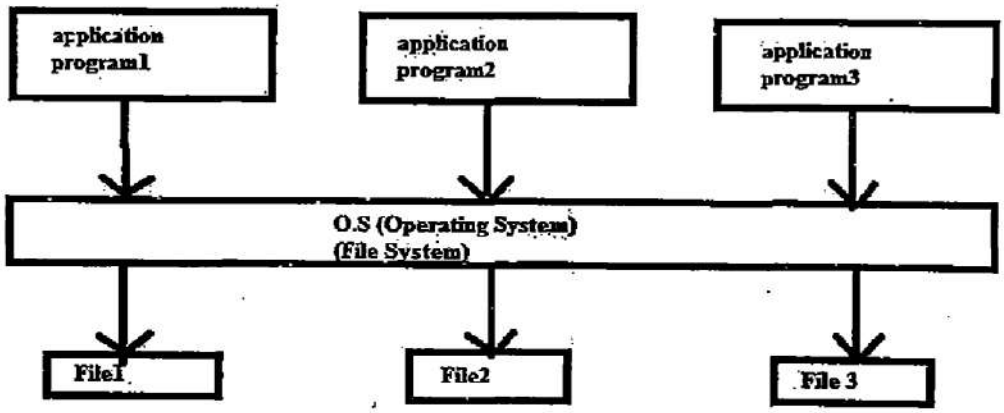
Prior to 1960 organisation stores data or information in secondary storage devices by using flat file mechanism in this method every application program in the organisation maintain on file from other application program separately.

In file based approach, every application program in the organization maintains its own file separate from other application program.

File Based Approach:



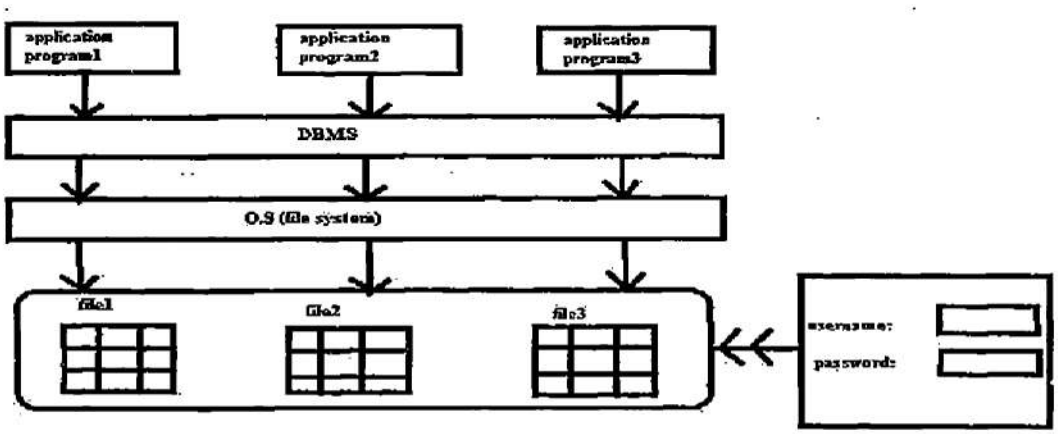
Here custno and custname attributes are duplicate data.



DBMS Approach to Data Management (This Is Old Approach)

Once we are installing DBMS software into our system then automatically some place is created in hard disc, this is called Physical Database, and also automatically a user interface is created.

Through the user interface we can also directly interactive with the database or indirectly interactive with the database using application programs in high level languages.



After Installing DBMS Software

Database: It is an organized collection of interrelated data used by application program in an organization. Once data stored in database it can be shared by number of users simultaneously and also this data can be integrated.

Every database having two types of structure these are:

1) **Logical Structure**
2) **Physical Structure**

1) **Logical Structure:** Structure which is not visible in OS is called logical structure.
   Example: Oracle logical structure having table, views, sequences, synonyms.

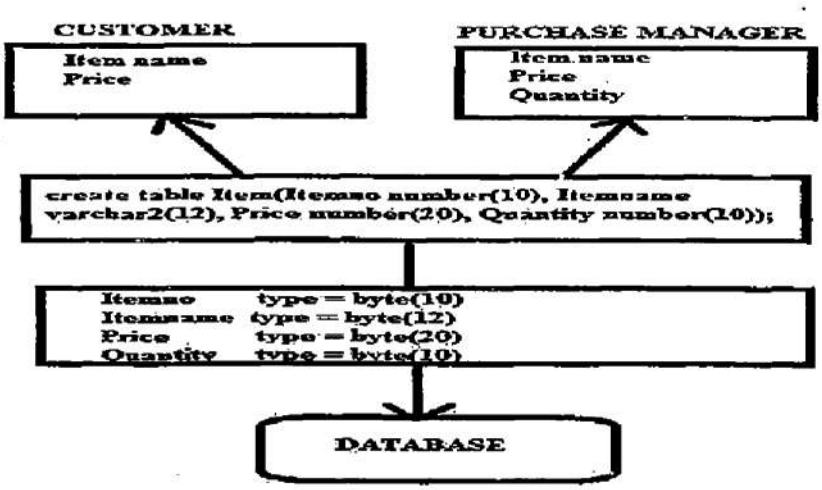
2) **Physical Structure:** Structure which is visible in OS is called physical structure. Physical structure is handling by database administrator (DBA).

DBMS Architecture:

American National Standard Institute (ANSI) has established three level architecture for database. This architecture is also called as "ANSI/SPARC" (Standard Planning and Requirements Committee) architecture. Main objective of DBMS architecture is to separate user's view of the database from the where physically it is stored.

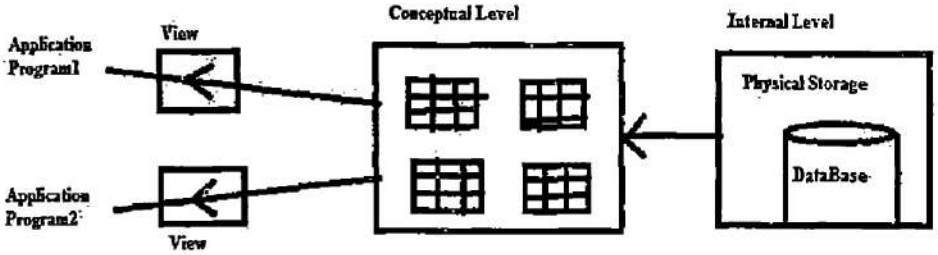
This architecture mainly consists of three levels. They are:

1) **Conceptual Level**
2) **External Level**
3) **Internal Level**



3) **Internal Level:** Internal level describes how physically data is stored in database. This level is handled by database administrator only. In relational databases indexes, cluster is available in internal level.

DBMS Architecture



Data Models

How data is represented at the conceptual level defined by means of "Data Model" in the history of database design three data models have been used.

1. Hierarchical Data Model
2. Network Data Model
3. Relational Data Model

1) **Hierarchical Data Model**

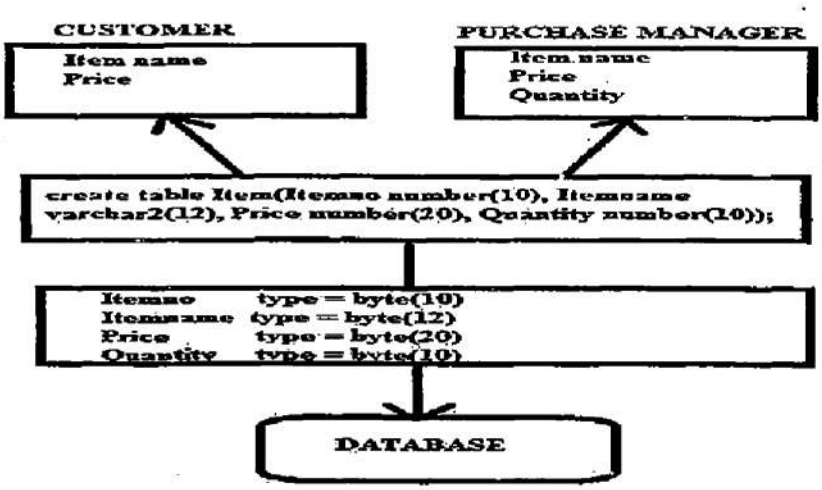
This model introduced in 1960. In this data model organizes data in tree like structure, we are representing data in parent child hierarchy. In this data model also data is represented in the format of records and also record type is also same as table in relational data model.

This Data model having more duplicate records because this data model is implemented based on one-to–many relationships. That is why in this data model always child segments are repeated.

In these data model products, we are retrieved data very slowly because in these data model products data base servers searching data based on root node onwards.

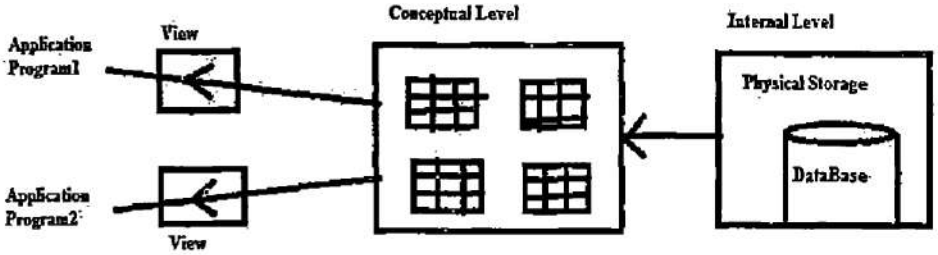
In 1960, IBM introduced IMS (Information Management System) product based on Hierarchical Data Model.

If we want to operate hierarchical data model products then we are using procedural language.



3) **Internal Level:** Internal level describes how physically data is stored in database. This level is handled by database administrator only. In relational databases indexes, cluster is available in internal level.

DBMS Architecture



Data Models

How data is represented at the conceptual level defined by means of "Data Model" in the history of database design three data models have been used.

1. Hierarchical Data Model
2. Network Data Model
3. Relational Data Model

1) **Hierarchical Data Model**

This model introduced in 1960. In this data model organizes data in tree like structure, we are representing data in parent child hierarchy. In this data model also data is represented in the format of records and also record type is also same as table in relational data model.

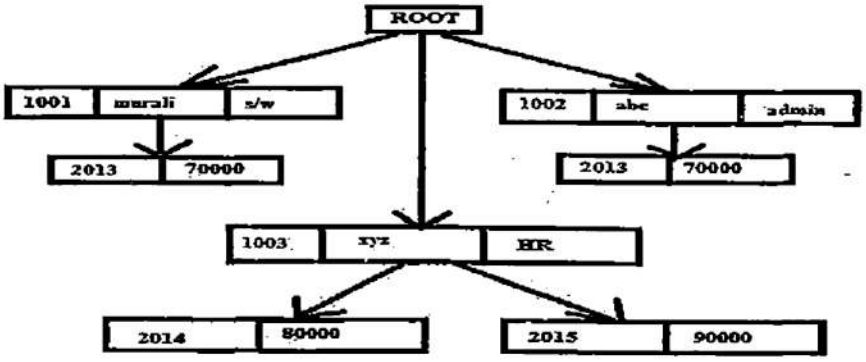
This Data model having more duplicate records because this data model is implemented based on one-to–many relationships. That is why in this data model always child segments are repeated.

In these data model products, we are retrieved data very slowly because in these data model products data base servers searching data based on root node onwards.

In 1960, IBM introduced IMS (Information Management System) product based on Hierarchical Data Model.

If we want to operate hierarchical data model products then we are using procedural language.

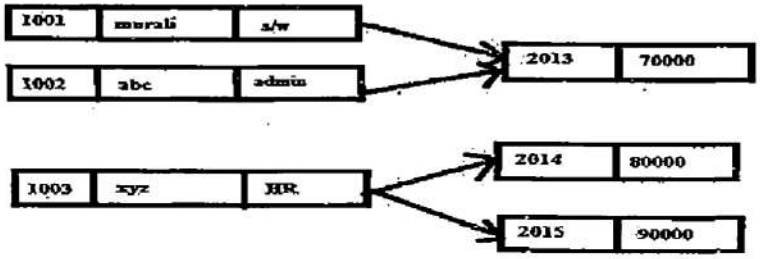
Hierarchical Data Model



2) Network Data Model

In 1970, CODASYL (Conference or Data System Language) committee introduced network data model. This data model is implemented based on many– to – many relationships. In this data model also data is stored in format of records and also records type is also same as table in Relational Data model.

In 1970's IBM company introduced IDMS (Information Data Management System) based on network data model. If we want to operate network data model products then we must use procedural language.



3) Relational Data Model

In 1970 E.F.Codd introduced Relational Data Model. In this data model we are storing data in 2-dimensional tables.

Relational Data Model

Primary key

EMP MASTER TABLE

empno	empname	deptname
1001	murali	s/w
1002	abc	admin
1003	xyyz	HR

Foreign key

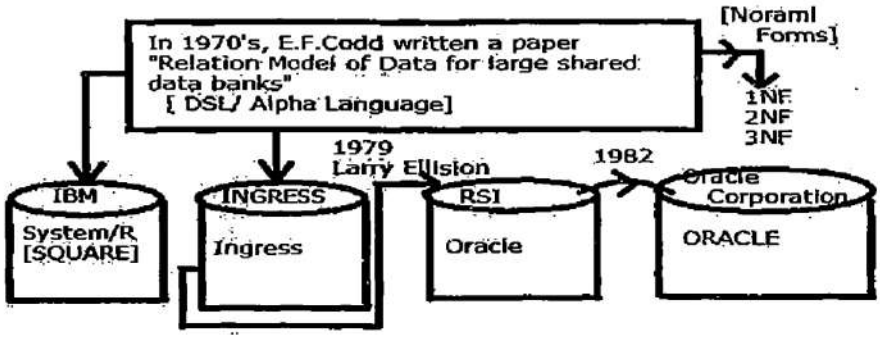
LOAN DETAILS CHILD TABLE

empno	year	loan amount
1001	2013	70000
1002	2013	70000
1003	2014	80000
1003	2015	90000

Relational data model mainly consist of 3 components.

1. Collection of database Objects. (Table,views,synonyms,Index,Procedures etc)
2. Set of Operators.
3. Set of Integrity rules.

If we want to operate relational data model product then we are using Non- Procedural Language SQL



- ✓ ***1977*** - Larry Ellison, Bob Miner and Ed Oates founded new company "Software Development Laboratories" (SDL) to undertake development work.
- ✓ ***1977*** - Bruce Scott was hire as first employee at SDL Company.
- ✓ ***1978*** - SDL Introduced oracle version1 (never released).
- ✓ ***1979*** - Version 2.0 of Oracle was released and it became first commercial relational database and first SQL database. The company changed its name to Relational Software Inc. (RSI).
- ✓ ***1979*** - SDL change its name to Relational Software Inc. (RSI).
- ✓ ***1981*** - RSI started developing tools for Oracle.
- ✓ ***1982*** - RSI was name changed into Oracle Corporation.

1) **Oracle 2.0 (1979)**
   - First public release
   - In this 2.0 only basic SQL functionality is there "joins"
2) **Oracle 3.0 (1983)**
   - Rewritten in C language
   - Commit, Roll back
3) **Oracle 4.0 (1984)**
   - Read consistency
   - Exp/imp utility programs [export/import]
4) **Oracle 5.0 (1985)**
   - Introduce client server architecture
5) **Oracle 6.0 (1988)**
   - Introduced PL/SQL
   - Row level locks
6) **Oracle 7.0 (1992)**
   - Roles
   - Integrity constraints
   - Stored Procedures
   - Stored Functions
   - Packages
   - Triggers
   - Data type "varchar" changed into "varchar2"
   - Truncate table
7) **Oracle 7.1 (1994)**
   - Introduced dynamic SQL
   - ANSI/ISO SQL-92
8) **Oracle 7.2 (1995)**
   - Inline views or sub queries used in from clause
   - Re cursor (cursor variable)
9) **Oracle 7.3 (1996)**
   - Bit map indexes
   - Utl_file package
10) **Oracle 8.0 (1997)**
   - Object technology
   - Columns increased per a table up to "1000"
   - Nested table, varray
   - Large object (Clob,Blob,bfile,data type)
   - Instead of triggers

11) Oracle 8i (i- internet) (1999)

- Materialized views
- Function based indexes
- Case conditional statements
- Analytical functions
- Autonomous transactions
- Rollup, cube
- Bulk bind
- Trim () function
- Instead of triggers

12) Oracle 9i (2001)

- 9i joins or ansi joins
- Merge statements
- Multi table insert
- Flash back queries
- Renaming a column

13) Oracle 10g (g- grid technology) (2003)

- Recyclebin
- Flash back table
- Indices of clause
- Regular expressions
- WM_concat() function

14) Oracle 11g (2007)

- Introduced continue statement in PL/SQL loops
- Read only tables
- Virtual Columns
- Pivot() function
- Regular expression Conn()function
- Compound trigger
- Enable, disable clauses used in trigger specification
- Follow clause in triggers
- Sequences used in PL/SQL without using dual table
- Named, mixed notations are used in a subprogram executed used select statement

15) Oracle 12 C (C-Cloud Technology) (2013)

- Invisible Column
- Multiple Indexes
- Truncate table .... Cascade
- Session specific sequence
- New, top-n analysis by using fetch first/next clauses
- New auto increment concept without using row level triggers
- With clause use in local function
- Accessible by clause used in stored procedures

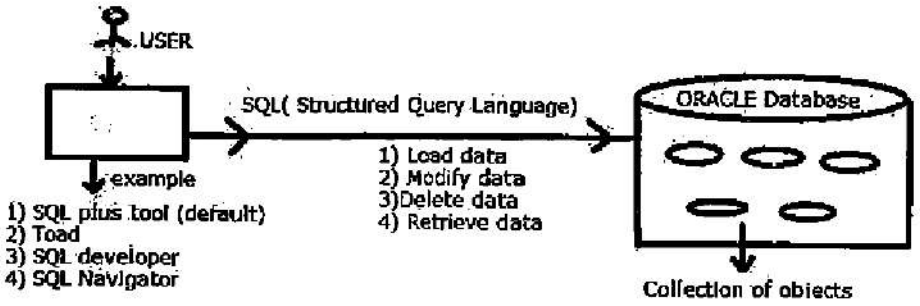
Oracle is a relational database procedure which is used to store data or information permanently in secondary storage devices.

If any user wants to store data in oracle database or retrieve data from oracle database then the users uses following two steps these are:

1) ***Connect*** *(Ex. SQL Plus tool)*
2) ***Communicate*** *(Ex. SQL Language)*

Connect

If any user wants to perform of oracle database that's the user first connect to oracle data by using client tools. Whenever we are installing oracle server then automatically SQL+ client tool is created that's why SQL+ is a default client tool in oracle database.



❖ **SQL Plus Tools**

Start > Programs >Oracle >Oracle DB 11g_home >Application Development >SQL Plus

Every oracle version has following two editions.

1) Express Edition
2) Enterprise Edition

In these two editions by default user name is system and password is manager.

User name: system
Password: manager

Express edition does not support all oracle features.

Example: Recyclebin is not available in express edition.

Enterprise edition support all oracle features and also an enterprise edition having one extra user scot along with system user.

This Scott user has following pre-defined table.

1) Emp
2) Dept

Enterprise Edition
User name: scott
Password: tiger

Oracle 10g, 11g, 12c (Enterprise Edition)

Username: scott
Password: tiger
Error: Account locked displayed for first time. So again we have to connect.
Username: \ sys as sysdba
Password: sys
SQL> alter user scott account unlock:
SQL> conn scott/tiger
Enter password: tiger
Confirm password: tiger

To view user

Syntax: show user;
User is: scott

To clear screen

Syntax: cl scr; (Or) Shift +delete (Through keyboard)

To view particular table:

Syntax: select * from tablename;

In SQL+ tool by default each line having 80 bytes. Whenever our relational table in a line more than 80 bytes that's the table is not display properly.
To overcome this problem we must set SQL plus tool environment by using following syntax.

Syntax: set line 100;

In a SQL plus tool by default each page having 14 lines. Our relational table having more than 14 lines then column are automatically repeated.
To overcome from this problem we must set page size by following syntax.

Syntax: set pagesize 100 (any numbers);

If you want to view all SQL plus tool comments then we have to use following syntax.

Syntax: show all;

Communication

Once users connect to oracle database by using a tool then only those users are allow communicating with oracle database by using SQL language. Basically SQL is a non-procedure language.

SQL is a Non- procedural language, which is used to operate all relational database products.

- In 1970 E.F.Codd introduced Relational Data Model and also DSL/ Alpha language which are used to operate relational data model products.
- In IBM Company "System/R" team modified DSL/Alpha language into "SQUARE" language. Again IBM Company change SQUARE into SEQUEL (Structured English Query Language). Then only SEQUEL became "SQL".
- In 1986 introducing ANSI in SQL
- In 1987 introducing ISO in SQL
- In 1989 releasing first version of ANSI/ISO SQL 89 that is SQL 89
- In 1992 ANSI/ISO SQL92 -> SQL92
- In 1999 ANSI/ISO SQL 99 -> SQL99
- In 2003 ANSI/ISO SQL 03 -> SQL03

SQL Sub Languages in every database

1) **Data Definition Language (DDL)**
   - Create
   - Alter
   - Drop
   - Truncate
   - Rename (Oracle 9i)

   Note: *In all database systems by default all DDL commands are automatically committed.*

2) **Data Manipulation Language (DML)**
   - Insert
   - Update
   - Delete
   - Merge (Oracle 9i)

3) **Data Retrieval Language (DRL) / Data Query Language (DQL)**
   - Select

4) **Transactional Control Language (TCL)**
   - Commit
   - Rollback
   - Save point

5) **Data Control Language (DCL)**
   - Grant
   - Revoke

Data types specify type of data within a table column. Oracle has following data types...

1) **Number (P,S)**
2) **Char - varchar2(max size)**
3) **Date**

1) **Number (P, S)**

P: Precision (total number of digits)
S: Scale (it is used to store fixed, floating point numbers).

Syntax: columnname number (p, s)

Example

```
SQL> create table test (sno number (7, 2));
SQL> insert into test values (12345.67);
SQL> select * from test;

        Sno
    ----------
    12345.67

SQL> insert into test values (123456.7)
Error: value larger than specified precision allows for this column.
```

Note 1: *Whenever we are using number (P, S) then total number of digits before decimal places up to "P-S" number of digits.*

[Example: p-s => 7-2=5]

After Decimal

```
SQL> insert into test values (12345.6789);
SQL> select * from test;
        Sno
    ---------
    12345.68

SQL> insert into test values (12345.6725);
SQL> select * from test;
        Sno
    ----------
    12345.67
```

Note 2: *Oracle server does not return any error whenever we are using number (p, s) and also if we are passing more number of digits after decimal place then oracle server internally automatically rounded that number based on specified "scale".*

Number (P): It is used to store fixed numbers only. Maximum length of the precision is up to "38".

Example

```
SQL> create table test1 (sno number (7));
SQL> insert into test1 values (99.9);
SQL> select * from test1;

        Sno
        ------
        100
SQL> insert into test1 values (99.4);
SQL> select * from test1;

        Sno
        ------
        99
```

2) **Char:** It is used to store fixed length "alpha numeric" data in bytes. Maximum limit is up to 2000 bytes.

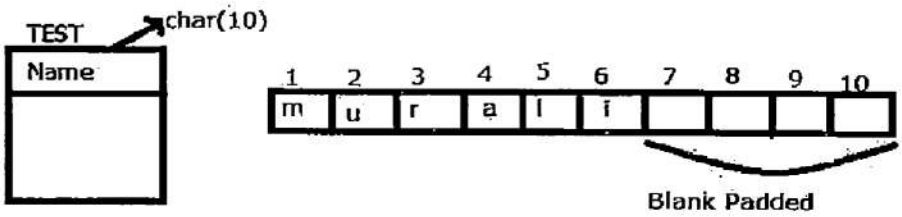
Syntax: columnname char (size);

By default character datatype has one byte. Character datatype has automatically "Blank Padded".

- ***Blank Padded:***

  Whenever we are passing less number of characters than the specified datatype size then oracle server automatically allocates blank spaces after the value.

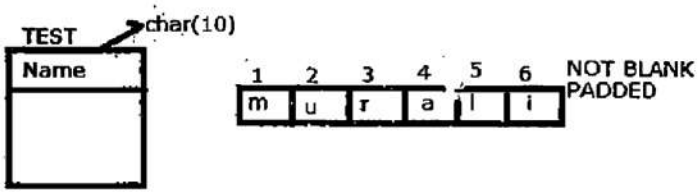
Example



- **Varchar2 (max size)**

  Oracle 7.0 introduced "varchar2()" datatype. It is used to store variable length alphanumeric data in bytes. Maximum limit is up to 4,000 bytes.
  Whenever we are using varchar2() datatype and also if we are try to store less no. of bytes then the maximum size specifies within varchar2() datatype, then oracle server does not add blank spaces in place of remaining bytes this is called variable length alpha numeric data that's why here space were never wasted.

Syntax: columnname varchar2 (max size);



- **Varchar (Size):**

  Prior to oracle 7.0 if you want to store variable length alpha numeric data then we are using varchar data type.

Syntax: columnname varchar (size);

Here maximum size is up to 2000 bytes.

Oracle 7.0 onwards whenever we are creating a column with varchar datatype also then oracle server internally automatically converts varchar data type into varchar2() datatype.
In varchar datatype also when we are trying to store less no. of bytes then the maximum size specified varchar datatype then oracle server does not add blank spaces in place of remaining bytes.

Difference between Varchar & Varchar2 Data Types

Varchar datatype is ANSI standard datatype that's why it's with all relational database products but varchar2() is an oracle standard that's why this datatypes works in oracle database only and also varchar() datatypes stores upto 2000 bytes where as varchar2() datatypes stores up to 4000 bytes.

- ***Long:*** In oracle if you want to store more than 4000 bytes of alpha numeric data then we are using long datatypes. Long data types store up to 2 GB data.

Syntax: columnname long;

Generally here can be an only one long column for a table and also we are not allowed to create primary key for long datatype column.

Example: SQL> create table test5 (column1 long, column2 long);

3) **Date:** It is used to store dates in oracle date format.

Syntax: columnname date;

In oracle by default date format is ***DD- MON-YY***

a) Create
b) Alter
c) Drop
d) Truncate
e) Rename (oracle 9i)

a) **Create:** It is used to create database objects like tables, views, sequences, indexes.

Creating a table:

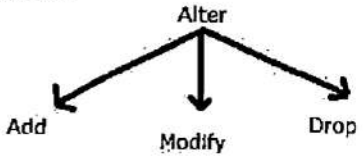
Syntax: create table tablename (colname1 datatype(size), colname2 datatype(size), ...);

Example: SQL> create table first (sno number(10), name varchar2(10), rollno number(10));
To view structure of the table:

Syntax: desc tablename;

Example: SQL> desc first;

b) **Alter:** It is used to change structure of the existing table.



Add: It is used to add number of columns into the existing table.

Syntax: alter table tablename add (colname1 datatype(size), colname2 datatype(size), ...);

Example: SQL> alter table first add sal number(10);
SQL> desc table;

Modify: It is used to change column datatype or column datatype size only.

Syntax: alter table tablename modify (colname1 datatype(size), colname2 datatype(size), ...);

Example: SQL> alter table first modify sno number(10);

Alter...Drop: It is used to remove columns from the existing table.

Method1: If we want to drop a single column at a time without using parenthesis "()" then we are using following syntax:

Syntax: alter table tablename drop column columnname;

Example
SQL> alter table first drop column sal;
SQL > desc first;

Method2: If we want to drop single or multiple columns with using parenthesis then we are using following syntax.

Syntax: alter table tablename drop (colname1, colname2, colname3...);

Example: SQL> alter table first drop (name, rollno);

Important:

- SQL> alter table first drop column sno;
  Error: cannot drop all columns in a table.

Note: *In all database systems we cannot drop all columns in a table.*

c) **Drop:** It is used to remove database objects from database. In all relational database we are allow to drop only one database object at a time.

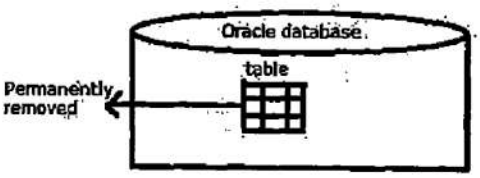
Syntax: drop objecttype objectname;

- ✓ Drop table tablename;
- ✓ Drop view viewname;

Dropping a Table

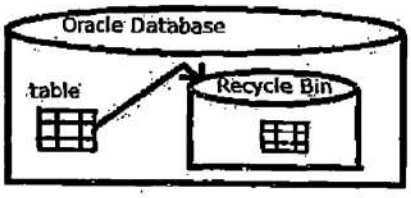
Before Oracle 10g:

Syntax: drop table tablename;



Oracle 10g, 11g, 12c (Enterprise Edition)

Syntax: drop table tablename;



Get it back from recyclebin:

Syntax: flashback table tablename to before drop;

To drop table permanently:

Syntax: drop table tablename purge;

Example: (dropping a table -> oracle 10g Enterprise Edition)

SQL> drop table first;
Table dropped.

Get it back the table from recyclebin

SQL> flashback table first to before drop;

Testing

Method 1

SQL> desc first;
Error: Object first does not exist.

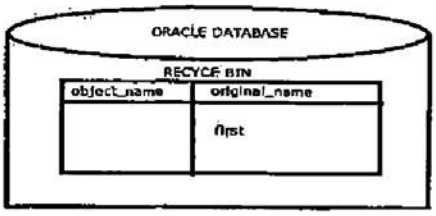
Method 2

SQL> flashback table first to before drop;
Error: object not in Recyclebin.

Oracle 10g enterprise edition introduced recyclebin which is used to store dropped tables. It is also same as window recyclebin. If we want to view recyclebin then we are using following syntax.

Syntax: show recyclebin;

Oracle recyclebin is same as windows recyclebin that's why we can also remove single table or we can also remove all tables at a time from recyclebin by using purge comment.
Recyclebin is a read only table whenever we are installing oracle then automatically so many read only tables are created. These read only tables are also called as "Data Dictionaries".



In oracle, we can also drop tables from recyclebin using "purge" command.

To Drop particular table from Recyclebin:

Syntax: purge table tablename;

Example

```
SQL> create table first (sno number (10));
SQL> drop table first;
SQL> desc recyclebin;
SQL> select original_name from recyclebin;

        Original_name
        -------------
            First

SQL> purge table first;
```

Testing

```
SQL> select original_name from recyclebin;
No rows selected;
```

To Drop all tables from Recyclebin:

Syntax: purge recyclebin;

Example

```
SQL> create table first (sno number (10));
SQL> create table second (sno number (10));
SQL> drop table first;
SQL> drop table second;
SQL> desc recyclebin;
SQL> select original_name from recyclebin;

        Original_name
        -------------
            First
            Second
```

```
SQL> purge recyclebin;
```

Testing

```
SQL> select original_name from recyclebin;
        No rows selected
```

Note: In oracle through recyclebin we can get it back dropped table but we cannot get it back dropped column.

d) **Truncate:** Oracle 7.0 introduced truncate command, whenever we are using "truncate" command total data permanently deleted from table.

Syntax: truncate table tablename;

Example

```
SQL>create table first as select * from emp;
SQL> select * from first;
SQL> truncate table first;
```

Testing

```
SQL> select * from first;
      No rows selected.
SQL> desc first;
```

e) **Rename:** It is used to rename a table and renaming a column also.

Renaming a Table

Syntax: rename oldtablename to newtablename;

Example: rename emp to emp1;

Renaming a Column: (Oracle 9i)

Syntax: alter table <table name> rename column <old column name> to <new column name>;

Example

```
SQL> alter table emp rename column empno to sno;
SQL> select * from emp;
```

Note: *In all database systems by default all **DDL** commands are automatically committed.*

a) **Insert**
b) **Update**
c) **Delete**
d) **Merge (oracle 9i)**

These commands are used to manipulate data in a table.

a) **Insert:** It is used to insert data into a table. Oracle provided three methods for inserting data in to a table.

Method1:

Syntax: *Insert into tablename values (value1, value2, value3 ...);*

Example

```
SQL> create table first (sno number(10), name varchar(20));
```

Inserting data into table

```
SQL> insert into first values (1, 'abc');
SQL> insert into first values (2, 'sachin');
SQL> select * from first;

      Sno        name
     -----      -------
       1          abc
       2         sachin
```

Method2: (Using Substitutional Operator '&') [& = Enter values for]

Syntax: insert into tablename values (&col1, &col2, &col3 ...);

Example

```
SQL> insert into first values (&sno, '&name');
     Enter value for sno: 3
     Enter value for name: xyz

SQL> /     [This gives the previous commands]

     Enter value for sno: 4
     Enter value for name: sachin

SQL> select * from first;
```

Method3: (Skipping Columns)

Syntax: insert into tablename (colname1, colname2...) values (value1, value2 ...);

Example

```
SQL> insert into first (name) values ('www');
SQL>select * from first;
```

b) **Update:** It is used to change data within a table.

Syntax: update tablename set columnname = newvalue where columnname = oldvalue;

Example

```
SQL> update emp set sal = 1000 where ename = 'SMITH';
SQL> select * from first;
```

Note: *In all database if we want to insert data into particular cell then we must use update command.*

Example

```
SQL> alter table first add address varchar2 (10);
SQL> update first set address = 'mumbai' where name = 'sachin';
SQL> select * from first;
```

SNO	NAME	ADDRESS
1	abc	
2	sachin	mumbai
3	xyz	
4	zzz	

Example

```
SQL> update first set address = null where name = 'sachin';
SQL> select * from first;
```

c) **Delete:** It is used to delete particular rows or all rows from a table.

Syntax: Delete from tablename; *[delete all rows from table]*

(Or)

Syntax: Delete from tablename where condition; *[delete particular rows from table]*

Example

```
SQL> delete from first;
     Rows deleted.
```

Get it back deleted data

```
SQL> rollback;
SQL> select * from first;
```

Difference between Delete and Truncate

Whenever we are using 'delete from tablename' (or) 'truncate table tablename' then automatically total data is deleted.

Whenever we are using 'delete from tablename' then deleted data automatically stored in a buffer. We can also get it back, this deleted data by using "rollback" command.

Whenever we are using 'truncate table tablename' then total data permanently deleted from a table because truncate is a DDL command. That's why we cannot get it back this data using "rollback".

Oracle 9i introduced "Merge" statement. Merge is a DML command which is used to transfer data from **"source table"** into **"target table"** when table structure are same.

Generally, merge statement used in data warehousing applications. In merge statement we are using **"update"**, **"insert"** command. This command is also called as **"UPSERT"** command (UP -> **Update**, SERT-> **Insert**).

Syntax:

```
merge into <target table name>
using <source table name>
on (join condition)
when matched then
update set TargetTableColumn1 = SourceTableColumn1, .....
when not matched then
insert (target table column names) values (source table column names);
```

Example

```
SQL> select * from dept; [target table]
SQL> create table depts as select * from dept;
SQL> insert into depts values(1, 'a', 'b');
SQL> select * from depts;

SQL> merge into dept d
     using depts s
     on (d.deptno=s.deptno)
     when matched then
     update set d.dname = s.dname, d.loc = s.loc
     when not matched then
     insert (d.deptno, d.dname, d.loc) values (s.deptno, s.dname, s.loc) ;

     5 rows merged
SQL> select * from dept;
```

Note: Through "Merge" statement we cannot update "ON" clause columns.

- **SELECT**

In all relational databases we can retrieve data from table using select statement.

Syntax: select columnname1, columnname2... from tablename where condition
group by columnname having condition order by columnname [asc/desc];

1. Select all cols and all rows
2. Select all cols and particular rows
3. Select particular cols and all rows
4. Select particular cols and particular rows.

Creating a new table from another table / copying a table from another table.

Syntax: create table <new table name> as select * from <existing table name>;

Example

```
SQL> create table test as select * from emp;
SQL> select * from test;
```

Note: *In all database systems whenever we are copying a table from another table constraints are never copied. (Primary key, foreign key...).*

Creating a new table from existing table without copying data:

Syntax: create table <new table name> as select * from <existing table name> where <false condition>;

Example: SQL> create table test1 as select * from emp where 1=2;

Testing:
```
SQL> select * from test1;
No rows selected.
SQL> desc test1;
```

Operators Used In "Select" Statement

1. Arithmetic Operator (+,-,*,/)
2. Relational Operator (=, <, <=, >, >=, [<> or!=] not equal)
3. Logical Operator (AND,OR,NOT)
4. Special Operator

1. **Arithmetic Operator**

Arithmetic operators are used in number datatype column.
Arithmetic operator is used for "select" (eg: select column1, column2 ...)
Relational, Logical and Special operators are used for "where" (eg: where condition)
We can also use arithmetic operators in "where" conditions.

- ***Write a query to display ename, sal, annsal from emp table?***

Ans: select ename, sal, sal*12 annsal from emp;

ENAME	SAL	ANNSAL
SMITH	700	8400

- ***Write a query to display the employees who are getting more than 30000 annual salaries from emp table?***

Ans: SQL> select ename, sal, sal*12 annsal from EMP where annsal>30000;
Error: ORA-00904: "ANNSAL": invalid identifier

Ans: SQL> select ename, sal, sal*12 annsal from EMP where sal*12>30000;

Note: *In oracle we are not allow to use column alias name in where clause.*

2. **Relational Operator**

Relational operator are allow to use in where clause only.

- ***Write a query to display the employees except job as clerk from EMP table?***

Ans: select * from emp where job<>'CLERK'; {or job != 'CLERK'}

- ***Write a query to display the employees who are getting more than 2000 salary from EMP table?***

Ans: select * from emp where sal>2000;

3. **Logical Operator**

In all relational databases if we want to define more than one condition within where clause then we are using either AND (or) OR logical operator.

- **AND Operator:** AND operator display a record if both 1st and 2nd condition is true.

Syntax: select * from tablename where condition1 and condition2;

- ***Write a query to display those clerks having more than 1500 salary from EMP table?***

Ans: select * from emp where job= 'CLERK' and sal>1500;

- **OR operator:** OR operator display a record either 1st or 2nd condition is true.

Syntax: select * from tablename where condition1 or condition2;

- **Write a query to display whose jobs are clerk either employee is getting more than 1500 salary from emp table.**

Ans: select * from emp where job= 'CLERK' or sal>1500;

Note: *In databases if we want to retrieve multiple values within a single column then we must use "OR" operator.*

Example: select * from emp where job= 'CLERK' or job='SALESMAN';

- ***Write a query to display the employees who are belongs to the department numbers 20,50,70,90?***

Ans: select * from emp where deptno=20 or deptno=50 or deptno=70 or deptno=90;

4. Special Operators

a) In	Not in
b) Between	Not between
c) Is null	Is not null
d) Like	Not like

A. **In Operator:** It is used to pick the values one by one from list of values. If we want to retrieve multiple values in a single column then we are using 'IN' operators in place of 'OR' operator, because 'IN' operator performance is very high as compared to 'OR' operator.

Syntax: select * from <table name> where <column name> in (list of values);

Columnname belongs to any data type is possible.

Example

```
SQL> select * from emp where deptno in (20, 50, 70, 90);
SQL> select * from emp where ename in ('SMITH', 'FORD');
```

Note: In all database systems whenever we are using multiple row subqueries then we must use "in" operator.

Example: select * from emp where deptno not in (10, 20, null);

Note: In all relational databases "not in" operator doesn't work with "null" values.

Note: Whenever we are using "IN" operator internally oracle server uses logical operator "OR" that's why "IN" operator works with null values. Whereas whenever we are using "NOT IN" operator then oracle server internally uses logical operator "AND" that's why "NOT IN" operator does not work with null values.

- **Null Operator:** Null is an undefined, unavailable, unknown value. It is not same as "zero". Any arithmetic operations performed on null values again it becomes "null".

  Example: null+50=> null.

- ***Write a query to display ename, sal, comm, sal+comm of the employee SMITH from EMP table?***

Ans: select ename, sal, comm, sal+comm from emp where ename= 'SMITH';

ENAME	SAL	COMM	SAL+COMM
SMITH	1100	null	null

To overcome this problem oracle provided "NVL()" function.

- **NVL():** NVL function is a predefined function which is used to replace or substitute user defined value in place of "null". NVL() always accepts two parameters.

Syntax: NVL(exp1, exp2);

Here exp1, exp2 must belong to same datatype. If exp1 is null then it returns exp2. Otherwise it returns exp1.

Example

1) NVL(null, 30) -> 30.
2) NVL(10, 20) -> 10.

Solution

```
SQL> select ename, sal, comm, sal+NVL(comm, 0) from emp where ename= 'SMITH';

ENAME      SAL     COMM    SAL+COMM
--------   -------  -------  -----------
SMITH      1100     Null     1100
```

- Sal+ nvl(comm,0)
- 1100 + nvl(1100,0)
- 1100 + 0
- 1100

- **NVL2():** Oracle 9i introduced NVL2 () function. This function accepts three parameters.

Syntax: NVL2 (exp1, exp2, exp3);

Here if exp1 is null, then it returns exp3. Otherwise it returns exp2.

Example

SQL> select nvl2 (null, 10, 20) from dual;
Output: 20

SQL> select nvl2 (30, 40, 50) from dual;
Output: 40

- **Update employee commission as follows using nvl2() from emp table;**

1) If comm is null then update comm -> 500
2) If comm is not null then update comm-> 500

Ans: SQL> update emp set comm=nvl2 (comm, comm+500,500);
SQL> select * from emp;

B. **Between:** This operator is used to retrieve range of values from a table column. This operator is also called as Between..... And operator.

Syntax: select * from <table name> where <column name> between <low value> and <high value>;

Example: select * from emp where sal between 2000 and 5000;

C. **Is null, is not null:** These two operators used in where condition only. These two operators are used to test whether a column having null values or not.

Syntax: select * from tablename where columnname is null;

Syntax: select * from tablename where columnname is not null;

- **Write a query to display the employees who are not getting commission from emp table.**

Ans: select * from emp where comm is null;

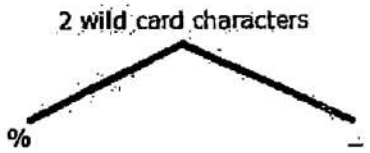
- **Write a query to display the employees who are getting commission from emp table?**

Ans: select * from emp where comm is not null;

D. **Like Operator:** It is used to search strings based on character pattern. "Like" operator performance is very high compare to all other searching functions in oracle database. In all databases along with like operator ANSI/ISOSQL provide two special characters.

1. **% (Percentile)**
2. **_ (Under score)**

These 2 special characters are called as wild card characters. These wild card characters have special meaning.



% (Percentile):- Group of characters if you want to match (or) string

_ (Under score):- Single character to match.

Syntax: select * from <table name> where <column name> like 'character pattern';

- ***Write a query to display the employees whose ename start with "M" from EMP table using like operator?***

Ans: select * from emp where ename like 'M%';
MARTIN
MILLER

- ***Write a query to display the employees where ename second letter will be 'L' from EMP table using 'like' operator?***

Ans: select * from emp where ename like '_L%';
ALLEN
CLARK
BLAKE

- ***Write a query to display the employees who are joining in the month December from the EMP table using like operator.***

Ans: select * from emp where hiredate like '_ _ _D%'; (or) '%DEC%';

- ***Write a query to display the employees who are joining in the year "81" from EMP table using like operator?***

Ans: select * from emp where hiredate like '%81';

Whenever wild card characters available in character data then we are trying to retrieve the data using like operator then database servers treated these wild card characters as special characters.

To overcome this problem ANSI/ISOSQL provided a special function "escape" along with **'like'** operator which is used to escape special characters and also this function treated as _ , % as same meaning as _ , %.

Syntax: select * from <table name> where <col name> like 'character pattern' escape 'escape character';

Note: *Escape character must be a single character having length '1' within character pattern we must use escape character before wild card character.*

Example

```
SQL> insert into emp (empno, ename) values (1, 'S_ MITH');
SQL> select * from emp;
```

- ***Write a query to display the employees whose ename start with 'S_' from EMP table using 'like' operator?***

Ans: select * from emp where ename like 'S_%';

ENAME
SMITH
SCOTT
S_MITH

Solution

```
SQL> select * from emp where ename like 'S? _%' escape '?'
Output: S_MITH
```

Example

```
SQL>insert into emp(empno, ename) values (2, 'S__MITH');
SQL> select * from emp;
```

Solution

```
SQL> select * from emp where ename like 'S? _? _%' escape '?';
Output: S__MITH
```

Concatenation Operator [|| double pipe]

It is not a special operator rarely used in "SQL" and regularly used in "PL/SQL". If we want to display column data along with literal strings then we must use concatenation operator.

Example

```
SQL>select 'My employee names is'|| ename from emp;
Output: My employee name isSMITH
```

If we want to display our own space in between the columns then also we can use concatenation operator.

```
SQL>select ename || '   '|| sal from emp;
```

Example

```
SQL> select 'My employee name is'||' '|| ename from emp;
Output: My employee name is SMITH
```

Functions are used to solve particular task and also functions must return a value. Oracle has two types of functions.

1. **Predefined functions**
2. **User defined functions**

1) **Predefined Functions:** there are 4 types.
   a) Number function
   b) Character function
   c) Date function
   d) Group function (or) Aggregate function

A. **Number function:** These functions operate over "number" data.

a. **Abs():** It is used to convert negative values into positive values.

Example

SQL> select abs (-50) from dual;
Output: 50

- **Dual ():** Dual is a predefined virtual table which contains only one row and one column, Dual table is used to test predefined, user defined functions functionality.

Example for testing predefined functions

SQL> select nvl (null, 30) from dual;
Output: 30

SQL> select nvl (20, 30) from dual;
Output: 20

SQL> select * from dual;

Note: *In oracle by default dual table column data type is varchar2().*

Example: select * from dual;

Generally, this table is also used to perform mathematical operations.

Example: SQL> select 10+50 from dual;
Output: 60

Example: SQL> select ename, comm, sal, abs (comm-sal) from emp where comm is not null;

b. **Mod (m, n): It** will give remainder after 'm' divided by 'n'.

Example

SQL> select mod (10, 5) from dual;
Output: 0

c. **Round (m, n):** It rounds given floated valued number 'm' based on 'n'.

Example

SQL> select round (1.8) from dual;
Output: 2
SQL> select round (1.23456, 3) from dual;
Output: 1.235

Note: *Round always checks remaining number if remaining number are above 50% then 1 automatically added to the rounded number.*

Example

SQL> select round (1285.456,-1) from dual;
Output: 1290

Execution

Step1: 1280 (here 5 replace with 0 and 5 is above 50% out of 10)

Step2: 1280
+1

1290

Example: SQL> select round (1285,-2) from dual;
Output: 1300

Execution

Step1: 1200 (here 85 replace with 00 and 85 is above 50% out of 100)

Step2: 1200
+1

1300

d. **Trunc (m, n):** It truncates given floating values 'm' based on 'n'. This function doesn't check remaining number is above 50% or below 50%.

Example

SQL>select trunc (1.8) from dual; o/p: 1
SQL> select trunc (1.23456, 3) from dual; o/p: 1.234

e. **Greatest(exp1,exp2,.... expn), Least(exp1,exp2,.... expn):**

Greatest returns maximum value among given expressions. Where as Least returns minimum value among given expressions.

Example

SQL> select greatest (3, 5, 8, 9) from dual;
Output: 9

SQL> select ename, sal, comm, greatest (sal, comm) from emp where comm is not null;
SQL> select max (sal) from emp;
Output: 6600

f. **Ceil() and Floor():** These two functions always return integer.
Ceil() returns nearest greatest integer where as floor() returns nearest lowest integer.

Example

SQL> select ceil (1.3) from dual;
Output: 2

Example

SQL> select floor (1.9) from dual;
Output: 1

g. **Rollup, Cube:** Oracle 8i introduced rollup, cube clause, these are optional clauses used along with group by clause only. These clauses are used to calculate subtotal, grant total automatically.

Syntax: Select colname1, colname2 ... from <table name> group by rollup (col1, col2 ...);
Syntax: Select colname1, colname2 ... from <table name> group by cube (col1, col2 ...);

Whenever GROUP BY clause having only 1 column then ROLLUP, CUBE clauses return same results, in this case these clauses return grant total.

Example

```
SQL>select deptno, sum(sal) from emp group by rollup (deptno);
```

DEPTNO	SUM(SAL)
10	11750
20	13425
30	10900
	36075

```
SQL>select deptno, sum(sal) from emp group by cube (deptno);
```

DEPTNO	SUM(SAL)
10	11750
20	13425
30	10900
	36075

Generally ROLLUP is used to calculate subtotal, grant total automatically used on a single column at a time, whereas if you want to calculate subtotal, grant total based on all grouping columns then we must use CUBE.

Example 1

```
SQL> select deptno, job, sum(sal) from emp group by rollup (deptno, job);

SQL> select deptno, job, sum(sal) from emp group by cube (deptno, job);
```

Example 2

```
SQL> select deptno, job, sum (sal), count (*) from emp group by cube (deptno, job) order by 1, 2;
```

a) ***Upper*(): It is used to convert a string or column values into "upper case".**

Example

SQL>select upper ('abc') from dual;
Output: ABC

For column: SQL> select upper ('ename') from dual;

- ***Dual*: In oracle dual is a pre-defined virtual table which contents one row and one column. Dual table is used to test pre-defined, user defined function and functionality.**

Example: Testing pre defined function

Nvl (exp1, exp2)
SQL> select nvl (null, 20) from dual;
Output: 20
SQL> select nvl (10, 20) from dual;
Output: 10

Whenever we are installing oracle server automatically create dual table within "***SYS***" user, that's why we cannot perform DML operation within dual table, because "***SYS***" user table are read only table but we can use dual table in all other users in oracle database.

Note: *By default dual table column datatype is varchar2()*

Example

*SQL> select * from dual;*
SQL> desc dual;

Note: *By using dual table we can also perform mathematical operation.*

Example

SQL > select 30 + 20 from dual;
Output: 50

SQL > select upper(ename) from emp;

b) ***Lower*(): It is used to convert string column values into lower case.**

Example

SQL>select lower (ename) from emp;
Updating or modifying data within table:
SQL> update emp set ename=lower (ename);

c) ***Initcap*(): It is used to convert first letter is capital and all remaining letters are small.**

Example

SQL> select initcap (ename) from emp;

SQL> select initcap ('ab cd ef') from dual;
Output: Ab Cd Ef

d) **Length():** It returns number datatype that is it returns total length of the string including spaces.

Example

SQL>select length ('AB_CD') from dual;
Output: 5

- **Substr():** It will extract portion of the string within given string based on last two parameters.

Example

SQL> select substr ('ABCDEF', 2, 3) from dual;
Output: BCD
SQL> select substr ('ABCDEF',-2, 3) from dual;
Output: EF
SQL> select substr ('ABCDEF',-4) from dual;
Output: CDEF

Syntax: substr (columnname (or) 'string name', starting position, no. of characters from position);

Starting position-> can be ***+ve*** or ***–ve***
Starting position and no. of characters position –> must be numbers

- **Write a query to display the employees whose ename second letter would be "LA" from emp table using substring function?**

Ans: select * from emp where substr (ename, 2, 2) = 'LA';
BLAKE
CLARK

Note: *In all relational database systems* *we are not allowed to use "GROUP" functions in "WHERE" clause* *but we are allowed to use number functions, character functions, date functions in "WHERE" clause.*

Example

SQL> select * from emp where sal=max (sal);
Error: group function is not allowed here.

- **Write a query to display the employee whose employee's length is 5 from EMP table.**

Ans: select * from emp where length (ename) =5;

- **Instr():** Instr always returns number datatype i.e. it returns position of the delimiter, position of the character, position of the string within given string.

Example

SQL> select instr ('ABC*D', '*') from dual;
Output: 4
SQL> select instr ('ABCDEFGHCDKMNHCDJL', 'CD',-5, 2) from dual;
Output: 3
SQL> select instr ('ABCDEFGHCDKMNHCDJL', 'CD',-4, 2) from dual;
Output: 9

Syntax: Instr (colname/ 'string name', 'str', searching position, no. of occurrences from searching position);

Searching position -> +ve (or) –ve
Searching and no. of occurrences -> must be numbers.

Always in string returns position based on last two parameters but oracle server counts no. of characters left side first position onwards.

- ***Lpad()***: It will fill remaining spaces with specified character on the left side of the given string. Here always second parameter returns total length of the string and also third parameter is an optional.

Syntax: Lpad (columnname (or) 'string name', total length, 'filled character');

Total length -> number

Example

```
SQL> select lpad ('ABCD', 10,'#') from dual;
Output: ######ABCD

SQL> select lpad ('ABC', 5) from dual;
Output: (space) (space) ABC

SQL> select lpad ('ABC', 2, '*') from dual;
Output: AB
```

- ***Rpad()***: It will fill remaining spaces with specified character on the right side of the given string.

Example

```
SQL> select rpad ('ABCD', 10, '#') from dual;
Output: ABCD######

SQL> select rpad (ENAME, 50, '-') ||sal from emp;

SQL> select lpad ('ABC', 2, '*') from dual;
Output: AB

SQL> select rpad ('ABC', 2, '*') from dual;
Output: AB
```

Note : *Whenever we are passing more no. of characters then the total length specified in second parameter then LPAD() & RPAD() returns same output because when we are submitting LPAD(), RPAD() into oracle server then oracle server always execute left side 1st character onwards within 1st parameter.*

- ***Ltrim***: It removes specified character on the left side of the given string.

Syntax: Ltrim (column name (or) 'string name', {set of characters});

Example

```
SQL> select ltrim ('SSMISSTHSS', 'S') from dual;
Output: MISSTHSS

SQL> select job, ltrim (job, 'CSM') from emp;

JOB          LTRIM (JOB)
---------    -----------
CLERK        LERK
SALESMAN     ALESMAN
MANAGER      ANAGER
```

- ***Rtrim***: It is used to remove specified character on the right side of the given string.

Example

```
SQL> select rtrim ('SSMISSTHSS', 'S') from dual;
Output: SSMISSTH
```

- ***Trim()***: Oracle 8i introduced trim function it is used to remove left and right side specified characters and also it is used to remove LEADING and TRAILING spaces.

Syntax: trim ('character' from 'string name');

Example

SQL> select trim ('S' from 'SSMISSTHSS') from dual;
Output: MISSTH

Note: *In oracle we can also convert TRIM function into LTRIM by using LEADING clause and also convert TRIM function to RTRIM by using TRAILING clause.*

SQL>select trim (leading 'S' from 'SSMISSTHSS') from dual;
Output: MISSTHSS
SQL>select trim (trailing 'S' from 'SSMISSTHSS') from dual;
Output: SSMISSTH

Note: *Using TRIM function we can also remove LEADING, TRAILING spaces.*

SQL>select length (trim (' welcome ')) from dual;
Output: length=7 [trim removed free space]

- ***Translate(), Replace():*** Translate is used to replaces character by character where as replace is used to replaces character by string (or) string by string.

Example: SQL>select translate ('india', 'in', 'xy'), replace ('india', 'in', 'xy') from dual;

Translate	Replace
xydxa	xydia

Syntax: translate ('str1', 'str2', 'str3'....);

Example

SQL>select translate ('ABCDEF', 'FEDCBA', 123456);
Output: 654321
SQL>select replace ('A B C', ' ', 'India') from dual;
Output: AindiaBindiaC
SQL>select job, replace (JOB, 'SALESMAN', 'MARKETING') from emp;

SQL>select replace ('SSMISSTHSS', 'S') from dual;
Output: MITH

Note: *Whenever we are not using 3rd parameter in REPLACE() then automatically 2nd parameter specified character permanently removed from the given string.*

If you want to count number of times particular character occurs within a given string then also we are using REPLACE() along with LENGTH().

- **Write a query to count number of times that particular 'I' occurred within given string 'india' using replace function?**

Ans: SQL> select length('India') - length(replace('india', 'i')) from dual;
Output: 2

- **Concat(str1,str2):** It is used to concatenate given two strings.

Example

SQL>select concat ('wel', 'come') from dual;
Output: Welcome

Date Arithmetic

1. Date + number
2. Date – number
3. Date1 + date2 *(It would not work)
4. Date1 – date2

Example

SQL> select sysdate+1 from dual;
Output: 02-MAY-19

SQL> select sysdate-1 from dual;
Output: 30-APR-19

SQL> select sysdate-sysdate from dual;
Output: 0

- **Write a query to display first date of the current month using predefined date functions sysdate, add_months(), last_date() functions.**

Ans: SQL> select last_day (add_months (sysdate,-1)) +1 from dual;

(or)

SQL> select add_months (last_day (sysdate),-1) +1 from dual;

Output: 01-MAY-19

Date Conversion Functions

Oracle provided two date conversion function. These are:

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

1) **To_Char():** It is used to convert date type into character type i.e. it converts date type into date.

Example

SQL> select to_char(sysdate, 'dd/mm/yy') from dual;
Output: 02/05/19

Note: Basically "to_char" character formats are case sensitive formats.

Example

SQL> select to_char (sysdate, 'DAY') from dual;
Output: THURSDAY

DAY-> THURSDAY (Upper case)
day -> thursday (Lower case)
Day-> Thursday (Sentence case)

Returns Number As Output	*Returns Character As Output*
D	DAY
DD	DY
DDD	MONTH
YY	MON
YYYY	YEAR
HH	
MI	
SS	

Example

SQL> select to_char(sysdate, 'DY')from dual;
Output: THU

Example

SQL> select to_char(sysdate, 'D')from dual;
Output: 5

D-> day of the week (sun-1, mon-2, tue-3, wed-4, thu-5, fri-6, sat-7)

DD-> day of the month (01,02,03,04,05,06,07,08,09,10,11,12)

DDD-> day of the year (tells how many days completed) (01, 02,... 365)

DDTH-> used as date with 'rd' or 'th' (eg: 02nd, 19th, 23rd)

Example

SQL> select to_char(sysdate, 'DDSPTH') from dual;
Output: TWELFTH

SP-> Spell out

Example

SQL> select to_char(sysdate, 'HH:MI:SS') from dual;
Output: 05:12:42

SQL> select to_char(sysdate, 'HH24:MI:SS') from dual;
Output: 17:12:42

By default 12-hours format, to convert into 24-hours format we have to **"HH24: MI: SS"** (24-hours format)

SQL> select to_char('15-JUN-05', '15-JUNE-05') from dual;
Error: invalid number

Note: *Whenever we are using TO_CHAR() always first parameter must be "oracle date type", otherwise oracle server returns an error.*

Returns Number As Output	*Returns Character As Output*
D	DAY
DD	DY
DDD	MONTH
YY	MON
YYYY	YEAR
HH	
MI	
SS	

Example

SQL> select to_char(sysdate, 'DY')from dual;
Output: THU

Example

SQL> select to_char(sysdate, 'D')from dual;
Output: 5

D-> day of the week (sun-1, mon-2, tue-3, wed-4, thu-5, fri-6, sat-7)

DD-> day of the month (01,02,03,04,05,06,07,08,09,10,11,12)

DDD-> day of the year (tells how many days completed) (01, 02,... 365)

DDTH-> used as date with 'rd' or 'th' (eg: 02nd, 19th, 23rd)

Example

SQL> select to_char(sysdate, 'DDSPTH') from dual;
Output: TWELFTH

SP-> Spell out

Example

SQL> select to_char(sysdate, 'HH:MI:SS') from dual;
Output: 05:12:42

SQL> select to_char(sysdate, 'HH24:MI:SS') from dual;
Output: 17:12:42

By default 12-hours format, to convert into 24-hours format we have to "HH24: MI: SS" (24-hours format)

SQL> select to_char('15-JUN-05', '15-JUNE-05') from dual;
Error: invalid number

Note: *Whenever we are using TO_CHAR() always first parameter must be "oracle date type", otherwise oracle server returns an error.*

2) To_Date(): It is used to convert char type into date type i.e. it converts date string into date type (oracle default date format)

Example

SQL> select to_date('24/JUNE/05') from dual;
Output: 24-JUN-05

SQL> select to_date('24/06/05') from dual;
Error: not a valid month

Rule

*Whenever we are using **TO_DATE()** always passed parameter return values must match with the default date format return values, otherwise oracle server returns an error.*

To overcome this problem use a 2nd parameter as same as 1st parameter format then only oracle server automatically converts "date string" into "oracle date type".

Solution

SQL> select to_date('24/06/05', 'DD/MM/YY') from dual;
Output: 24-JUN-05

***Q)* Write a query which add 5 days to the given date 09-feb-05?**

Ans: Example: 1
SQL> select '09-FEB-05'+5 from dual;
Error: Invalid number

Solution
SQL>select to_date('09-FEB-05')+5 from dual;
Output: 14-FEB-05

Example: 2
SQL>select to_date('09-02-05')+5 from dual;
Error: not a valid month

Solution
SQL> select to_date('09-02-05', 'DD-MM-YY')+5 from dual;
Output: 14-FEB-15

- **Write a query to convert given date string into client requirement format using to_char() function? Given date is '15-jun-05 and Expected format '15/JUNE/05'**

Ans: SQL> select to_char('15-june-05', 'DD/MONTH/YY') from dual;
Error: invalid number

Rule

In oracle whenever we are using TO_CHAR() always 1st parameter must be oracle date type, otherwise oracle returns an error. To overcome this problem we must used TO_DATE() within 1st parameter of the TO_CHAR().

Solution

SQL> select to_char(to_date('15-jun-05'), 'DD/MONTH/YY') from dual;
Output: 15/JUNE /05

Whenever we are using **TO_CHAR()** month, day formats then oracle server internally automatically allocate 9 bytes of memory. Whenever we are passed month having less than 9 characters then oracle server return ***"spaces"*** in place of remaining bytes.

To overcome this problem oracle provided **Fill Mode (FM)** format which separates blank spaces but also separates leading zeros (0). This mode is used in second parameter of the **to_char()**.

Example

```
SQL> select to_char (to_date('15-JUN-05'), 'DD-MON-YY') from dual;
Output: 15-JUN-05
SQL> select to_char (to_date('15-JUN-05'), 'DD/FMMONTH/YY') from dual;
Output: 15/JUNE/5
```

- **Write a query to display the employees who are joining in the month December from "emp" table using to_char()?**

Ans:

```
SQL> select * from emp where to_char(HIREDATE, 'MON')= 'DEC';
                    OR
SQL> select * from emp where to_char(HIREDATE, 'MM')=12;
                    OR
SQL> select * from emp where to_char(HIREDATE, 'FMMONTH')='DECEMBER';
```

- **Write a query to display the employees who are joining in the year '81' from emp table by using to_char()?**

Ans:

```
SQL> select * from emp where to_char(hiredate, 'YY') = 81;
SQL> select hiredate, to_char(hiredate, 'YYYY') from emp;
```

Rule:

In Oracle, whenever we are passing date string into oracle predefined date functions (ADD_MONTHS, NEXT_DAY, MONTHS_BETWEEN) then we are not required to use TO_DATE() because here oracle server uses implicit conversion but here passed parameter return values must match with default date format return values, otherwise oracle server returns an error.

Example: 1

Automatic Conversion

```
SQL> select last_day('12-JUN-05') from dual;
Output: 30-JUN-05
```

But here passed parameter format must be default date format otherwise oracle server returns an error. To overcome this problem we must use **to_date()**.

Example: 2

Explicit Conversion

```
SQL> select last_day('12-06-05') from dual;
Error: invalid month
```

Solution

```
SQL> select last_day(to_date('12-06-05', 'DD-MM-YY')) from dual;
Output: 30-JUN-05
```

Inserting dates into oracle table:

Syntax: columnname date;

Example

```
SQL> create table test(col1 date);
SQL> insert into test values(sysdate);
SQL> select * from test;
        COL1
      ---------
      24-MAR-19

SQL> insert into test values ('15-aug-05');
SQL> select * from test;
        COL1
      ---------
      24-MAR-19
      15-AUG-05

SQL> insert into test values ('15-08-05');
Error: not a valid month

SQL> insert into test values (to_date ('15-08-05', 'DD-MM-YY'));
1 row inserted
SQL> select * from test;
        COL1
      ---------
      24-MAR-19
      15-AUG-05
      15-AUG-05
```

Rule:

*In Oracle whenever we are inserting dates into DATE datatype column then oracle server automatically converts DATE string into DATE type when inserted date string is in oracle date format. Otherwise oracle server returns an error. To overcome this problem also then we must use **TO_DATE()**.*

In oracle database date datatype has two parts. These are...

I. **Date Part**
II. **Time Part / Time Portion**

Example

```
SQL> select to_char(sysdate, 'DD-MON-YYYY  HH24:MI:SS') from dual;
Outpt: 02-MAY-19     17:18:07
```

In Oracle whenever we are using the ROUND() (or) TRUNC() then oracle server automatically changes in date part based on time portion and also time portion is automatically set to zeroes (0).

Whenever we are using ROUND() then oracle server automatically adds one day to the given date, if time portion is greater than or equal to 12 noon (>=12) then time portion automatically set it to "zero" (0).

When applying round()

```
SQL> select to_char(round(sysdate), 'DD-MON-YYYY  HH24:MI:SS') from dual;
Output: 03-MAY-2019    00:00:00
```

When applying trunc()

Whenever we are using **trunc()** oracle server automatically returns same date if time portion is greater than or equal to 12 noon (>=12) also here time portion is automatically set to "zero"(0).

Example

```
SQL> select to_char(trunc(sysdate), 'DD-MON-YYYY HH24:MI:SS') from dual;
Output: 02-MAY-2019    00:00:00
```

Write a query to display the employees who are joining today from emp table?

Ans:
```
SQL> Insert into emp(empno, ename, hiredate) values(1, 'SHAILENDRA',02-MAY-19);
SQL> select * from emp where hiredate=sysdate;
Error: no row selected
```

In oracle whenever we are comparing DATE in WHERE clause by equality operator then oracle server not only compares date parts but also compares time parts, that's why oracle server doesn't return proper result when we are comparing date.
To overcome from this problem we must set time portion is zeroes (0) by using trunc().

Solution

```
SQL> select * from emp where trunc(hiredate) = trunc(sysdate);

EMPNO     ENAME          HIREDATE
-------   ----------     ----------
  1       SHAILENDRA     02-MAY-19
```

- ***Write a query to display the employees who are joining today from emp table by using to_date() function ?***

Ans:
```
SQL> insert into emp(empno,ename,hiredate) values(1, 'Shailendra', sysdate);
SQL> Select * from emp;

SQL> Select ename, to_char(hiredate, 'DD-MON-YY HH24:MI:SS') from dual;

ENAME        TO_CHAR()
----------   -----------------------
Shailendra   02-MAY-19    16:25:19

SQL> Select * from emp where hiredate=sysdate;
Error: No row selected
```

Solution

```
Select * from emp where hiredate=to_date ('02-MAY-19 16:25:19', 'DD-MON-YY HH24:MI:SS');

EMPNO   ENAME       HIREDATE
------  ----------  ---------
1       Shailendra  02-MAY-19    16:25:19
```

Note

*In oracle whenever we are inserting SYSDATE into date datatype column then oracle server internally automatically insert current time of the system. Whereas when we are inserting our own date by "**to_date()**" then oracle server internally inserts default time. In oracle default time is midnight values (12:00:00 PM (or) 00:00:00)*

*In oracle whenever we are inserting time part by using "**to_date()**" without specifying date part then oracle server automatically inserts first date of the current month.*

Note

*In oracle we can also return first date of the year, first date of the month by using **round()** and **trunc()** through following syntax.*

Syntax: 1

```
Round(date, 'year')
Round(date, 'month')
```

Syntax: 2

```
Trunc(date, 'year')
Trunc(date, 'month')
```

Note

Whenever we are rounding year in oracle server checked out given month is above 50% (jan-jun). If it is above 50% then it adds one year and also rounds to the Jan 1.

Example: round ()

SQL> select round (sysdate, 'year') from dual;
Output: 01-JAN-19

SQL> select round (sysdate, 'month') from dual;
Output: 01-MAY-19

SQL>select round (sysdate, 'day') from dual;
Output: 05-MAY-19 (Next Sunday)

Example: trunc ()

SQL>select trunc (sysdate, 'year') from dual;
Output: 01-JAN-19

SQL>select trunc(sysdate, 'month') from dual;
Output: 01-MAY-19

SQL>select trunc(sysdate, 'day') from dual;
Output: 28-APR-19 (Previous Sunday)

In all databases group function (or) aggregate function operates over number of values in a column and returns single value. Oracle server having following group function these are...

1. **max()**
2. **min()**
3. **avg()**
4. **sum()**
5. **count(*)**
6. **count(ColumnName)**

1. **Max():** It returns maximum value from a table column.

Example

SQL> select max(sal) from emp;
Output: 5000

SQL> select max(hiredate) from emp;
Output: 23-MAY-87

SQL> select max(ename) from emp;
Output: WARD (A=1..... Z=26)

2. **Min():** It returns minimum value from a table column.

Example

SQL> select min(sal) from emp;
Output: 800
SQL> select min(hiredate) from emp;
Output: 17-DEC-80

Note: *In all relational database systems we are not allowed to use "GROUP" functions in "WHERE" clause.*

SQL> select * from emp where sal= min(sal);
Error: group function is not allowed here

3. **Avg():** It returns average from number datatype column.

Example

SQL> select avg (sal) from emp;
Output: 2073.21429

SQL> select avg (comm) from emp;
Output: 550

Note: *In oracle by default all group functions ignores null values except COUNT(*). If you want to count null value within group function then we must use NVL() within group function.*

Example

SQL> select avg(nvl(comm,0)) from emp;
Output: 157.142857

4. **Sum():** It returns total from number datatype column.

Example

```
SQL> select sum(sal) from emp;
Output: 29025
```

5. **Count(*):** It counts number of rows in a table (including null values)

Example

```
SQL> select count(*) from emp;
Output: 14
```

6. **Count(ColumnName):** It counts number of not null values within a column.

Example

```
SQL> select count(comm) from emp;
Output: 4
```

Note: *In oracle if you want to count no. of distinct values from a table column then we must use distinct clause within count function.*

Example

```
SQL>select count(distinct(deptno)) from emp;
Output: 3
```

Group by clause is used to arrange similar data items into set of logical groups. Whenever we are using group by clause database server select similar data items from a table column and then reduces number of data item in each group. Group by clause is used in SELECT statement.

Syntax: select columnname, ... from tablename group by columnname;

- ***Write a query to display number of employees in each department from emp table using "group by" clause?***

Ans: select deptno, count (*) from emp group by deptno;

DEPTNO	COUNT(*)
10	3
20	5
30	6

- ***Write a query to display number of employees in each job from emp table using group by clause?***

Ans: select job, count(*) from emp group by job;

JOB	COUNT(*)
CLERK	4
SALESMAN	4
PRESIDENT	1
MANAGER	3
ANALYST	2

- ***Write a query to display deptno, minimum and maximum salary from emp using group by?***

Ans: select deptno, min(sal), max(sal) from emp group by deptno;

DEPTNO	MIN(SAL)	MAX(SAL)
30	1400	3300
20	1400	3500
10	2800	3900

Note: *In all relational database systems we can also use "GROUP BY" clauses without using "GROUP" functions.*

Example: select deptno from emp group by deptno;

Rule: *Other than GROUP function columns specified after select those all columns must be specified after "GROUP BY". Otherwise oracle server returns an error "not a GROUP BY expression".*

Example

SQL> select deptno, sum(sal),job from emp group by deptno;

Error: not a GROUP BY expression

Solution

SQL> select deptno, sum(sal),job from emp group by deptno,job;

SQL> select deptno, job from emp group by deptno, job;

Note: *In all database after GROUP BY clause all specified column not required to specified after SELECT.*

Example: select deptno from emp group by deptno, job, sal;

Whenever we are submitting "GROUP BY" clauses queries then oracle server first select specified column from "GROUP BY" clause and then only oracle server arrange similar data items from that column into set of logical group and then only this data internally storing results and table. Then only oracle server selects specified columns in select list from this result set table.

Example: select deptno from emp group by deptno;

DEPTNO
10
20
30

Note: *In all relational databases whenever we are trying to display group functions with normal function then database server return an error. To overcome this problem in this case we must use "GROUP BY" clause.*

Example

Step 1

SQL> select sum(sal) from emp;
Output:37475

Step 2

SQL> select deptno, sum(sal) from emp;
Error: not a single-group group function

Solution

SQL> select deptno, sum(sal) from emp group by deptno;

Output:

DEPTNO	SUM(SAL)
30	9400
20	10875
10	8750

- ***Write a query to display those departments having more than 3 employees from emp table by using group by clause?***

Ans: SQL> select deptno, count(*) from emp group by deptno where count(*)>3;
Error: SQL commond not properly ended.

Solution

SQL> select deptno, coutnt(*) from emp group by deptno having count(*)>3;

Output:

DEPTNO	COUNT(*)
30	6
20	5

In all relational databse systems after "GROUP BY" clause we are not allowed to use "WHERE" clause. In place of this one ANSI/ISOSQL provided another clause "HAVING".

Generally if we want to restrict rows from a table then we are using "WHERE" clause, whereas if we want to restrict groups after "GROUP BY" clause then we must use "HAVING" clause.

Generally we are not allowed to use "GROUP" functions in "WHERE" clause where as in "HAVING" clause we can also use "GROUP" functions.

- ***Write a query to display those departments having more than 12,000 sum(sal) from emp table by using group by clause?***

Ans: select deptno, sum(sal) from emp group by deptno having sum(sal)>12,000;

DEPTNO	SUM(SAL)
20	13925
10	12050

- ***Write a query to display year, number of employees per year in which more than one employee was hired from emp table using "group by"?***

Ans: select to_char(hiredate, 'YYYY') year, count(*) from emp group by to_char(hiredate, 'YYYY');

(or)

select to_char(hiredate, 'YYYY') year, count(*) from emp group by to_char(hiredate, 'YYYY') having count(*)>1;

YEAR	COUNT(*)
1981	10
1987	2

Note: *In all relational databases we are not allowed to use column alias name within "GROUP BY" clause.*

Note: *In all relational databases we can also use invisible "GROUP" function within "HAVING" clause, because whenever we are using "GROUP" function report by using "GROUP BY" clause then database server internally having all other "GROUP" functions.*

- ***Write a query to display those deptno, sum(sal) having more than 3 employees from EMP table by using group by clause?***

Ans: select deptno, sum(sal) from emp group by deptno having count (*)>3;

Output:

DEPTNO	SUM(SAL)
30	11500
20	13925

Order by clause is used to shorting rows either in ascending order (or) in descending order. This clause also used in "SELECT" statement, along with "ORDER BY" clause we are using two keywords these are "ASC" and "DESC". In all databases by default "ORDER BY" clause has ascending order.

Syntax: select * from tablename order by columnname [asc/desc];

Example

```
SQL> select sal from emp order by sal desc;
SQL> select sal from emp order by sal asc;
```

Note: In all relational databases we can also use column alias name or expression within "ORDER BY clause.

Example

```
SQL> select ename, sal*12 annsal from emp order by annsal desc;
SQL> select ename, sal*12 annsal from emp order by sal*12 desc;
```

Note: *We can also use number in place of columnname within ORDER BY clause, in this case this number represent column position within select list.*

Example select empno, ename sal from emp order by 3 desc;

Note: *In all relational databases when we are using null value column within "ORDER BY" clause then by default null value are treated as highest value. To overcome this problem if you want to control this value then oracle provided "NULLS FIRST" (or) "NULLS LAST" clause.*

Syntax: select * from tablename order by columnname [asc/desc] {nulls first/nulls last};

Example

```
SQL> select ename, comm from emp order by comm desc;
SQL> select ename, comm from emp order by comm desc nulls last;
```

Note: *In oracle we can also use more than one column within "ORDER BY" clause, in this case oracle server shorting database on a first column. If first column has duplicate data then only that group is shorted in second column.*

Select Statement

Syntax: select col1, col2,... from <tablename> <where condition> group by <column name>[asc/desc];

Example

```
select deptno, count(*) from emp
where sal>1000 group by deptno
having count(*)>3 order by deptno desc;
```

DEPTNO	COUNT(*)
30	6
20	5

Joins are used to retrieve data from multiple tables. In all relational databases when we are joining "n" tables then we must use "n-1" joining conditions.

Oracles having following types of joins.These are:

1. **Equi Join (Or) Inner Join**
2. **Non Equi Join**
3. **Self Join**
4. **Outer Join**

These 4 joins are also called as oracle "8i joins".

9i joins or ANSI joins

1. **Inner Join**
2. **Left Outer Join**
3. **Right Outer Join**
4. **Full Outer Join**
5. **Natural Join**

Note: *In oracle, we can also retrieve data from multiple tables without using join condition, in this case oracle server internally uses cross join (default join) but cross join is implemented based on Cartesian product that's why this join returns more duplicate data.*

Example: SQL> select ename, sal, dname, loc from emp, dept;

1) Equi Join (Or) Inner Join

Based on equality condition we are retrieving data from multiple tables. Here joining conditional columns must belong to same datatype.

Generally, whenever tables having common columns then only we are allowed to use equi join and also these common columns must belongs to same data type.

```
Syntax: select col1, col2, col3 ..... from table1, table2
        where table1.CommonColumnName = table2.CommonColumnName;
```

(Tablename.CommonColumnName is a join condition)

Example

SQL> select ename, sal, deptno, dname, loc from emp, dept where emp.deptno= dept.deptno;
Error: column ambiguously defined.

Solution: select ename, sal, dept.deptno, dname, loc from emp, dept where emp.deptno=dept.deptno;

Note: *Generally, to avoid ambiguity in future we must specify every column name along with table name using full stop or dot(.) operator after select.*

Note: *Always Equi join returns matching rows only. (Here deptno 40 doesn't display when we are using dept.deptno also)*

Using Alias Names

In oracle we can also use table ALIAS names in join, whenever joins having ALIAS name then those joins improve performance of the query because those join retrieve data very fast from database.
In this case first we must create table ALIAS name in "FROM" clause and then only we must use those alias name in place of table name within SELECT list and also within joining condition.

Syntax: from tablename1 aliasname1, tablename2 aliasname2;

These alias names must be different names.

Note: *In relational database when a join having alias names then those types of joins improve performance of the application.*

Example: select ename, sal, d.deptno, dname, loc from emp e, dept d where e.deptno=d.deptno;

ENAME	SAL	DEPTNO	DNAME	LOC
CLARK	2450	10	ACCOUNTING	NEW YORK
KING	5000	10	ACCOUNTING	NEW YORK
MILLER	1300	10	ACCOUNTING	NEW YORK
JONES	2975	20	ACCOUNTING	DALLAS
FORD	1100	20	RESEARCH	DALLAS
ADAMS	800	20	RESEARCH	DALLAS
SMITH	3000	20	RESEARCH	DALLAS
SCOTT	1250	20	RESEARCH	DALLAS
WARD	1500	30	RESEARCH	CHICAGO
TURNER	1500	30	SALES	CHICAGO
ALLEN	1600	30	SALES	CHICAGO
JAMES	950	30	SALES	CHICAGO
BLAKE	2850	30	SALES	CHICAGO
MARTIN	1250	30	SALES	CHICAGO

In the above query, here deptno ("40") doesn't display if we are using **d.deptno** also.

Note: *In all databases always EQUI joins returns matching rows only.*

2) Non - Equi Join

Based on other than equality condition (not equal to, >, <, <>, >=, <=, between, in,...) we can retrieve data from multiple tables.

Example

```
SQL> create table test1(deptno number(10), ename varchar2(20));
SQL> insert into test1 values(10, 'a');
SQL> insert into test1 values(20, 'b');
SQL> select * from test1;
```

DEPTNO	ENAME
10	a
20	b

```
SQL> create table test2(deptno number(10),dname varchar2(20));
SQL> insert into test2 values(10, 'c');
SQL> insert into test2 values(20, 'd');
SQL> insert into test2 values(30, 'e');
SQL> select * from test2;
```

DEPTNO	DNAME
10	c
20	d
30	e

```
SQL> select * from test1, test2 where test1.deptno > test2.deptno;
```

DEPTNO	ENAME	DEPTNO	DNAME
20	b	10	c

Note: *In oracle, using non-equi join we can also retrieve data from multiple tables when table doesn't have common columns but in this case one table one column values lies between another tables to column.*

Example

```
SQL> select * from emp;
SQL> select * from salgrade;
SQL> select ename, sal, losal, hisal from emp, salgrade where sal between losal and hisal;
```

(or)

```
SQL> select ename, sal, losal, hisal from emp, salgrade where sal >= losal and sal <= hisal;
```

Equi, Non-equi Join Query

- ***Write a query to display the employees who are working in the location CHICAGO from emp, dept tables using equi join?***

Ans: select ename, loc from emp, dept where emp.deptno=dept.deptno and loc= 'CHICAGO';

ENAME	LOC
WARD	CHICAGO
TURNER	CHICAGO
ALLEN	CHICAGO
JAMES	CHICAGO
BLAKE	CHICAGO
MARTIN	CHICAGO

***Note**: If we want to filter the data after joining condition then we must use "AND" operator in 8i joins where as in 9i joins we are using either "AND" operator or "WHERE" clause also.*

- ***Write a query to display dname, sum (sal) from emp, dept table by using equi join?***

Ans: select dname, sum (sal) from emp e, dept d where e.deptno = d.deptno group by dname;

DNAME	SUM(SAL)
ACCOUNTING	8750
RESEARCH	10875
SALES	9400

```
SQL> select dept.deptno, dname, sum(sal) from emp,dept
     where emp.deptno=dept.deptno
     group by dept.deptno,dname;
```

DEPTNO	DNAME	SUM(SAL)
10	ACCOUNTING	8750
20	RESEARCH	10875
30	SALES	9400

- ***Write a query to display number of employees, min(sal), max(sal), sum(sal) in each location from emp, dept tables using equi join?***

Ans:
```
SQL> select loc, count(*), min(sal),max(sal),sum(sal) from emp e, dept d
     where e.deptno=d.deptno group by loc;
```

LOC	COUNT(*)	MIN(SAL)	MAX(SAL)	SUM(SAL)
NEW YORK	3	1300	5000	8750
CHICAGO	6	950	2850	9400
DALLAS	5	800	3000	10875

3) Self Join

Joining a table itself is called "SELF JOIN". Here joining conditional columns must belong to same datatype. Before we are using "SELF JOIN" we must create ALIAS names for the table in "FROM" clause.

In all relational databases self join is used in following two scenarios these are...

- Compare one value with all other values in a same column.
- Compare two different column values in a single table then we must use self-join, but here these two columns must belong to same datatype.

Method 1: Compare one value with all other column values in same column

In all databases systems whenever we are refer same table more than one time for query results then we must create table ALIAS names in "FROM" clause. This ALIAS name must be different name. These ALIAS names are also called as "Reference names" for the table. These ALIAS names internally behaves like an exact tables when query execution time.

Syntax: from tablename aliasname1, tablename aliasname2;

- ***Write a query to display the employees who are getting same salary as SCOTT salary from emp table using self join?***

Ans:

```
select e2.ename, e1.sal
 from emp e1, emp e2
 where e1.sal = e2.sal
  and e1.ename = 'SCOTT';
```

***Note**: When we are comparing one column any value with all other value in same column by using "self join" then we must display data from second ALIAS table only.*

Method 2: Compare two different column values from a same table (These columns belongs to same data type)

In all relational databases we can also store tree structure data into relational table. In this case that tables having minimum 3 columns. In these 3 columns 2 columns must belongs to same datatype and also these 2 columns having some related data.

In all relational databases if you want to retrieve this hierarchical data then we must compare these hierarchical columns by using self join.

- ***Write a query to display employee names and their manager names from emp table using self join?***

Ans:

```
select e1.ename "employees", e2.ename "managers"
 from emp e1, emp e2
 where e1.mgr = e2.empno;
```

(or)

```
select e1.ename "employees", e1.mgr, e2.empno, e2.ename "managers"
 from emp e1, emp e2
 where e1.mgr = e2.empno;
```

- ***Write a query to display the employees who are getting more salary than their manager salary from emp table using "self join"?***

Ans:

```
select e1.ename "employees", e1.sal, e2.ename "manager", e2.sal
 from emp e1, emp e2
 where e1.mgr = e2.empno
  and e1.sal > e2.sal;
```

- ***Write a query to display the employees who are joining before their managers from emp table using self join?***

Ans:

```
select e1.ename "employees", e1.hiredate, e2.ename "manager", e2.hiredate
 from emp e1, emp e2
 where e1.mgr = e2.empno
  and e1.hiredate < e2.hiredate;
```

4) Outer Join

Outer join is used to retrieve all rows from one table and matching rows from another table.
Generally, Equi Join returns matching rows only. If we want to retrieve non-matching rows also then we are using join operator **(+)**.
Within joining condition of the EQUI JOIN this join is also called as ORACLE 8i Outer join.

***Note**: This join operator can be used only one side at a time within joining condition.*

Example

```
SQL> select ename, sal, d.deptno, dname, loc from emp e, dept d where e.deptno (+) = d.deptno;
```

e.deptno → Matching rows
d.deptno → All rows

ENAME	SAL	DEPTNO	DNAME	LOC
CLARK	2450	10	ACCOUNTING	NEW YORK
SCOTT	1250	20	RESEARCH	DALLAS
MARTIN	1250	30	SALES	CHICAGO
-	-	40	OPERATIONS	BOSTON

Note

*In oracle if you want to retrieve matching or non-matching rows from all tables then we are using **"full outer join"**, but **"full outer join"** is a 9i join , prior to oracle 9i if we want to retrieve all data then we are using join operator within joining condition one time left side and another time right side.*

Example

```
select ename, sal, d.deptno, dname, loc
 from emp e, dept d
 where e.deptno(+) = d.deptno
Union
select ename, sal, d.deptno, dname, loc
 from emp e, dept d
 where e.deptno = d.deptno(+);
```

9i Joins (or) ANSI Joins

Oracle 9i onwards oracle also supports ANSI standard joins same like all other relational databases, this type of join is called as ANSI joins or 9i joins.

1. **Inner Join**
2. **Left Outer Join**
3. **Right Outer Join**
4. **Full Outer Join**
5. **Natural Join**

1) **Inner Join**

This join also return matching rows only, here also joining conditional column must belong to same data type. Whenever a table having common columns then only we are allowed to use inner join.

Inner join (9i) performance is very high compare to oracle 8i equi join.

Example: SQL> select ename, sal, d.deptno, dname, loc from emp e join dept d on e.deptno=d.deptno;

- **Write a query to display the employees who are working in the location "CHICAGO" from emp, dept tables using 9i inner join?**

Ans:

```
select ename, loc
 from emp e
 join dept d
   on e.deptno = d.deptno
 where loc = 'CHICAGO';
```

Example

```
SQL> create table t1(a varchar2(10), b varchar2(10),c varchar2(10));
SQL > insert into t1 values('x','y','z');
SQL > insert into t1 values('p','q','r');
SQL> select * from t1;

SQL> create table t2(a varchar2(10), b varchar2(10));
SQL>insert into t2 values('x','y');
SQL.> select * from t2;
```

A	B	C
X	y	z
P	q	r

A	B
x	Y

8i Equi Join: SQL> select * from t1, t2 where t1.a=t2.a and t1.b=t2.b;

9i Inner Join: SQL> select * from t1 join t2 on t1.a=t2.a where t1.b=t2.b;

Output

A	B	C
X	Y	Z

A	B
X	Y

USING Clause

In 9i joins we can also use "USING" clauses in place of "ON" clause. "USING" clause performance is very high compare to "ON" clause and also when we are using "USING" clause then oracle server automatically returning common columns one time only.

Syntax: select * from tablename1 join tablename2 using (Common Column Names);

Example: SQL> select * from t1 join t2 using (a, b);

A	B	C
X	Y	Z

Note: *Whenever we are using "USING" clause then we are not allowed to use ALIAS name (or) joining conditional columns.*

Example: SQL> select ename, sal, deptno, dname, loc from emp join dept using (deptno);

2) Left Outer Join

This join always returns all rows from left side table and matching rows from right side table and also returns "NULL" values in place of non-matching rows in another table.

Example

SQL>select * from t1;

A	B	C
x	y	z
p	q	r

SQL> select * from t2;

A	B
x	y
s	t

SQL> select * from t1 left outer join t2 on t1.a=t2.a and t1.b=t2.b;

A	B	C	A	B
x	y	z	x	y
p	q	r	Null	Null

3) Right Outer Join

This join returns all rows from right side table and matching rows from left side table and also returns null values in place of "non-matching" rows in another table.

Example SQL> select * from t1 right outer join t2 on t1.a = t2.a and t1.b = t2.b;

A	B	C	A	B
X	y	z	x	y
-	-	-	s	t

4) Full Outer Join

This join always returns all rows from all the tables because it is the combination of LEFT and RIGHT OUTER JOINS. This join always returns matching and non-matching rows and also this join returns NULL values in place of non-matching rows.

Example SQL> select * from t1 full outer join t2 on t1.a=t2.a and t1.b=t2.b;

A	B	C	A	B
x	y	z	x	y
p	q	r	-	-
-	-	-	s	t

5) Natural Join

This join also returns matching rows only. This join performance very high compare to INNER JOIN. In this join we are not allowed to use joining condition explicitly because whenever table having common columns then based on those common columns oracle server only internally automatically establishes joining condition based on those common columns.

Syntax: select * from tablename1 natural join tablename2;

Note: *Natural Join performance is very high compared to inner join.*

Note: *Natural join internally having uses "USING" clause, that's why this join also returns common columns one time only.*

Example

SQL> select * from t1 natural join t2;

Output

A	B	C
x	y	Z

Note: *When we are using "NATURAL JOIN" also then we are not allowed to use ALIAS name on joining conditional column because "NATURAL JOIN" internally uses "USING" clause.*

Example SQL> select ename, sal, deptno, dname, loc from emp e natural join dept d;

Cross Join

Example

```
SQL>select ename, sal, dname, loc from emp cross join dept;
```

When emp –> 14 rows
dept –> 4 rows
Then output –> 14*4 = 56 rows selected

Note: *In all relational databases we can also retrieve data from multiple tables by using joins, when this table does not have common column also.*

8i JOINS

Syntax:

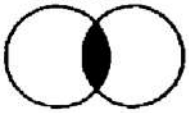
```
select col1, col2, ..............
  from table1, table2, table3
where table1 . CommonColumnName = table2 . CommonColumnName
  and table2 . CommonColumnName = table3 . CommonColumnName;
```

9i JOINS

Syntax:

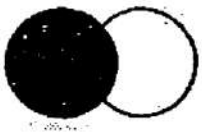
```
select col1, col2, ...........
  from table1
join table2
  on table1 . CommonColumnName = table2. CommonColumnName
join table3
  on table2 . CommonColumnName = table3. CommonColumnName;
```

INNER JOIN



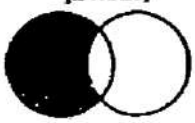
Syntax :
select * from tableA join tableB on tableA.key = tableB.key;

LEFT JOIN



Syntax :
Select * from tableA left join tableB on tableA.key = tableB.key;

LEFT JOIN
(If NULL)

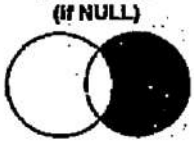


Syntax :
Select * from tableA left join tableB on tableA.key =tableB.key
where tableB.key is null;

RIGHT JOIN

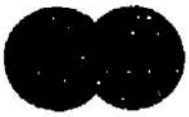
Syntax :
Select * from tableA right join tableB on tableA.key = tableB.key;

RIGHT JOIN
(If NULL)



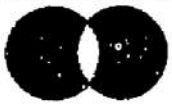
Syntax :
Select * from tableA right join tableB on tableA.key = tableB.key
where tableA.key is null;

FULL OUTER JOIN



Syntax :
Select * from tableA full join tableB on tableA.key = tableB.key;

FULL OUTER JOIN



(If null)

Syntax :
Select * from tableA full join tableB on tableA.key = tableB.key
where table A.key is null;

Constraints are used to prevent or stops invalid data entry into our tables. Generally constraints are created on table columns.

Oracle server has following types of constraints.

1. **Not Null**
2. **Unique**
3. **Primary Key**
4. **Foreign Key**
5. **Check**

In all databases all above constraints are defined into two ways.

A. Column Level
B. Table Level

A. Column Level

In these methods we are defining constraints on individual columns i.e. whenever we are defining the column then only immediately we are specifying constraint type.

```
Syntax: create table tablename(col1 datatype(size) constrainttype, col2 datatype(size) constrainttype,.....);
```

B. Table Level

In this method we are defining constraints on group of columns i.e. in this method first we are defining all columns and last only we are specifying constraint type along with group of columns.

```
Syntax:  create table tablename (col1 datatype(size),col2 datatype(size),....., constrainttype (col1,col2,....));
```

1) Not Null

In all relational databases not null does not support table level. Not null constraint doesn't accept null values, but it will accept duplicate values.

Column Level

Example: SQL> create table t1 (sno number(10) not null, name varchar2(10));

Testing

```
SQL> insert into t1 values (null, 'abc');
Error: ORA-1400: cannot insert null into SNO;

SQL> insert into t1 values(1, 'x');
SQL> insert into t1 values(1, 'y');
SQL> select * from t1;
```

SNO	NAME
1	x
1	y

2) Unique

This constraint is defined on column level, table level. This constraint doesn't accept duplicate values but it will accept null values.

Note: *Whenever we are creating unique constraints then oracle server internally automatically creates b-tree index on those columns.*

Column level: SQL> create table t2(sno number(10) unique, name varchar2(10));

Table level: SQL> create table t3(sno number(10), name varchar2(10), unique(sno,name));

Example:

```
SQL> select * from t3;
```

SNO	NAME
1	shailendra
1	abc

```
SQL> insert into t3 values(1, 'abc');
Error: ORA-00001: Unique constraint violated.
```

3) Primary Key

Primary key "Uniquely identifying a record in a table". There can be only one primary key in a table and also primary key doesn't accept duplicate, null values.

Note: *Whenever we are creating primary key then oracle server automatically creates "b-tree" indexes on those columns.*

Column level: SQL> create table t4 (sno number(10) primary key, name varchar2(10));

Table level: SQL> create table t5 (sno number(10), name varchar2(10), primary key(sno,name));

This is also called as "Composite Primary Key" i.e. it is the combination of columns as a "single primary key".

4) Foreign Key

In all relational databases if you want to establish relationship between tables then we are using *"Referential Integrity Constraints"* (Foreign Key).

Generally one table foreign key must belongs to another table *"PRIMARY KEY"* and also these PRIMARY, FOREIGN columns must belong to same datatypes. Always *"FOREIGN KEY"* values are based on *"PRIMARY KEY"* values only. Generally PRIMARY KEY doesn't accept duplicate, null values where as *"FOREIGN KEY"* accepts duplicate, null values.

Column level (References)

```
Syntax: create table tablename(col1 datatype(size) references MasterTableName(primary colname),col2
datatype(size).....);
```

Example

```
SQL> create table h4 (sno number(10) references t4);
                    (or)
SQL> create table g4(x number (10) references t4(sno));
```

Table Level (Foreign key references)

Syntax: *create table tablename (col1 datatype(size), col2 datatype(size),....., foreign key (col1,col2,.....) references MasterTableName (primary columns));*

Example: SQL> create table h5 (sno number(10), name varchar2(10), foreign key(sno, name) references t5);

- Whenever we are establishing relationship between tables then oracle server violates following two rules:-

  A. **Deletion in "Master" table**
  B. **Insertion in "Child" table**

A. **Deletion in "Master" table**

Whenever we are trying to delete a master table record in master table, if the record is available in child table then oracle server returns an error "ORA-2292".

To overcome this problem if we want to delete master table record in master table then first we must delete child table records in "child table" and then only we are allow deleting those records in "master table", otherwise using an "ON DELETE CASCADE" clause.

On Delete Cascade

This is an optional clause used along with "FOREIGN KEY" constraint only. Whenever we are using "ON DELETE CASCADE" clause in child table then if we are deleting a record from "master table" then oracle server automatically that records is deleted from "master table" and also those record are automatically deleted from "child table".

Syntax: *create table tablename (col1 datatype(size) references mastertablename (primary key colname) on delete cascade,.....);*

Example

```
SQL> create table mas(sno number(10) primary key);
SQL> insert into mas values(.............);
SQL> select * from mas;
       SNO
  ----------
         1
         2
         3

SQL> create table child (sno number(10) references mas(sno) on delete cascade);
SQL> insert into child values(........);
SQL> select * from child;
       SNO
  ----------
         1
         1
         2
         2
         3
         3
```

Testing: (Deletion in Master Table)

```
SQL> delete from mas where sno=1;
1 row deleted
SQL> select * from mas;
                    SNO
                    ----------
                     2
                     3

SQL> select * from child;
                    SNO
                    ----------
                     2
                     2
                     3
                     3
```

Note: *Oracle also supports "ON DELETE SET NULL" clause along with FOREIGN KEY.*

On Delete Set Null

In Oracle foreign key constrains also having another optional clause "ON DELETE SET NULL". Whenever we are using these clause in child table then we are deleting primary key value from master table, then automatically that value is deleted from master table and also those value are automatically set to null in foreign key within child table.

Syntax: *create table tablename(col1 datatype(size) reference master tablename(primary key colname) on delete set null,.....);*

B. **Insertion In Child Table**

In Oracle, whenever we are trying to insert other than "PRIMARY KEY" values into "FOREIGN KEY" then oracle server returns an error "ORA-2291" because in all relational database systems always FOREIGN KEY values are based on PRIMARY KEY values only.

Example

```
SQL> insert into child values(5);
Error:  ORA-2291: Intergrity constraint violated-parent key not found.
```

Solution

```
SQL> insert into mas values(5);
SQL> select * from mas;
SQL> insert into child values(5);
SQL> select * from mas;
```

e) Check: Check constraints are used to define logical conditions according to client's business rules.

Note: *In Oracle check constraints does not work with SYSDATE.*

Column level

Syntax: *create table tablename (col1 datatype(size) check (logical condition)),col2 datatype(size) .....);*

Example 1

```
SQL> create table test (name varchar2(10) check (name=upper(name)));
SQL> insert into test values ('shailendra');
Error:   check constraints (SCOTT.SYS_C006176) violated.
SQL> insert into test values ('SHAILENDRA');
        1 row created.
SQL> Select * from test;
```

Example 2

```
SQL> create table test1 (sal number(10) check (sal>5000));
SQL> insert into test1 values(2000);
Error: check constraints (SCOTT.SYS_C006177) violated
SQL> insert into test1 values(9000);
```

Table Level

Syntax: *create table tablename (column1 datatype(size), column2 datatype(size),.....*
check (condition1 and condition2,.....));

Example

```
SQL> Create table test2(name  varchar2(10), sal  number(10) check(name=upper(name) and sal>5000));
SQL> insert into test2 values('india', 9000);
Error: check constraint (SCOTT.SYS_C006178) violated
SQL> insert into test2 values('INDIA',9000);
```

Assign user defined names to constraints (or) User defined constraint names

In all relational database systems whenever we are creating constraints then database servers internally automatically generates a unique identification number for indentifying a constraint uniquely.

In Oracle also whenever we are creating constraints then oracle server internally automatically generates a unique identification number in the format of "SYS_CN" for identifying a constraint uniquely, this format is called as a "Predefined Constraint Name". In place of this one we can also create our own name by using "Constraint Keyword". This is called "User defined Constraint Name".

Syntax: *Constraint <user defined name> <constraint type>;*

Example: 1 (Predefined Constraint Name)

```
SQL> create table test (sno number(10) primary key);
```

Testing

```
SQL> insert into test values(1);
SQL> insert into test values(1);
Error: unique constraint (SCOTT.SYS_C006180) violated
```

Example: 2 (User defined Constraint Name)

```
SQL> create table test1 (sno number(10) constraint pk_abc  primary key);
```

Testing

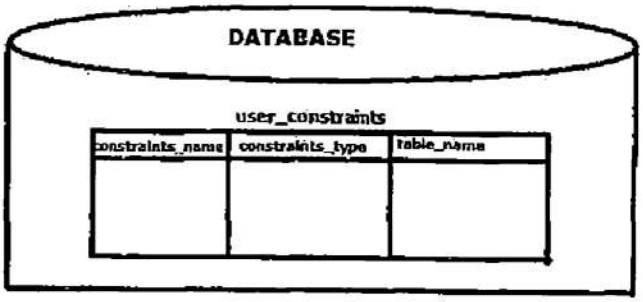
```
SQL> insert into test1 values(1);
SQL> insert into test1 values(1);
Error: unique constraints (SCOTT.PK_ABC) violated
```

Whenever we are installing oracle server then oracle server automatically creates some read only tables. These read only tables are also called as "Data Dictionaries".
These read only table stores specific information related to database objects. This information is also called as Meta data. This table stores column information, data type information, constraints information, indexes information.
In oracle if you want to view all read only tables then we are using following select statement.

Syntax: *select * from dict;*

We cannot insert, update, delete data in read only table and also we cannot drop this read only table.

Note 1: In Oracle, all constraints information stored under "user_constraints" data dictionary.



Example

SQL> desc user_constraints;
SQL> select CONSTRAINT_NAME, CONSTRAINT_TYPE from USER_CONSTRAINTS where table_name= 'EMP';

- Here, EMP is capital letter only.

Output:

CONSTRAINT_NAME	CONSTRAINT_TYPE
PK_EMP	P
FK_DEPTNO	R

SQL> create table emp(empno number(4) constraint pk_emp primary key,..... deptno number(2) constraint fk_deptno references dept(deptno));

Note 2: In Oracle, if you want to view column name along with constraints name then we are using "user_cons_columns" data dictionaries.

Example

SQL> desc user_cons_columns;
SQL> select constraint_name, column_name from user_cons_columns where table_name= 'EMP';

Output

CONSTRAINT_NAME	COLUMN_NAME
PK_EMP	EMPNO
FK_DEPTNO	DEPTNO

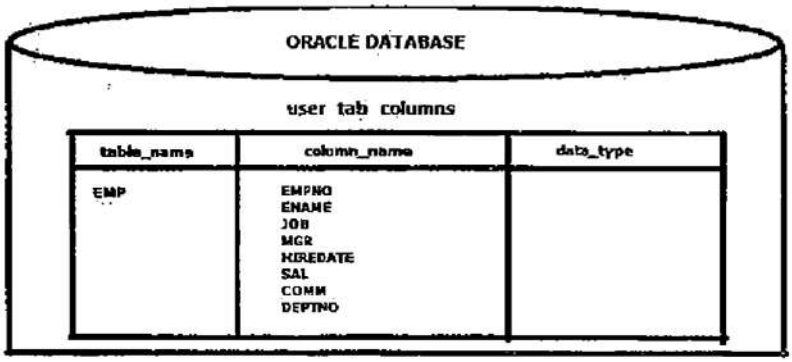
Note 3: In Oracle, if you want to view logical conditions of the check constraint then we are using "search_condition" property from "user_constraints" data dictionary.

Example

```
SQL> desc user_constraints;
SQL> select search_condition from user_constraints where table_name= 'TEST';
```

SEARCH_CONDITION
Name=upper(name)

Note: In Oracle, all columns information stored under "user_tab_columns" data dictionary.



Example

```
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

- **Write a query to display number of columns from emp table?**

Ans: SQL> select count(*) from user_tab_columns where table_name = 'EMP';

COUNT(*)
8

Default Clause

In all relational databases if you want to provide default value into the table column then we are using default clause.

Syntax: *columnname datatype(size) default defaultvalues;*

Example

```
SQL> create table b1(name varchar(10), sal number(10) default 5000);
SQL> insert into b1(name) values ('abc');
SQL> select * from b1;
```

NAME	SAL
abc	5000

Note: In Oracle if you want to view default values of a table column then we are using "**data_default**" properties from "**user_tab_columns**" data dictionary.

Example:

```
SQL> desc user_tab_columns;
SQL> select column_name, data_default from user_tab_columns where table_name='B1';
```

COLUMN_NAME	DATA_DEFAULT
SAL	5000

Default On Null (Oracle 12C)

In all relational databases always by default null values overwrites default values.

To overcome this problem oracle 12C introduced default or null clause with table column. Whenever we are using this clause automatically oracle server insert default values, when we are using insert statement.

Syntax: *colname datatype(size) default on null defaultvalues;*

Example

```
SQL> create table test (name varchar2(10), city varchar2(10) default 'hyd');
SQL> Insert into test values ('abc' , null);
SQL> select * from test;
```

NAME	CITY
abc	hyd

Solutions: (Oracle 12c – Default on null)

```
SQL> create table test1 (name varchar2(10), city varchar2(10) default on null 'hyd');
SQL> insert into test1 values ('abc' , null);
SQL> select * from test1;
```

NAME	CITY
abc	

Truncate Table Cascade (12C)

Generally we cannot truncate master table. To overcome this problem Oracle 12C introduced cascade clause along with truncate table, which is used to truncate master table but before we are using this clause child table "FOREIGN KEY" must have "on delete cascade" clause.

Syntax: *truncate table <table name> cascade;*

Example

```
SQL> create table mas(sno number(10) primary key);
SQL> insert into mas values(........);
SQL> select * from mas;
```

SNO
1
2
3

```
SQL> create table child (sno number(10) references mas on delete cascade);
SQL> insert into child values(.........);
SQL> select * from child;
```

SNO
1
1
2
2
3
3

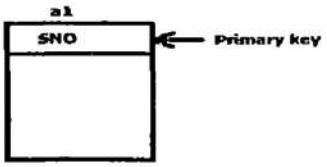
```
SQL> truncate table mas;
Error: Unique primary key in table referenced by enabled foreign keys.
```

Solution SQL> truncate table mas cascade;

In Oracle, by using "ALTER" command we can also add or drop constraints of existing table.

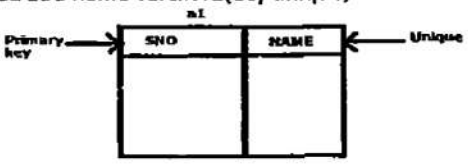
Note 1: In all relational databases if you want to add constraints on existing table or existing column then we are using "table level syntax method".

Example: SQL> alter table a1 add primary key(sno);



Note 2: In Oracle If we want to add a new column along with constraint then we are using "Column level syntax method".

Example: SQL> alter table a1 add name varchar2(10) unique;



Adding Foreign Key

Example

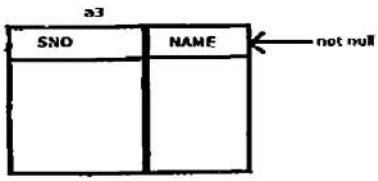
```
SQL> create table a2(sno number(10));
SQL> alter table a2 add foreign key(sno) references a1(sno);
Table altered
```

Note 3: In Oracle If we want to add "NOT NULL" constraints into existing table or existing column then we are using "alter with modify".

Syntax: *alter table <table name> modify <column name> not null;*

Example:

```
SQL> create table a3(sno number(10));
SQL> alter table a3 modify sno not null;
```



- **Write a query to add new column with not null into a3?**

Ans: SQL> alter table a3 add name varchar2(10) not null;

Note: In Oracle, whenever we are copying a table from another table then all constraints are never copied, but in all database systems "**not null**" constraints only copied.

Example

```
SQL> create table a4 as select * from a3;
"not null is copied"
SQL> desc a4;
```

Example

```
SQL> create table emp1 as select * from emp;
SQL> select * from emp1;
SQL> alter table emp1 add primary key(empno);
SQL> alter table emp1 add foreign key deptno references emp1(deptno) on delete cascade;
```

Dropping Constraints

By dropping **"constraint name"** two methods for dropping constraints.

Method 1

Syntax: Alter table <table name> drop constraint <constraint name>;

Method 2

Syntax: Alter table <table name> drop primary key;
Syntax: Alter table <table name> drop unique(col1,col2,....);

Example 1

```
SQL> create table test (sno number(10) primary key);
SQL> alter table test drop constraint primary key;
```

Note: In Oracle, if we want to drop primary key along with references foreign key then we are using "CASCADE" clause along with "Alter... drop".

Syntax: *alter table <table name> drop primary key cascade;*

Example 2

```
SQL> alter table a1 drop primary key;
Error: this unique/primary key is referenced by some foreign keys.
```

Solution: SQL> alter table a1 drop primary key cascade;

Example 3

```
SQL> select COLUMN_NAME, CONSTRAINT_NAME from USER_CONS_COLUMNS
     where TABLE_NAME = 'A3';
```

COLUMN_NAME	CONSTRAINT_NAME
NAME	SYS_C005526
SNO	SYS_C005525

```
SQL> alter table a1 drop CONSTRAINT SYS_C005526;
SQL> desc a1;
```

Query within another query is called "Subquery" or "Nested Query". Subquery is used to retrieve data from "single" or "multiple" tables based on "more than one step process".

All relational databases having two type of subquery. These are ...

1. **Non- Correlated Subquery**
2. **Correlated Subquery**

In Non-correlated subquery "child query" is executed first then only "parent query" is executed, where as in correlated subquery "parent query" is executed first then only "child query" is executed.

1) **Non-Correlated Subquery:** Every non-correlated subquery mainly consists of two parts.

a) Child query
b) Parent query

a) **Child Query:** A query which provides values to another query is called "Child Query or Inner Query".
b) **Parent Query:** A query which receives values from another query is called "Parent Query or Outer Query". In non-correlated subquery maximum limit is up to "255" queries.

Non-Correlated Sub Query

a) Single row subquery
b) Multiple row subquery
c) Multiple column subquery
d) Inline view (or) subquery are using in "from" clause

- ***Write a query to display the employees who are getting more salary than the avg(sal) from emp table?***

Ans: SQL> select ename, sal from emp where sal > (select avg(sal) from emp);

In the above query, this is a "single row subquery" because here child query returns single value.
In Single row subquery we are using =, <, >, <=, >=, between operators.

Execution

Step 1: SQL> select avg(sal) from emp;
Output: 2830.35714
Step 2: SQL> select * from emp where sal>2830.35714

- ***Write a query to display the employees who are working in sales department from emp, dept table?***

Ans: SQL> select ename from emp where deptno = (select deptno from dept where dname= 'SALES');

90 % : Same column
5 % : Group function
3 % : Expression
2 % : Different column

Note: Generally In all relational databases, we are not allowed to use child query table column with parent query, because subquery always returns parent query table column in final output.
To overcome this problem if you want to use child query table column within "Parent query", then we must use "joins" within parent query.

Example

- ***Write a query to display the employees who are working in sales department from emp, dept tables?***

Ans:

```
select ename, dname from emp e, dept d
where e.deptno = d.deptno
and d.deptno = (select deptno from dept where dname = 'SALES');
```

ENAME	DNAME
ALLEN	SALES
WARD	SALES
MARTIN	SALES
BLAKE	SALES

- ***Write a query to display senior most employee details from emp table?***

Ans: select * from emp where hiredate = (select min(hiredate) from emp);

- ***Write a query to display the employees who are working same as "SMITH" deptno from dept table?***

Ans: select ename from emp where deptno = (select deptno from emp where ename= 'SMITH');

- ***Write a query to display the employees who are getting more salary than the highest paid employee in 20th department from emp table?***

Ans: select * from emp where sal > (select max(sal) from emp where deptno=20);

- ***Write a query to display the employees who are getting more salary than the lowest paid employee in 10th department from emp table?***

Ans: select * from emp where sal > (select min(sal) from emp where deptno=10);

- ***Write a query to display 2nd highest salary from emp table?***

Ans: select max(sal) from emp where sal < (select max(sal) from emp);

- ***Write a query to display 2nd highest sal employee details from emp table?***

Ans: select * from emp where sal = (select max(sal) from emp where sal <(select max(sal) from emp));

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7788	SCOTT	ANALYST	7566	19-APR-87	3100		20
7902	FORD	ANALYST	7566	03-DEC-81	3100		20

- ***Write a query to display highest sal details emp table?***

Ans: select * from emp where sal=(select max (sal) from emp);

- ***Write a query to display highest paid employee department, dname from emp, dept tables?***

Ans: select dname from dept where deptno = (select deptno from emp where sal=(select max(sal) from emp));

Output:

DNAME
Accounting

GROUP BY clause used in "Single Row Subquerys"

In all relational databases we can also use GROUP BY clause in parent query.

- ***Write a query to display lowest average salary job from emp table by using group by clause?***

Ans: select job, avg(sal) from emp group by job having avg(sal) = (select min(avg(sal)) from emp);

Error: nested group function without Group By

Note: In all relational database systems whenever child query having nested group functions then we must use "Group by" clause within child query, otherwise database server returns an error.

Solution

```
select job, avg(sal) from emp group by job
having avg(sal) = (select min(avg(sal)) from emp group by job);
```

JOB	AVG(SAL)
SALESMAN	2362.5

- ***Write a query to display those job average salary is more than the "CLERK" job average salary from emp table?***

Ans:

```
select job, avg(sal) from emp group by job
having avg(sal) > (select avg(sal) from emp where job = 'CLERK');
```

JOB	AVG(SAL)
PRESIDENT	5100
MANAGER	4083.3333
ANALYST	4200

- ***Write a query which is used to display highest no. of employees department from emp table by using group by clause?***

Ans:

```
select deptno, count(*) from emp group by deptno
having count(*) = (select max(count(*)) from emp group by deptno);
```

DEPTNO	COUNT(*)
30	6

Different Columns

Whenever table having hierarchal data and also when we are retrieving hierarchal data by using subquery then we are using different column name in subquery, but those columns belongs to same data type.

- ***Write a query to display the employees who are working under "BLAKE" from emp table using "empno", "mgr" columns?***

Ans: select * from emp where mgr = (select empno from emp where ename = 'BLAKE');

b) Multiple Row Subquerys

Whenever child query returns multiple values those types of subquerys are called "Multiple row subquerys".

- ***Write a query to display the employee details who are getting max(sal) in each department from emp table?***

Ans: select * from emp where sal = (select max(sal) from emp group by deptno);

Error: single row subquery returns more than one row.

This is a multiple row subquery because here child query returns multiple values. In multiple row subquerys we are using ***"in", "all", "any"*** operators.

Note: In all database systems we can also use "in" operator in single row subqueries.

Solution

SQL> select * from emp where sal in(select max(sal) from emp group by deptno);

(or)

SQL> select ename, sal, deptno from emp where sal in(select max(sal) from emp group by deptno);

ENAME	SAL	DEPTNO
TURNER	2900	30
FORD	4200	20
SCOTT	4200	10

- ***Write a query to display the employees who are working in "SALES" or "RESEARCH" department from emp, dept table?***

Ans:

```
select ename, deptno from emp
 where deptno in (select deptno from dept where dname = 'SALES' or dname = 'RESEARCH');
```

(or)

```
select ename, deptno from emp
 where deptno in (select deptno from dept where dname in ('SALES', 'RESEARCH'));
```

- ***Write a query to display the employees who are working as "Supervisors" (managers) from emp table using empno, mgr?***

Ans: select * from emp where empno in (select mgr from emp);

- ***Write a query to display the employees who are not working as "supervisors" (managers) from emp using empno, mgr?***

Ans: select * from emp where empno not in (select nvl(mgr,0) from emp);

c) **Multiple Column Subquerys**

In all relational databases we can also compare multiple column values of the child query with the multiple column of the parent query. This type of subquery is also called as "Multiple Column Subquerys".

In all relational databases in Multiple column subquerys we must specify parent query WHERE conditional columns within parenthesis "()".

*Syntax: select * from tablename where (col1, col2,.....) in (select col1, col2,..... from tablename where condition).*

- ***Write a query to display the employees whose job, mgr match with the job, mgr of the employee "SCOTT" from EMP table by using multiple column subquerys?***

Ans: select * from emp where (job, mgr) in (select job, mgr from emp where ename = 'SCOTT');

- ***Write a query to display ename, dname, sal of the employees whose salary, commission match with the salary, commission of the employees working in the location "DALLAS" from emp ,dept table by using multiple column subquery ?***

Ans:
```
select ename, dname, sal from emp e, dept d
where e.deptno = d.deptno
and (sal, nvl(comm, 0)) in
(select sal, nvl(comm, 0) from emp
where deptno = (select deptno from dept where loc = 'DALLAS'));
```

(or)

```
select ename, dname, sal from emp e, dept d
 where e.deptno = d.deptno
  and (sal, nvl(comm, 0)) in
    (select sal, nvl(comm, 0) from emp e, dept d
     where e.deptno = d.deptno
      and loc = 'DALLAS');
```

- ***Write a query to display the employees who are getting maximum salary in each department from emp table by using multiple row subquery? *****

Ans:
```
SQL>  update emp set sal = 2800 where ename= 'FORD';
      1 row updated
SQL>  select deptno, sal, ename
      from emp
      where sal in (select max(sal) from emp group by deptno);
```

DPETNO	SAL	ENAME
20	2800	FORD
30	2800	JAMES
20	5000	SCOTT
10	8550	CLARK

In all relational databases whenever child query multiple value compare with parent query group of values by using multiple row subquery then database server returns wrong result. To overcome this problem ANSI/ISO SQL introduced multiple column subqueries.

Solution

```
select deptno, sal, ename
from emp
where (deptno, sal) in (select deptno, max(sal) from emp group by deptno);
```

DPETNO	SAL	ENAME
30	2800	JAMES
20	5000	SCOTT
10	8550	CLARK

- ***Write a query to display senior most employees in each job from EMP table using multiple column subquery?***

Ans:

```
select hiredate, ename, job
from emp
where (hiredate, job) in (select min(hiredate), job from emp group by job);
```

HIREDATE	ENAME	JOB
17-DEC-80	SMITH	CLERK
20-FEB-81	ALLEN	SALESMAN
17-NOV-81	KING	PRESIDENT
02-APR-81	JONES	MANAGER
03-DEC-81	FORD	ANALYST

- ***Write a query to display the employee who are getting more salary than the highest paid employee in 20th department from emp table?***

Ans: select * from emp where sal > (select max(sal) from emp where deptno=20);

- ***Write a query to display the employees who are getting more than the lowest paid emp in 10th dept?***

Ans: select * from emp where sal > (select min(sal) from emp where deptno=10);

Note

Whenever resource table having large amount of data and also child query contains max() or min() and also when we are comparing then those type of subquery's degrades performance of the application.

To overcome this problem to improve performances of this type of query ANSI/ISO SQL. These are "ALL", "ANY". These subquerys special operators are used along with relational operators in parent query "WHERE" condition.

Example

```
SQL> select * from emp where sal > all (select sal from emp where deptno=20);

SQL> select * from emp where sal > any (select sal from emp where deptno=10);
```

If you want to display first *"n"* highest (or) lowest value for a table then we are using top-n-analysis query in all relational databases. In Oracle if you want to implement top-n-analysis query then we are using following clauses. These are ...

1. **INLINE VIEW**
2. **ROWNUM**

1. **INLINE VIEW:** Oracle 7.2 introduced "Inline views". Inline views are special type of query used in top-n-analysis. In Inline views we are using "subquerys" in place of "table name" within parent query.

*Syntax: select * from (subquery or select statement);*

Generally, in oracle we are not allowed to use "ALIAS NAME" in "WHERE" clause. In Oracle if you want to use column ALIAS NAME in "WHERE" clause then we must specified ALIAS NAME query in place of table name within parent query. This type of query is also called as subquery is used in "FROM" clause (or) "INLINE VIEW". Whenever we are using "INLINE VIEW" internally column ALIAS NAME automatically behaves like a table column name.

- ***Write a query to display the employee who are getting more than 30000 annual sal from emp table?***

Example

```
SQL> select ename, sal, sal*12 annsal from emp where annsal>30000;
Error: ANNSAL: Invalid Idenfier.
```

Solution: select * from (select ename, sal, sal*12 annsal from emp) where annsal > 30000;

ENAME	SAL	ANNSAL
JONES	2975	35700
BLAKE	2850	34200
SCOTT	3000	36000
KING	5000	60000
FORD	3000	36000

Select * from (

Ename	Sal	Annsal

) where annsal >30000;

Generally, in all relational database systems we are not allowed to use "ORDER BY" clause in child queries.

To overcome this problem if you want to execute "ORDER BY" clause prior to all other clauses then we are using "ORDER BY" clause query within "INLINE VIEW" in Oracle.

2. **ROWNUM: (Magic Column)**

- ROWNUM is a PSUEDO column (generalised column) it behaves like a table column.
- ROWNUM is used to restrict no. of rows in a table.
- Whenever we are installing oracle server then automatically some PSUEDO columns are created in oracle database. These PSEUDO columns belongs to all databases, these are ROWNUM, ROWID.
- ROWNUM is a PSUEDO column which is automatically assigned number into each row in a table at the time of selection. Generally, ROWNUM having temporarily values.

Example

SQL> select rownum, ename from emp;

ROWNUM	ENAME
1	SMITH
2	ALLEN
3	WARD

SQL> select rownum, ename from emp where deptno = 10;

ROWNUM	ENAME
1	CLARK
2	KING
3	MILLER

Generally ROWNUM PSEUDO column is used to filter rows in a table. It is also same as LIMIT clause in MYSQL database.

- **Write a query to display first row from emp table by using rownum?**

Ans: select * from emp where rownum = 1;

- **Write a query to display second row from emp table by using rownum?**

Ans: select * from emp where rownum = 2;
O/p: No rows selected
Generally, rownum doesn't work with more than "1" +ve integer i.e. it works with <, <= operators.

- **Write a query to display first 5 rows from emp table by using rownum?**

Ans: select * from emp where rownum <= 5;

Note: *Whenever we are requesting data from the table by using ROWNUM oracle server retrieve data from the table based on following algorithm.*

```
SQL> select * from emp where rownum = 2;
o/p: no row selected
```

Algorithm

```
rownum=1
for i in(select * from emp)
loop
   if(rownum=2) then
      output record;
      rownum=rownum+1;
   end if;
end loop;
```

- **Write a query to display first five highest salary employees from emp by using rownum?**

Ans: select * from (select * from emp order by sal desc) where rownum <= 5;

- **Write a query to display 5th highest salary employee from emp table by using rownum?**

Ans: select * from (select * from emp order by sal desc) where rownum <= 5
minus
select * from (select * from emp order by sal desc) where rownum <= 4;

- **Write a query to display rows between 1 to 5 from emp table by using "Rownum"?**

Ans: select * from emp where rownum between 1 and 5;

- **Write a query to display rows between to 5 to 9 from table?**

Ans: SQL> select * from emp where rownum between 5 and 9;
O/p: No rows selected

Solution
SQL> select * from emp where rownum <= 9 minus select * from emp where rownum <= 5;

- **Write a query to display 2nd record from emp using rownum?**

Ans: select * from emp where rownum <=2 minus select * from emp where rownum <= 1;

- **Write a query to display last 2 records from emp table using rownum?**

Ans: select * from emp where rownum <= 14 minus select * from emp where rownum <= 12;

Solution
Select * from emp minus select * from emp where rownum <= (select count(*) - 2 from emp);

Why rownum=2 doesn't work?

ROWNUM is a PSEUDO column which is used to filter data in a table i.e. ROWNUM is not a column it is special column which gets value only at runtime. The value of ROWNUM gets incremented by 1 after fetching the row data. That's why rownum doesn't works with some operators.

ROWNUM PSEUDO column automatically assign number to each row in a table at a time of selection and also ROWNUM PSEUDO column is used to filter data in a table.

***Note**: Oracle 7.2 onwards whenever we are using "ALIAS NAME" for "ROWNUM" in "INLINE VIEW" then that ALIAS NAME works with all SQL operators because whenever we are creating any ALIAS NAME in any value the ALIAS NAME behaves like an exact table column.*

- ***Write a query to display 2nd record from emp table using "rownum" alias name?***

Ans: select * from (select rownum r, ename, job, sal from emp) where r=2;

R	ENAME	JOB	SAL
2	ALLEN	SALESMAN	1400

***Note**: If we want to display all the columns from the table using ROWNUM ALIAS NAME then we must use **tablename.star** i.e. [e.*] along with ROWNUM ALIAS NAME after SELECT statement. Otherwise create an ALIAS NAME for the table and use "**alias name**.*" after SELECT along with ROWNUM ALIAS NAME.*

```
SQL> select * from (select rownum r, emp.* from emp) where r=2;
                    (or)
SQL> select * from (select rownum r, e.* from emp e) where r=2;
```

R	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
2	7499	ALLEN	SALESMAN	7698	20-FEB-81	1700	300	30

- ***Write a query to display 2nd, 3rd, 5th, 7th, 8th rows from emp table using rownum alias name?***

Ans: select * from (select rownum r, e.* from emp e) where r in (2, 3, 5, 7, 8);

- ***Write a query to display records between 5 to 9 from emp table using rownum alias name?***

Ans: select * from (select rownum r, emp.* from emp) where r between 5 and 9;

- ***Write a query to display first and last records from emp table using rownum alias name?***

Ans: select * from (select rownum r, emp.* from emp) where r=1 or r=(select count(*) from emp);

- ***Write a query to display even number of records from emp table using rownum alias name?***

Ans: select * from(select rownum r, emp.* from emp) where mod(r,2)=0;

- ***Write a query to display 5th highest salary employee from emp table by using rownum alias name?***

Ans: select * from (select rownum r, ename, sal from (select * from emp order by sal desc)) where r=5;

- ***Write a query to display nth highest salary employee from emp table by using rownum alias name?***

Ans: select * from (select rownum r, ename, sal from (select * from emp order by sal desc)) where r=&n;

Enter value for n: 1

- ***Write a query to display skip first 5 rows from emp table using rownum alias name?***

Ans: select * from (select rownum r, emp.* from emp) where r>5;

Analytical functions are used in inline views

Oracle 8i introduced "Analytical Functions". Analytical Functions are similar to "Group functions" but group functions always reduce number of rows in each group, where as "Analytical Functions" doesn't reduces number of rows in each group and also analytical functions execute each and every row in a table and also analytical function performance is very high compare to all other functions in databases.

Example: Using Group functions

SQL> select deptno, avg (sal) from emp group by deptno;

DEPTNO	AVG(SAL)
10	2916.66667
20	2175
30	1566.66667

Example: Using Analytical Functions

SQL> select deptno, avg(sal) over(partition by deptno) from emp;

DEPTNO	AVG(SAL)
10	2916.66667
10	2916.66667
10	2916.66667
20	2175
20	2175
20	2175

Oracle having following Analytical functions.

1. **Row_number()**
2. **Rank()**
3. **Dense_rank()**

These three analytical functions automatically assigns "RANKS" to each row in a table either "GROUP BY" or rows wise in a table.

Syntax: *AnalyticalFunctionName() over ([partition by <columnname>] order by <columnname> [asc/desc]);*

ROW_NUMBER() analytical function automatically assigns different rank numbers when values are same, Where as ***RANK()*** and ***DENSE_RANK()*** analytical functions automatically assigns same rank numbers when values are same but ***RANK()*** skips next consecutive rank numbers (1,1,3,4,5) where as ***DENSE_RANK()*** doesn't skips next consecutive rank numbers.

Example

Sql > select deptno, ename, sal, row_number() over(partition by deptno order by sal desc) r from emp;

Row_number()

DEPTNO	ENAME	SAL	R
20	SCOTT	4200	1
20	FORD	4200	2
20	JONES	3200	3

Rank()

Sql > select deptno, ename, sal, rank() over(partition by deptno order by sal desc) r from emp;

DEPTNO	ENAME	SAL	R
20	SCOTT	4200	1
20	FORD	4200	1
20	JONES	3200	3

Dense_rank()

Sql > select deptno, ename, sal, dense_rank() over(partition by deptno order by sal desc) r from emp;

DEPTNO	ENAME	SAL	R
20	SCOTT	4200	1
20	FORD	4200	1
20	JONES	3200	2

- ***Write a query which is used to display employees in each department and also automatically assign rank in each department salary wise descending order from emp table by using analytical function?***

Ans: select * from (select deptno, ename, sal, row_number() over(partition by deptno order by sal desc) r from emp) where r<=10;

- ***Write a query to display highest salaries to lowest salaries in each department from emp table using row_number() analytical function?***

Ans: select deptno, ename, sal, row_number() over (partition by deptno order by sal desc) r from emp;

- ***Write a query to display 2nd highest salary employee in each department from emp table using analytical function? ******

Ans: select * from (select deptno, ename, sal, dense_rank() over(partition by deptno order by sal desc) r from emp) where r=2;

DEPTNO	ENAME	SAL	R
10	CLARK	7750	2
20	JONES	3200	2
30	MARTIN	2750	2

- ***Write a query to display 5th highest salary employee from emp table using analytical function? ******

Ans: select * from (select deptno, ename, sal, dense_rank() over(order by sal desc) r from emp) where r=5;

Note: In analytical functions "Partition by" clause is an optional clause.

- ***Write a query to display nth highest salary employee from emp table using analytical function? ******

Ans: select * from (select deptno, ename, sal, dense_rank() over(order by sal desc) r from emp) where r = &n;

Enter value for n: 1;

DEPTNO	ENAME	SAL	R
10	MILLER	8400	1

ROWID is a PSEUDO column it behaves like a table column, ROWID store physical address of a row in a table.

In Oracle whenever we are inserting row into a table then oracle server internally automatically generates a UNIQUE identification number for identifying a record uniquely in a hexa decimal format. This is also called as "ROW ADDRESS" or "ROWID".

Generally, in Oracle by using "ROWID" we can retrieve data very fastly from oracle database, that's why "ROWID" query improve performance of the application. Generally "ROWNUM" has temporary values whereas "ROWID" has fixed values.

Example

```
SQL> select rownum, rowid, ename from emp;

SQL> select rownum, rowid, ename from emp where deptno=10;
```

Note: In Oracle, by default rowid's are in ascending order, that's why we can also use min(), max() in rowid.

- ***Write a query to display 1st row from emp table using rowid?***

Ans: select * from emp where rowid = (select min(rowid) from emp);

- ***Write a query to display last record from emp table using rowid?***

Ans: select * from emp where rowid = (select max(rowid) from emp);

Note: We can also use rowid in ORDER BY clause within analytical function.

- ***Write a query to display 2nd record from emp table using analytical function, rowid from emp table?***

Ans: select * from (select deptno, ename, sal, row_number() over(order by rowid) r from emp) where r = 2;

- ***Write a query to display last 2 records from EMP table using rowid, row_number, analytical function?***

Ans:
```
select * from (select deptno, ename, sal,
row_number() over(order by rowid desc) r from emp) where r <= 2;
```

DEPTNO	ENAME	SAL	R
10	MILLER	1300	1
20	FORD	3000	2

- ***Write a query to display each department 2nd row from emp table using analytical function, rowid?***

Ans:
```
select * from (select deptno, ename, sal,
row_number() over(partition by deptno order by rowid) r from emp) where r = 2;
```

DEPTNO	ENAME	R
10	KING	2
20	JONES	2
30	WARD	2

Note: In oracle if you want to delete duplicate row from a table, then we must use ROWID.

- ***Write a query to display duplicate data from the following table?***

```
SQL> create table test (sno number (10));

SQL> insert into test values (&sno);
Enter values for sno: 10
Enter values for sno: 10
Enter values for sno: 10
Enter values for sno: 20
Enter values for sno: 20
Enter values for sno: 30
Enter values for sno: 30
Enter values for sno: 40
Enter values for sno: 50
SQL> select * from test;
```

SNO
10
10
10
20
20
30
30
40
50

```
SQL> select sno, count (*) from test group by sno having count (*) >1;
```

SNO	COUNT(*)
30	2
20	2
10	3

- ***Write a query to delete duplicate rows from the above table by using rowid (or)***
- ***Delete duplicate rows except one row from the above table using rowid?***

Ans:

```
SQL> delete from test where rowid not in (select min (rowid) from test group by sno);
SQL> select * from test;
```

SNO
10
20
30
40
50

Delete duplicate rows in a table

In all relational databases delete duplicate rows except one row in each group is also called as deleting duplicate rows from a table. In oracle by using ROWID only we are allow to delete duplicate rows in a table.

Example:

```
SQL> delete from test where rowid not in (select max (rowid) from test group by sno);
```

SNO
10
20
30
40
50

Note: For improve performance of a query we can also use ROWID in order by clause within analytical function, because ROWID retrieve very fast data from database and also analytical function use more performance in database application.

These two analytical functions are used to compare current row value with previous or following row values.

Lag(): It display previous row data along with current row data. This function accepts 3 parameters and also here last 2 parameters are optional.

Syntax: *lag (colname, offset, default value) over (partition by <colname> order by <colname> [asc/desc]);*

Here offset returns integers, these integers starts with 1 onward, here offset 1 represent 1st previous value and offset 2 represent 2nd previous value.

If previous value is not available then oracle server returns null value. In place of null value we can also substitute default value by using 3rd parameter.

Example

```
SQL> select ename, sal, lag(sal) over(order by sal desc) prev_sal from emp;
```

ENAME	SAL	PREV_SAL
MILLER	8400	
CLARK	7750	8400
KING	5100	7750
SCOTT	4200	5100

```
SQL> select ename, sal, lag(sal, 1, 0) over(order by sal desc) prev_sal from emp;
```

ENAME	SAL	PREV_SAL
MILLER	8400	0
CLARK	7750	8400
KING	5100	7750
SCOTT	4200	5100

Lead(): It display next row value along with current row value. These function also except 3 parameters.

Syntax: *lead (colname, offset, defaultvalue) over (partition by colname order by colname [asc/desc]);*

Example

```
SQL> select ename, sal, lead(sal,1,0) over(order by sal desc) next_sal from emp;
```

ENAME	SAL	PREV_SAL
MILLER	8400	7750
CLARK	7750	5100
KING	5100	4200
SCOTT	4200	0

Generally in non-correlated subquery "child query" is executed first then only "parent query" is executed. Whereas in correlated subquerys "parent query" is executed first then only "child query" is executed.

Generally, in non-correlated subquery "child query" is executed only once per "parent query" table whereas in correlated subquery internally "child query" is executed for each and every row for "parent query" table.

In all relational database systems when we are using correlated subquery then we must create ALIAS NAME for the "parent query" table within "parent query" and then we must use this ALIAS NAME in child query "WHERE" clause.

Syntax: *select * from tablename aliasname where columnname = (select * from tablename where columnname = (any operator) aliasname.columnname);*

Whenever we are submitting correlated subquery into oracle server then oracle server get a candidate row from the "parent query" table and then control passed into "child query" WHERE condition and also based on evaluation value, it compare with "parent query" WHERE condition.

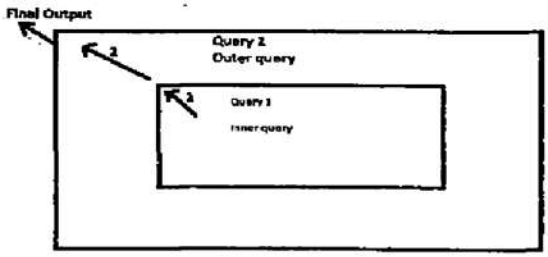
Generally, correlated subquery are used in demoralization process in this process we are using correlated updates i.e. by using correlated updates we are modifying one table column values based on another related table.

Syntax: *update tablename1 aliasname1 set columnname = (select columnname from tablename2 aliasname2 where aliasname1.commoncolumnname = aliasname2.commoncolumnname);*

Example

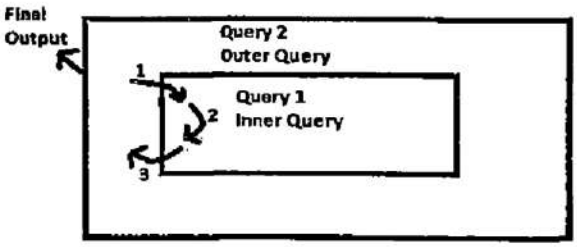
```
SQL> alter table emp add dname varchar2 (10);
SQL> update emp e set dname= (select dname from dept d where e.deptno=d.deptno);
        14 rows updated
SQL> select * from emp;
```

Non-Correlated Subquery



Uni Directional

Correlated Subquery



By Directional

- ***Write a query to display 1st highest salary employee from emp table using correlated subquery?***

Ans: select * from emp e1 where 1 = (select count(*) from emp e2 where e2.sal >= e1.sal);

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

- ***Write a query to display 2nd highest salary employee from following table using correlated sub query?***

Ans:
```
SQL> create table test(ename varchar(20), sal number(10));
SQL> insert into test values('&e', &s);

Enter the values for e: abc
Enter the values for s: 100
SQL> /
Enter the values for e: xyz
Enter the values for s: 150
SQL> /
Enter the values for e: zzz
Enter the values for s: 200
SQL> /
Enter the values for e: bbb
Enter the values for s: 300

SQL> select * from test;
```

ENAME	SAL
abc	100
xyz	150
zzz	200
bbb	300

```
SQL> select * from test e1 where 2 = (select count(*) from test e2 where e2.sal >= e1.sal);
```

Execution Process

Phase 1

Step 1: get a candidate row [where cursor pointer points a record is called "Candidate Row"]
(first row ..... abc 100)

Step 2: select count(*) from test e2 where e2.sal >= 100; 4

Step 3: select * from test e1 where 2=4; (False)

Phase 2

Step 1: get a candidate row (xyz 150)

Step 2: select count(*) from test e2 where e2.sal >=150; 3

Step 3: select * from test e1 where 2=3; (False)

Phase 3

Step 1: get a candidate row (zzz 200)
Step 2: select count(*) from test e2 where e2.sal >=200; 2
Step 3: select * from test e1 where 2=2; (True)
Output: zzz 200

Note: When where condition returns true then only that candidate row return into result.

Note: Whenever resource table column having duplicate data and also when we are try to retrieve data based on duplicate value by using above query then oracle server doesn't return any result.
To overcome this problem we must use "DISTINCT" clause within "Count()" function.

Example

```
SQL> insert into test values('shailendra',200);
SQL> select * from test;
```

Output

ENAME	SAL
abc	100
xyz	150
zzz	200
bbb	300
shailendra	200

```
SQL> select * from test e1 where 2=(select count(*) from test e2 where e2.sal >=e1.sal);
Output: No Rows Selected
```

Solution

```
SQL> select * from test e1 where 2 =(select count(distinct(sal)) from test e2 where e2.sal >=e1.sal);
```

Output

ENAME	SAL
zzz	200
shailendra	200

Note: We can also write above query using "n-1" method. In this case we are not allowed to use "=" operator in "Where" condition of "Child Query".

Example

```
SQL> select * from test e1 where (2-1) = (select count(distinct(sal)) from test e2 where e2.sal > e1.sal);
```

- ***Write a query to display n^{th} highest salary employee from emp table using "Correlated Subquery" ?***

Ans: select * from emp e1 where &n = (select count (distinct (sal)) from emp e2 where e2.sal >=e1.sal);

Enter the value for n : 1

ENAME	SAL
KING	5000

- ***Write a query to display the employees who are getting more salary than the average salaries of their job from emp using correleated subquery?***

Ans: select * from emp e where sal > (select avg(sal) from emp where job=e.job);

EXISTS operator always returns "Boolean values" either "TRUE" or "FALSE". In all relational databases we can also used in "CORRELATED SUBQUERY". EXISTS operator performance is very high compare to "IN" operator. EXISTS operator is used in "WHERE" condition of the "parent query" only and also when we are using EXISTS operator then we are not allowed to use "column name" along with EXISTS operator in "WHERE" condition of the parent query.

Syntax: *Select * from tablename aliasname where exists (select * from tablename where columnname = aliasname.columnname);*

In all database systems if we want to test whether one table values are available or not available in another table columns then we must use correlated subquery "Exists Operator".

Note: Exists Operator is used to test whether a given set is empty (or) non-empty. EXISTS operator on non-empty set returns "true" whereas EXISTS operator on empty set returns "false".

Example 1: exists {1, 2, 3, 4, 5} = true
Example 2: exists { } = false

- ***Write a query to display those departments from dept table having employees in emp table by using correlated subquery exists operator?***

Ans: select * from dept d where exists (select * from emp where deptno = d.deptno);
Output:

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO

- ***Write a query to display those departments from dept table having employees in emp table by using non- correlated subquery in operator?***

Ans: select * from dept where deptno in (select deptno from emp);

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO

Note: EXISTS operator performance is very high compare to "IN" operator.

- **Difference between IN, EXISTS operator?**

Example

Dept ----> 1000 rows
Emp ----> 1000 rows

IN	EXISTS
SQL> select * from dept where deptno in (select deptno from emp); **Execution:** All rows in emp table will read for every row in dept table i.e. oracle server reads 10, 00,000 rows from emp table.	SQL> select * from dept d where exists (select * from emp where deptno = d.deptno); **Execution:** A maximum of one row will be read for emp table for each row in dept table i.e. it reduces processing stats then improve performance of the query.

In all relational databases "EXISTS" operator give more performance than the "IN" operator. Whenever we are testing first table column values are available in another related table.

Example

Non-Correlated Subquerys

```
SQL> select * from dept where deptno in (select deptno from emp);
```

Correlated Subquerys

```
SQL> select * from dept d where exists (select * from emp where deptno = d.deptno);
```

In the above example whenever we are using non-correlated subquery for dept table deptno 10 then oracle server will see all deptno in emp table whether the match is found or match is not found.

Where as in correlated subquerys for dept table deptno 10 then EXISTS operator returns true, when at list one match is found in emp table and also oracle server doesn't check remaining number of departments in emp table, that's why in this case exists operator gives more performance.

- ***Write a query to display those departments from dept table doesn't have employees in EMP table using correlated subquery?***

Ans: select * from dept d where not exists (select * from emp where deptno = d.deptno);

DEPTNO	DNAME	LOC
40	OPERATIONS	BOSTON

- ***Write a query to display those department from dept table doesn't have employees in emp table using non-correlated subquerys?***

Ans: select * from dept where deptno not in (select deptno from emp);

DEPTNO	DNAME	LOC
40	OPERATIONS	BOSTON

Note: In all relational datbases "NOT IN" operator doesn't work with NULL values, whereas "NOT EXISTS" operator works with NULL value. So overcome from this problem we are using "NOT EXISTS" operator along with "correlated subquery".

```
SQL> select * from emp;
SQL> select * from dept where deptno not in (select deptno from emp);
        No Rows Selected
SQL> select * from dept d where not exists (select * from emp where deptno = d.deptno);
```

DEPTNO	DNAME	LOC
40	OPERATIONS	BOSTON

Note: In Oracle, whenever we are testing one table column values are available (or) not available in another table using non correlated and correlated queries. This type of querys is also called as "Anti Joins".

Example select * from dept where deptno not in (select deptno from emp);

- ***Write a query to display the employees who are getting same salary as SCOTT salary from emp table using correlated subquery exists operator?***

Ans: select * from emp e1 where exists (select * from emp e2 where e2.ename= 'SCOTT' and e2.sal=e1.sal);

Output:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20

Note

In relational databases we can also use EXISTS operator in non-correlated subquerys. Whenever we are using EXISTS operator in non-correlated subquery then database server checks child query return data or not.

Whenever child query returns data then only EXISTS operator return true. Whenever child query doesn't returns any data then EXISTS operator return false in non-correlated subquery.

Subquery special operators are used in non-correlated subquery (All, Any) (Multiple Row Subquery)

- ***Write a query to display the employees who are getting more salary than the highest paid employees in 20th dept from emp table?***

Ans: select * from emp where sal > (select max (sal) from emp where deptno = 20);

- ***Write a query to display the employees who are getting more salary than the lowest paid employees in 10th dept from emp table?***

Ans: select * from emp where sal > (select min (sal) from emp where deptno = 10);

In all relational databases relational operators compares only one value at a time. If you want to compare these relational operators with multiple values at a time then ANSI/ISO SQL provided subquery special operators. These subquery special operators are ...

a) All
b) Any

These operators are used along with relational operator in parent query where condition.

Example

1. select * from emp where sal > all (select sal from emp where deptno = 20);
2. select * from emp where sal > any (select sal from emp where deptno = 10);

IN: - It returns same values in the list (child query)
ALL: - It satisfies all values in the list (child query)
ANY: - It satisfies any value in the list
IN → = ANY

Example

SQL> select * from emp where deptno = (10, 20);
Error: This operator must be followed by ANY or ALL

Solutions: SQL> select * from emp where deptno = any (10, 20);

- ***Write a query to display the employees who are getting more than the all salaries of the "CLERK" from emp table by using subquery special operator and also display final output salary wise desc order?***

Ans: select * from emp where sal > all (select sal from emp where job = 'CLERK') order by sal desc;

Note: Whenever we are using subquery special operator "ALL" then database servers internally uses logical operator "AND" whereas whenever we are using subquery operator "ANY" then database server internally uses logical operator "OR".

Example

SQL> select * from emp where deptno > all(10,20);
Output: 30

SQL> select * from emp where deptno > any(10,20);
Output: 20 30

Note: In all relational databases "IN" operator is also same as subquery special operator "=ANY" but "NOT IN" is not same as "<>ANY".

Example 1

SQL> select * from emp where deptno in (10, 20);
Output: 10 20
SQL> select * from emp where deptno = any (10, 20);
Output: 10 20

Example 2

SQL> select * from emp where deptno not in (10, 20);
Output : 30
SQL> select * from emp where deptno <>any (10, 20);
Output : 10 20 30

Note: Not in operator is also same as subquery operator "<>all".

Example

SQL> select * from emp where deptno not in (10, 20);
Output: 30
SQL> select * from emp where deptno <>all (10, 20);
Output: 30

View is a database object which is used to provide authority level of security.
Generally VIEW are created from tables those tables are also called as base tables.
Generally data security point of view "Database Administrator" creating VIEWS from the table and then those views given to the number of users.
Generally VIEW doesn't store data that's why VIEW is also called as "Virtual table" or "Window of a table".
Generally VIEW is created from "Base tables".

Based on the type of base tables views are categorized into two types.

1. **Simple View**
2. **Complex view (or) Join view**

1. **Simple view:** Simple view is a view which is created from only One Base tables where as "Complex view" is a view which is created from Multiple base tables.

Syntax: *create [or replace] view <view name> as select * from <table name> where condition;*

DML Operations Performed On Simple View

In oracle we can also perform DML operation through simple view to base table based on following restrictions:

1. If a simple view having GROUP FUNCTION, GROUP BY CLAUSE, ROWNUM, DISTINCT, SET OPERATORS, JOINS then we can't perform DML operation through simple views to base table.

2. We must include "Base table" NOT NULL column into the view then only we can perform "Insertion operation" through simple view to base table. Otherwise oracle server returns an error.

Example 1

```
SQL> create or replace view v1 as select * from emp where deptno = 10;
SQL> select * from v1;
SQL> insert into v1(empno,ename,deptno) values(1, 'shailendra',30);
        1 row created
SQL> select * from emp;
```

Example 2

```
SQL> create or replace view v2 as select ename, sal, deptno from emp where deptno=10;
SQL> select * from v2;
SQL> insert into v2(ename, sal, deptno) values('abc',1000,30);
```

Error: Cannot insert null into empno.

In all databases whenever we are creating a view then automatically view definition (select statement) are permanently store in database. In oracle if we want to view these definitions then we are using "user_views" data dictionary.

Note

Views also used for simplifying query purpose i.e. regularly used query, we are putting in a VIEW and whenever necessary select that VIEW when a VIEW contain FUNCTIONS or EXPRESSION then we must create ALIAS NAME for those function or expressions otherwise oracle server returns an error.

Example

SQL> create or replace view v3 as select depnto, max (sal) from emp group by deptno;
Error: must name this expression with column alias.

Solution

SQL> create or replace view v3 as select depnto, max(sal) a from emp group by deptno;
SQL> select * from v3;

DEPTNO	A
30	3350
20	3800
10	7700

Example: For viewing alias name A =?

SQL> desc user_views;
SQL> select text from user_views where view_name= 'V3';

Output

TEXT
select deptno,max(sal) a from emp group by deptno;

Note: In oracle when a view having ROWNUM then also we must create ALIAS NAME for ROWNUM.

Example

SQL> create or replace view v4 as select rownum, ename from emp;
Error: must name this expression with column alias.

Solution

SQL> create or replace view v4 as select rownum sno, ename from emp;
SQL> select * from v4;

SNO	ENAME
1	SMITH
2	MILLAR
3	JONE

SQL> desc user_views;
SQL> select text from user_views where view_name= 'V4';

Output:

TEXT
select rownum sno , ename from emp;

2. Complex View (or) Join view: Complex view is a view which is created from multiple base tables.

Example

```
SQL>  create or replace view v5
      as
      select ename,sal,dname,loc
      from emp,dept where emp.deptno=dept.deptno;

SQL> select * from v5;
```

DML Operations on Complex View

Example

```
SQL> update v5 set ename = 'abc' where ename = 'SMITH';
        1 row updated
SQL> update v5 set dname = 'xyz' where dname = 'SALES';
```

Error: Cannot modify a column which maps to a non key preserved table.

Generally, in all database systems we cannot perform DML operations through "COMPLEX VIEW" to base tables. In oracle when we are trying to perform DML operations through "COMPLEX VIEW" to base table then some table columns are affected and some other are not affected. In Oracle, if we want to view affected, unaffected columns then we are using "user_updatable_columns" data dictionary.

Example

```
SQL> desc user_updatable_columns;
SQL> select column_name, updatable from user_updatable_columns where table_name= 'V5';
```

COLUMN_NAME	UPDATABLE
ENAME	YES
SAL	YES
DNAME	NO
LOC	NO

Generally in Oracle, also we cannot perform DML operations through Complex Views to base table. To overcome this problem Oracle 8.0 introduced INSTEAD of triggers in PL/SQL.

Whenever we are creating INSTEAD of trigger in this COMPLEX VIEW then only we are allow performing DML operations through COMPLEX VIEW to base table, by default INSTEAD of trigger are ROW LEVEL trigger and INSTEAD of trigger are created on VIEW.

Trigger is also same as "Stored Procedures" and also it will automatically INVOKE whenever a DML operations performed on a table.

All database systems having 2 types of triggers.

1. **Statement Level Triggers**
2. **Row Level Triggers**

In Statement level trigger, triggers body is executed only once per DML statements. Where as in Row level trigger, trigger body is executed for each row for DML statements.

Syntax:

Create or replace trigger triggername
before/after Insert/Update/Delete on tablename
[for each row] --→ row level trigger

```
begin
-----------------
-----------------
-----------------
end;
```

Trigger body

Difference between statement level and row level triggers:

Example: SQL> create table test (col1 date);

Statement Level Trigger

```
create or replace trigger tv1
after update on emp
begin
insert into test values (sysdate);
end;
/
```

Testing

```
SQL> update emp set sal = sal+100 where deptno = 10;
     3 rows updated
SQL> select * from test;
SQL> delete from test;
SQL> drop trigger tv1;
```

Row Level Triggers

```
create or replace trigger tv2
after update on emp
for each row
begin
insert into test values(sysdate);
end;
/
```

Testing

```
SQL> update emp set sal=sal+100 where deptno=10;
3 rows updated
SQL> select * from test;
SQL> drop trigger tv2;
```

Output

COL1
24-APR-15
24-APR-15
24-APR-15

Row Level Triggers

In Row level triggers, triggers body is executed per each row for DML statements, that's why we are using for each row clause within trigger specification and also DML transactional values internally storing TWO "ROLL BACK" segment qualifiers. These are...

a) **Old**
b) **New**

These buffers are used in either in trigger specification or in the trigger body, when we are using the buffer in body then we must use column (:) in front of the buffer name.

Syntax: - :old.columnname
Syntax: - :new.columnname

	INSERT	UPDATE	DELETE
:NEW	✓	✓	✗
:OLD	✗	✓	✓

- ***Write a PL/SQL row level trigger on emp table whenever user deleting data then automatically those deleted records are stored in another table?***

Ans:

```
SQL> create table test as select * from emp where 1=2;

SQL> select * from test;
SQL> desc test;
SQL> create or replace trigger tg1
     after delete on emp for each row
     begin
     insert into test values(:old.empno, :old.ename, :old.job, :old.mgr, :old.hiredate, :old.sal,
     :old.comm, :old.deptno);
end;
```

Testing

```
SQL> delete from emp where sal>2000;
SQL> select * from test;
```

Instead of Trigger

Generally, we can't perform "DML operations" through complex view to "Base table". To overcome this problem Oracle 8.0 introduced "Instead of trigger" in PL/SQL. By default "Instead of Triggers" are "Row lever Triggers" and also "Instead of trigger" are created on "Views".

Syntax:

```
Create or replace trigger triggername
instead of insert/update/delete on viewname
for each row

begin
——
——
——
end;
```

Example

```
SQL> select * from v5;
SQL> desc user_updatable_columns;
SQL> select COLUMN_NAME, UPDATABLE from user_updatable_columns where table_name='v5';
```

COLUMN_NAME	UPDATABLE
ENAME	YES
SAL	YES
DNAME	NO
LOC	NO

Solution (By using instead of triggers)

```
create or replace trigger tv1
instead of update on v5
for each row
begin
update dept set dname=:new.dname where dname=:old.dname;
update dept set loc=:new.loc where loc=:old.loc;
end;
/
```

Testing

Example 1: SQL> update v5 set dname='xyz' where dname='SALES';

Example 2:

```
SQL> desc user_updatable_columns;
SQL> select COLUMN_NAME, UPDATABLE from user_updatable_columns where table_name= 'V5';
```

COLUMN_NAME	UPDATABLE
ENAME	YES
SAL	YES
DNAME	YES
LOC	YES

- Oracle 8i introduced "materialized view".
- Materialized view is handled by database administrator.
- Materialized view is used in data warehousing applications.
- Generally, "VIEW" doesn't store data where as "Materialized view" store data.
- Generally materialized view is used to improve performance of the "Joins" or "Aggregatable Querys".
- Materialized view stores replication of the remote database into local node.
- Materialized view stores data same like a table, but whenever we are refreshing "materialized view" it synchronizes data based on "base table".

Syntax: create materialized view <view name> as <select statement>;

Difference between View and Materialized View

View	Materialized View
• A view doesn't store data.	• Materialized view stores data.
• Security purpose.	• Improve performance purpose.
• When we are dropping "Base table" then view cannot be accessible.	• If we are dropping "Base table" also Materialized view can be accessible.
• We can perform DML operations on view.	• We cannot perform DML operations on Materialized view.

In oracle before we are creating materialized view then database administrator must give **"create any materialized view"** privilege to user, otherwise oracle server returns an error "insufficient privilege".

Syntax: *Grant create any materialized view to username;*

Example

```
SQL> conn scott/tiger
SQL> create materialized view mn1 as select * from emp;
Error: Insufficient Privileges.
```

Solution

```
SQL> conn sys as sysdba
        Enter password: sys
SQL> grant create any materialized view to scott;
SQL> conn scott/tiger
        Connected
SQL> create materialized view mn1 as select * from emp;
        Materialized View Created.
```

Note: In Oracle sometimes VIEW also returns insufficient privileges error. To overcome from this problem database administrator must give "create any view" privilege to users.

Syntax: *Grant create any view to username;*

Example

```
SQL> create table test (sno number (10));
SQL> create materialized view v1 as select * from test;
Error: Insufficient privileges

SQL> conn sys as sysdba/sys
SQL> grant create any materialized view to scott;
SQL> conn scott/tiger
SQL> create or replace view v1 as select * from test;
        View created
```

In oracle one of the MATERIALIZED VIEW based table must have a PRIMARY KEY then only create MATERIALIZED VIEW among those table, otherwise oracle server returns an error.

Example

```
SQL> create table test (sno number (10));
SQL> create materialized view mc1 as select * from test;
Error: table "TEST" does not contain a primary key constraint.
```

Note: In oracle 11g, 12c we can also create materialized view when base table doesn't have primary key.

Example

```
SQL> create table base (sno number (10) primary key, name varchar2 (20));
SQL> insert into base values ('&s', '&n');
SQL> select * from base;
SQL> commit;
SQL> create or replace view v1 as select * from base;
SQL> create materialized view mv1 as select * from base;
```

In this case MATERIALIZED VIEW is also same as VIEW because whenever we are creating VIEW then automatically VIEW definitions are permanently stored in database.

Whenever we are creating materialized view definitions are automatically stored in database same like a VIEW definition. In Oracle, if we want to view MATERIALIZED VIEW definitions then we are using "user_mviews" data dictionary.

Example

```
SQL> desc user_mviews;
SQL> select query from user_mviews where mview_name= 'MV1';
SQL> select rowid, sno, sname from base;
SQL> select rowid, sno, name from v1;
```

Execution

Whenever we are creating a VIEW then automatically VIEW definitions are permanently stored in a database. Whenever we are requesting VIEW each and every time query definition are executed, that's why each and every time base table is affected that's why VIEW doesn't improve performance of the query.

Whenever, we are creating a MATERIALIZED VIEW then Oracle server automatically stores MATERIALIZED VIEW definitions in a "Data Dictionary" and also oracle server automatically stores query result within MATERIALIZED VIEW.

Whenever user requesting MATERIALIZED VIEW by using "SELECT" statement then Oracle server each and every time retrieves data from MATERIALIZED VIEW. In this case oracle server doesn't execute query definition that's why in this case base table are not affected, that's why a MATERIALIZED VIEW improves performance of the query.

Whenever we are refreshing MATERIALIZED VIEW then only oracle server executes MATERIALIZED VIEW definitions, In this case only base tables are affected.

Example SQL> select rowid, sno, name from mj1;

In this case MATERIALIZED VIEW "ROWID" is different from base table "ROWID" that's why MATERIALIZED VIEW stored data.

SQL> select * from base;

Sno	name
1	a
2	b
3	c
4	d

SQL> update base set name = upper (name)
SQL> select * from base;

SNO	NAME
1	A
2	B
3	C
4	D

MATERIALIZED VIEW also stores data same like a table but when we are refreshing MATERIALIZED VIEW its synchronized database on base table. If we want to refresh MATERIALIZED VIEW then we are using refresh procedure from dbms_mview package.

Syntax: exec dbms_mview.refresh ('materlaliazed view name');

Example

SQL> exec dbms_mview.refresh ('mj1');
SQL> select * from mj1;

SNO	NAME
1	A
2	B
3	C
4	D

Oracle having two types of Materialized View

1. **Complete refresh materialized view**
2. **Fast refresh materialized view**

1. **Complete Refresh Materialized View**

In Oracle by default Materialized View are "Complete Refresh Materialized View". This type of Materialized View does not give more performance when we are refreshing Materialized View number of times, because in this Materialized View whenever we are refresheing Materialized View internally ROWID are Recreated if we are not modifying data in base table also.

To overcome this problem to improve more performance of query then oracle introduced "Fast Refresh Materialized View".

Syntax: *Create materialized view <view name> refresh complete as <select statement>;*

Example

```
SQL> select rowid, sno, name from mv1;
SQL> exec dbms_mview.refresh ('mv1');
SQL> select rowid, sno, name from mv1;
        [Here rowid's are changed]
```

2. **Fast Refresh Materialized View**

These types of Materialized View are also called as Incremental Refresh Materialized View. These Materialized View performance is very high compare to "Complete Refresh Materialized View" because in these Materialized View ROWID are never changed when we are refreshing Materialized View number of times also.

Syntax: *create materialized view <view name> refresh fast as <select statement>;*

Before we are using "Fast Refresh Materialized View" it needs a mechanism to capture any changes made to its base table, this refreshment is also called as "Materialized View Log". That's why before creating "Fast Refresh Materialized View" we must create Materialized View Log on "base table".

Syntax: *Create materialized view log on basetablename;*

Example: SQL> select * from base;

SNO	NAME
1	A
2	B
3	C
4	D

```
SQL> create materialized view log on base;
SQL> create materialized view mk1 refresh fast as select * from base;
SQL> select rowid, sno, name from mk1;
SQL> update base set name= 'xy' where sno=1;
SQL> exec dbms_mview.refresh ('mk1');
SQL> select rowid, sno, name from mk1;
        [Here rowid's are not changed only data is affected]
```

On demand / on commit

In Oracle we can refresh materialized views in 2 ways.

1. **Manually**
2. **Automatically**

1. **Manually:** In manual method we are refreshing materialized view by using "**dbms_mview**" package. This method is also called as "**On Demand**" method. In oracle by default method is "On demand".

2. **Automatically:** We can also refresh materialized view without using "**dbms_mview**" package. This method is also called as "**On Commit**" method.

Syntax: create materialized view <view name> <refresh complete / refresh fast>
<on demand / on commit> as <select statement>;

Example

```
SQL> create materialized view mk2 refresh fast on commit as select * from base;
SQL> select * from mk2;
```

SNO	NAME
1	XY
2	B
3	C
4	D

```
SQL> update base set name = 'zzz' where sno = 2;
SQL> select * from base;
```

SNO	NAME
1	XY
2	ZZZ
3	C
4	D

```
SQL> select * from mk2;
```

SNO	NAME
1	XY
2	B
3	C
4	D

```
SQL> commit;
SQL> select * from mk2;
```

SNO	NAME
1	XY
2	ZZZ
3	C
4	D

Generally Materialized View are used to improves performance of the "Join" & "Aggregate Query" where as "VIEW" are used to provide security i.e. In all database systems, if we want to restrict table columns from one user into another user then only we are creating "VIEW" with the required column then only those "VIEW" given to the number of users by using "GRANT" command.

1. **Grant** (Viewing permissions)
2. **Revoke** (Cancel permissions)

Creating a User

```
SQL> conn sys as sysdba
     Enter Password: sys
```
(or) SQL> conn system/manager

Syntax: Create user username identified by password;
Syntax: Grant connect, resource to username; (or) DBA
Syntax: conn username/password;

Creating a user

```
SQL> conn sys as sysdba;
     Enter password: sys
SQL> create user murali identified by murali;
SQL> grant connect, resource to murali;
SQL> conn murali/murali;
SQL> select * from emp;
Error: table or view does not exists;

SQL> conn scott/tiger;
SQL> grant all on emp to murali;
SQL> conn murali/murali;
SQL> select * from emp;
Error: table (or) view does not exists.

SQL> select * from scott.emp;
SQL> create synonym ab for scott.emp;
SQL> select * from ab;
```

Privilege

Privilege is a right given to another user whenever we are giving privileges (permission) then only those users allow performing some operations of the database. Data security points of view all database systems having two types of user privileges. They are:

1. **System Privileges**
2. **Object Privileges**

1. **System Privileges**

These privileges enable users to perform actions in database. These priveleges are given by database administrator only. Whenever administrator giving there privileges to no. of user then only no. of users allow performing particular actions in the database, otherwise oracle server returns an error insufficient privileges.

Oracle having more than 80 system privileges these are: Create session, create table, create any view, create procedure, create trigger, create any materialized view, create any index...

Syntax: grant system privileges to username1, username2...

Example

```
SQL> conn sys as sysdba;
Enter password: sys
SQL> grant create procedure, create trigger, create any materialized view to scott, shailendra;
```

Note: Oracle having created system privilege which is used to connect clients (users) to the oracle database.

Example

```
SQL> conn sys as sysdba;
Enter password: sys

SQL> create user shailendra identified by shailendra;
SQL> conn shailendra/shailendra;
Error: user shailendra locks create session privileges; logon denied
```

Solution

```
SQL> conn sys as sysdba;
Enter password: sys
SQL> grant create session to shailendra;
SQL> conn shailendra / shailendra;
      Connected to database .....
SQL> show user;
SQL> user is shailendra
```

ROLE

Role is nothing but collection of either system privileges or collection of object privilege.

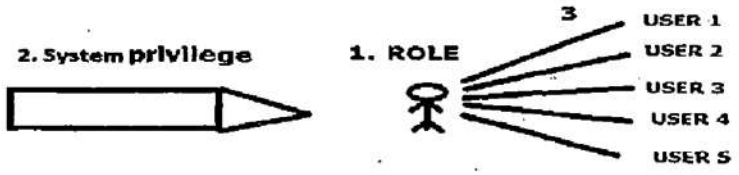
Roles are created by database administrator only. In a multi user environment, number of users works on same project. In this case, sometime number of users requires common set of privileges. In this case only database administrator creating a user defined role and then assigns set of privileges into role and then only that role given to the number of users.

All database system having two types of role:

a) **User Defined Roles**
b) **Predefined Roles**

a) **User Defined Roles**

This role is creating by DBA only. In multiuser environment numbers of user's works on same project, in this case some user requires common set of privileges. In this case only database administrator creates a role and assigns common set of privileges role and then only that role given to the number of users.



Step 1: Create a role

Syntax: create role rolename;

Step 2: Assign system privileges to rolename

Syntax: grant system privilege to rolename;

Step 3: Assign rolename to number of users.

Syntax: grant rolename to username1, username2,....;

Example:

```
SQL> conn sys as sysdba
     Enter password: sys
SQL> create role r1;
     Role created
SQL> grant create any materialized view, create trigger, create procedure to r1;
SQL> grant r1 to scott, murali, shailendra;
```

In Oracle, if we want to view all the system privileges related to role then we are using "role_sys_privs" data dictionary.

Example

```
SQL> desc role_sys_privs;
SQL> select role, privilege from role_sys_privs where role= 'R1';
```

ROLE	PRIVILEGE
R1	CREATE TRIGGER
R1	CREATE ANY MATERIALIZED VIEW
R1	CREATE PROCEDURE

b) **Predefined Roles:** Whenever we are installing oracle server then automatically 3 predefined roles are created. They are:

1. Connect -> used by end users
2. Resource -> used by developers
3. DBA -> used by database administrator

Note: In Oracle, connect role internally having create session privilege. This privilege is used to users allowed to connect to the Oracle Server.

Example

```
SQL> desc role_sys_privs;
SQL> select role, privilege from role_sys_privs where role in ('connect', 'resource');
SQL> select role, privilege from role_sys_privs where role in('DBA');
```

2) **Object Privilege**

Object Privileges are given by either database developers or database administrator.

Whenever we are using object privileges then only users allow performing some operation of the objects.

Oracle has select, insert, update, delete, execute, read, write object privileges.

Note: In oracle we can also use all keyword in place of select, insert, update, delete, object privileges.

Syntax: grant object privilege on objectname to usernames/rolename/public;

```
SQL> conn scott/tiger;
SQL> grant all on emp to murali;
SQL> grant all on emp to r1;
SQL> grant all on emp to public;
```

Note: In Oracle, if you want to view object privileges related to user then we are using **"user_tab_privs"** data dictionary.

Example: SQL> desc user_tab_privs

REVOKE This command is used to "cancel" system or privileges from users.

Syntax: Revoke system privileges from user1,user2,....;
Syntax: Revoke object privileges on objectname from username1, username2, ...;

Generally in all databases if you want to restrict table columns from one users to another user then only we are using views i.e. we are creating a view with the required column then only that view given to the number of users by using object privileges.

Syntax: grant object privileges on viewname to username1,usernme2,..... ;

```
SQL> conn scott/tiger
SQL> create or replace view v1 as select empno, ename, sal, deptno from emp;
SQL> select * from v1;
SQL> grant all on v1 to murali;
SQL> conn murali/murali;
SQL> select * from scott.v1;
```

Note: In all database through the views we can achieve to logical data independence i.e. whenever we are adding new column into existing table it is not affected within view in the external level.

```
SQL> conn scott/tiger;
SQL> alter table emp add address varchar2 (10);
SQL> select * from emp;
SQL> conn murali/murali;
SQL> select * from scott.v1;
```

Uses of views in all databases

1. Views provides security
2. Simplifying complex query
3. We can achieve logical data independence

With Check Option

If we want to provide constraint type mechanism on views then only we are using with check option clause. When the views having check option clause then we are not allowed to insert other than where condition values through views to base table.

Syntax: create or replace view viewname as select * from tablename where condition with check option;

Example

```
SQL> create or replace view v4 as select * from emp where deptno=10 with check option;
SQL> select * from v4;
```

Testing

```
SQL> insert into v4 (empno, ename, deptno) values (1, 'abc', 30);
Error: view with check option where clause violated.
SQL> insert into v4(empno,ename,deptno) values(1, 'abc',10);
        1 Row created
SQL> select * from emp;
```

Read Only Views

When a views having with read only clause then those types of views is called read only views. In Read only views we cannot perform DML operations. These views are used for reporting purpose.

Syntax: Create or replace view viewname as select * from tablename where condition with read only;

Example

```
SQL> create or replace view v3 as select * from emp with read only;
SQL> delete from v3 where deptno=10;
Error: cannot delete from view.
```

Force View (or) Forced View

In Oracle, we can also create view without using base tables. These types of views are also called as "Force Views".

Syntax: create or replace force view viewname as select * from anyname;

Example

```
SQL> create or replace force view v2 as select * from welcome;
Warning: view created with compilation errors.
SQL> create table welcome (sno number(10));
SQL> alter view v2 compile;
SQL> desc v2;
```

Note: In all databases throw the views we can achieve logical data independence i.e. whenever we are adding a column into the table it is not affected in view within external level.

Example

```
SQL> conn scott/tiger;
SQL> grant all a v1 to murali;
SQL>Alter table emp add address varchar (10);
SQL> select * from emp;

SQL> conn murali/murali;
SQL> select * from scott v1;
```

We can also drop view by using drop view viewname

Example

```
SQL>drop view v1;
View dropped
```

Synonym is a database object which provides security because synonym hides another schema username, object name. Synonym is an ALIAS NAME (or) reference name of original object. Synonym also created by database administrators.

All database systems having 2 types of synonyms:

1. **Private Synonyms**
2. **Public Synonyms**

In all database systems by default Synonyms are "Private Synonyms".

Syntax: create synonym <synonym name> for username.objectname@database link;

In Oracle before we are creating **Public Synonyms** then we are using "**Create Public Synonym**" system privilege to user otherwise oracle server returns as "Insufficient Privilege Error".

Syntax: grant create public synonym to username;
Syntax: create public synonym <synonym name> for username. objectname@database link name;

scott/tiger	SQL> conn murali/murali	SQL> conn a1/a1
SQL> grant all on emp to murali, a1;	SQL> select * from scott.emp; SQL> create synonym abc for scott.emp; SQL> select * from abc;	SQL> select * from scott.emp; SQL> select * from abc; SQL> select * from xy;
	SQL> create public synonym xy for scott.emp; Error: Insufficient Privilege	
	SQL> conn sys as sysdba; Enter password: sys	
	SQL> grant create public synonym to murali; SQL> conn murali/murali; SQL> create public synonym xy for scott.emp; SQL> select * from xy;	

SQL> drop synonym xy;
Synonym dropped.

We can also drop synonyms by using "**drop synonym <synonym name>**"
All Synonyms information has stored under "**user_synonyms**" data dictionary.

<u>Example</u> SQL> desc user_synonyms;

SEQUENCE is a database object which is used to generate SEQUENCE of integers automatically.
Generally SEQUENCE database object is used to generate PRIMARY KEY values automatically.
Generally SEQUENCE is an independent database object once a SEQUENCE has been created then number of users simultaneously accesses that SEQUENCE database object.

Syntax:

```
Create sequence <sequence name>
  [ START WITH <value> ]
  [ INCREMENT BY <value> ]
  [ MINVALUE <value> ]
  [ MAXVALUE <value> ]
  [ CYCLE / NOCYCLE ]
  [ CACHE / NOCACHE ];
```

In Oracle, if we want to generate SEQUENCE of number or if you want to access SEQUENCE database object then oracle provided following 2 PSEUDO columns for SEQUENCE database object. These are...

1. CURRVAL
2. NEXTVAL

```
Syntax (currval) : <sequence name>.currval;
Syntax (nextval) : <sequence name>.nextval;
```

These PSEUDO columns are used in INSERT, UPDATE, DELETE, SELECT statements. In Oracle if we want to generate SEQUENCE values by using sequences PSEUDO column through SELECT statement then we must use DUAL table.

```
Syntax: select <sequence name>.currval from dual;
Syntax: select <sequence name>.nextval from dual;
```

Example

```
SQL> create sequence S1
       start with 5
       increment by 2
       maxvalue 100;

SQL> select S1.currval from dual;

Error: Sequence S1.CURRVAL is not yet defined in this session.
```

In Oracle if we want to generate first sequence number then we must use "NEXTVAL" PSEUDO column because "CURRVAL" PSEUDO column always returns current value of the sequence session. If sequence session already having a value.

In all database systems we are using "CURRVAL" PSEUDO column after using "NEXTVAL" PSEUDO columns only.

Example

```
SQL> select s1.nextval from dual;
        5
SQL> select s1.nextval from dual;
        7
SQL> select s1.nextval from dual;
        9
SQL> select s1.nextval from dual;
        11
SQL> select s1.currval from dual;
        11
```

Note: Always "NEXTVAL" PSEUDO column uses globalization session, where as "CURRVAL" PSEUDO column uses local session.

Example

SQL> connect scott/tiger Session 1	SQL> connect scott/tiger Session 2
• SQL> select s1.nextval from dual; 1	• SQL> select s1.nextval from dual; 4
• SQL> select s1.nextval from dual; 2	• SQL> select s1.nextval from dual; 5
• SQL> select s1.nextval from dual; 3	• SQL> select s1.nextval from dual; 6
• SQL> select s1.currval from dual; 3	• SQL> select s1.currval from dual; 6

In oracle all sequences information stored under "**user_sequences**" data dictionary.

```
SQL> desc user_sequences;
```

In oracle sequences we can also change sequences parameter values by using "**ALTER**" command but we cannot change starting "SEQUENCE" number.

Syntax: *alter sequence <sequence name> <parameter name> <new value>;*

Example

```
SQL> Create sequence S1
       start with 5
       increment by 1
       minvalue 4
       maxvalue 100;
SQL> select S1.nextval from dual;
        5
SQL> select S1.nextval from dual;
        6
SQL> select S1.nextval from dual;
        7
SQL> select S1.nextval from dual;
        8
```

```
SQL> alter sequence S1
        Increment by -1;

        Sequence altered.....

SQL> select S1.nextval from dual;
        7
SQL> select S1.nextval from dual;
        6
SQL> select S1.nextval from dual;
        5
SQL> select S1.nextval from dual;
        4
Error: Sequence S1.NEXTVAL goes below MINVALUE and cannot be instantiated

SQL>alter sequence S1
        Start with 8;
Error: cannot alter starting sequence number
```

Note: In sequence START WITH value cannot be less than **MINVALUE.**

Example

```
Create sequence S1
Start with 4
Increment by 1
Minvalue 5;
Error: Start with cannot be less than minvalue.
```

Generating Primary Key Value Automatically

In oracle if you want to generate PRIMARY KEY value automatically then we were using SEQUENCE database.

Example

```
SQL> Create table test (sno number (10) primary key, name varchar2 (10));
SQL> Create sequence S1;
SQL> Insert into test (sno, name) values (S1.nextval, '&name');

Enter value for name: a
SQL> /
Enter value for name: b
SQL> /
Enter value for name: c
SQL> Select * from test;
```

SNO	NAME
1	a
2	b
3	c

```
SQL> alter table test add rno number (10);
SQL> select * from test;
```

SNO	NAME	RNO
1	a	
2	b	
3	c	

```
SQL> create sequence S2
        Start with 1001;
SQL> update test set rno = S2.nextval;
        3 rows updated
SQL> select * from test;
```

SNO	NAME	RNO
1	a	1001
2	b	1002
3	c	1003

CACHE

CACHE is a memory area which is used to pre-allocate set of SEQUENCE number. CACHE memory area is used to access SEQUENCE value very fast. Generally when we are using CACHE optional clause in SEQUENCE database object then those type of SEQUENCE improve performance of the application.

In oracle whenever we are creating a SEQUENCE then automatically that SEQUENCE is created in hard disk. Whenever we are requesting SEQUENCE by using sequence PSEUDO column then client tool first checks requested SEQUENCES is available in CACHE memory area or not. If requested SEQUENCE is not available in cache memory area then oracle server fetches SEQUENCES name from hard disk into CACHE memory area and also based on SEQUENCE PSEUDO column SEQUENCE number is transfer from hard disk into CACHE memory area then only oracle server return this SEQUENCE value from cache memory area into client tool.

Whenever we are requesting SEQUENCE more number of time then this default process degrades performance because in this case SEQUENCE value are generated very slowly.

To overcome this problem to access SEQUENCE values very fast then oracle provided CACHE optional clause in database object. When we are defining CACHE optional clause in SEQUENCE database object then oracle server pre-allocates set of SEQUENCE number in CACHE memory area. Whenever we are requesting SEQUENCES values by using SEQUENCE PSEUDO columns then oracle server directly retrieve SEQUENCE values from the CACHE memory area not from the hard disk, that's why this process automatically improves performance of the SEQUENCE because this process reduces disk i/o (input/output).

Note: Whenever system crashes (or) power failure then automatically CACHE values are lost.

In Oracle by default CACHE value is 20 and also CACHE minvalue are 2.

Example

```
create sequence s1
start with 1
increment by 1
cache 1
```

Error: The number of values to CACHE must be greater than 1.

In oracle sequences information are stored under "user_sequences" data dictionary.

```
SQL> desc user_sequences;
```

Index is a database object which is used to retrieve data very fast from database, that's why indexes are used to improvement of query.
Generally indexes are created on table columns and also indexes are created by database administrator.
In all database we are creating Indexes in 2 ways:

1. **Automatically**
2. **Manually**

1. **Automatically:** Whenever we are creating "PRIMARY KEY" or "UNIQUE KEY" then oracle servers automatically creates "B-TREE" indexes on those columns.
2. **Manually:** We can also create "Indexes" explicitly by using "create index" command.

In Oracle, we are creating index explicitly by using following syntax:

Syntax: Create index <index name> on tablename (col1, col2 ...);

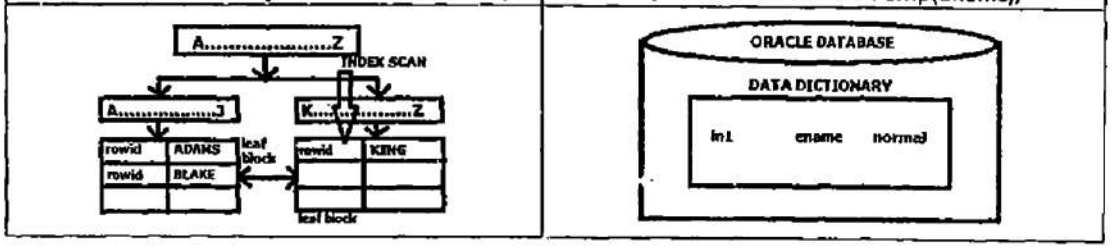
Note:
Whenever we are requesting data by using "WHERE" clause or "ORDER BY" clause then only Oracle server searching for indexes.
Whenever "WHERE" (or) "ORDER BY" clause columns having indexes then oracle server uses index scan mechanism for retrieving data very fast from the database. If that column doesn't have indexes then oracle server internally uses full table scan for retrieving data from the database.
In Oracle whenever "WHERE" clause contains <> NOT EQUAL TO (or) IS NULL (or) IS NOT NULL operator then oracle server doesn't search for indexes, If those columns already having indexes also.

Oracle having 2 types of Indexes:

1. B-TREE Indexes
2. BIT Map Indexes

1. B-Tree **Indexes:** In Oracle, by default indexes are B-TREE indexes. Whenever we are creating B-TREE indexes then oracle server automatically creates B-TREE structure based on indexed columns values, in this B-TREE structure always leaf blocks stores actual data along with ROWID.
   Whenever user requesting data by using indexes columns in "WHERE" clause then oracle server automatically uses indexes scan of the B-TREE structure for retrieving data very fast from the leaf blocks. Whenever we requesting data by using "WHERE" clause, if "WHERE" clause columns doesn't have any indexes then oracle server internally uses full table scan for retrieving data from database.

Developers	Data Base Administrators
SQL> select * from emp where ename= 'KING';	SQL> create index in1 on emp(ename);



Note: In Oracle all indexes information stored under "user_indexes" data dictionary.

Example

```
SQL> create index in1 on emp (name);
SQL> desc user_indexes;
SQL> select index_name, index_type from user_indexes where table_name= 'EMP';
```

Output

INDEX_NAME	INDEX_TYPE
PK_EMP	NORMAL
IN1	NORMAL

Note: In Oracle if you want to view column names along with Index name then we are using "user_ind_columns" data dictionary.

Example

```
SQL> desc user_ind_columns;
SQL> select index_name, column_name from user_ind_columns where table_name= 'EMP';
```

Output

INDEX_NAME	COLUMN_NAME
PK_EMP	EMPNO
IN1	ENAME

Calculating performance of a query in oracle through plan table

Step 1: Use an explain plan for clause before select query by using following syntax:

Syntax: explain plan for <select statement>;

Whenever we are submitting this query oracle server internally creates plan table based on query execution. If you want to view plan table then we are using display function from **"dbms_xplan"** package by using following syntax.

Syntax: select * from table (dbms_xplan.display ());

Step 2: (display plan table)

Syntax: select * from table (dbms_xplan.display);

Example: (without using Indexes)

```
SQL> select * from emp where ename = 'KING';
```

Testing Performance

```
SQL> explain plan for select * from emp where ename= 'KING';
SQL> select * from table (dbms_xplan.display ());
```

Example (With Using Indexes)

```
SQL>create index in1 on emp (ename);
SQL> select * from emp where ename= 'KING';
```

Testing

```
SQL> explain plan for select * from emp where ename = 'SCOTT';
SQL> select * from table (dbms_xplan.display ());
```

Function Based Indexes

Oracle 8i introduced "Function Based Indexes" by default "Function Based Indexes" are B-TREE indexes. In oracle whenever we are requesting data by using functions (or) expressions in "WHERE" clause then oracle server doesn't search for Indexes if those columns already having "Indexes" also.

To overcome this problem Oracle 8i introduced extension of the B-TREE indexes called "Function Based Indexes" which is used to create Indexes on columns along with functions (or) expressions, then only Oracle server searching for indexes.

Syntax: create index <index name> on tablename (function name (column name or expression));

Example (Without using functions based Indexes)

```
SQL> select * from emp where upper (ename) = 'KING';
```

Testing

```
SQL> explain plan for select * from emp where upper (ename) = 'KING';
SQL> select * from table (dbms_xplan.display());
```

[Here oracle server does not search for indexes]

Example (With using function based Indexes)

```
SQL> create index in2 on emp (upper (ename));
SQL> select * from emp where upper (ename) = 'KING';
```

Testing Performance

```
SQL> explain plan for select * from where upper (ename) = 'KING';
SQL> select * from table (dbms_xplan.display());
SQL> desc user_indexes;
SQL> select index_name, index_type from user_indexes where table_name = 'EMP';
```

INDEX_NAME	INDEX_TYPE
IN2	FUNCTION BASED INDEXES

Virtual Columns

Oracle 11g introduced "Virtual Columns" in a table which stores "Stored Expression" directly into Oracle database, prior (before) to 11g we can also store "stored expressions" into oracle database indirectly by using VIEWS, FUNCTION based indexes. In oracle 11g we can also store stored expression directly in oracle database by using "**generated always as**" clause in virtual column.

Syntax: columnname datatype (size) generated always as (stored expression) [virtual];

Example

```
SQL> create table test (a number (10), b number (10), c number (10) generated always as (a+b) virtual);
SQL> insert into test (a, b) values (20, 40);
SQL> select * from test;
```

A	B	C
20	40	60

Note: *In oracle if you want to view "Virtual Column" expression then we are using "data_default" property from "user_tab_columns" data dictionary.*

Example

```
SQL> desc user_tab_columns;
SQL> select column_name, data_default from user_tab_columns where table_name = 'TEST';
```

COLUMN_NAME	DATA_DEFAULT
C	"A+B"

Oracle has two types of b-tree indexes.

a) **Non Unique B-Tree Indexes**
b) **Unique B-Tree Indexes**

In oracle by default automatically created Indexes are "Unique b-tree Indexes" and also all explicit Indexes are "Non unique b-tree Indexes" by default "Unique b-tree Indexes" performance is very high compare to "Non unique b-tree Indexes". We can also create "Unique b-tree Indexes" explicitly by using following syntax.

Syntax: Create unique index <index name> on tablename (columnname);

Note: We are not allowed to create unique indexes on duplicate value columns.

Example

```
SQL> create unique index in3 on emp (ename);
Index created
SQL> create unique index in4 on emp (job);
ERROR: cannot CREATE UNIQUE INDEX: duplicate key found.
```

2. **Bit Map Index:** Oracle 7.3 introduced Bit Map Indexes. Bit Map Indexes are used in data ware housing applications. Generally Bit Map Indexes are created on "Low Cardinality Columns".

Syntax: create bit map index <index name> on tablename (column name);

$$\text{CARDINALITY OF A COLUMN} = \frac{\text{NUMBER OF DISTINCT VALUES}}{\text{TOTAL NUMBER OF VALUES}}$$

Example

Cardinality of empno = 14/14 = 1 (high cardinality) -> b-tree
Cardinality of job = 5/14 = 0.367.... (Low cardinality) -> Bit map indexes

Whenever we are creating a "Bit Map Indexes" internally oracle server automatically creates a "Bit Map Table" by using no of bits based on the indexed column values.

Whenever user requesting data by using logical operators (or) equality operator then oracle server directly operates bits within bitmap table then resultant bitmap automatically convert into rowid by using an internal bitmap function.

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

```
SQL> create bitmap index abc on emp(job);
SQL> desc user_indexes;
SQL> select index_type, index_name from user_indexes where table_name= 'EMP';
```

Output

INDEX_TYPE	INDEX_NAME
NORMAL	PK_EMP
BITMAP	ABC

In Oracle, we can also drop Index by using

Syntax: drop index <index name>;

Note: In all databases through the indexes we can achieve physical data independences i.e. whenever we are adding an index at internal level, it is not affected in structure of the table within conceptual level but performance will be affected.

Set operators are used to retrieve data from single (or) multiple tables. These operators are also called as "Vertical Joins".

1. **Union:** - It returns unique values and also automatically sorting data.
2. **Union all:** - It returns unique + duplicate data. (no automatic sorting)
3. **Intersect:**- It returns common values
4. **Minus:** - Values are in first query those values are not in second query.

Example

```
select job from emp where deptno=10
union
select job from emp where deptno=20;
```

JOB
ANALYST
CLERK
MANAGER
PRESIDENT

Note: Whenever we are using set operators always corresponding expressions must belong to same data type & also set operators always returns first query column and also names as column headings.

Example

```
select dname from dept
union
Select ename from emp;
```

Note: Using set operators we can also retrieve data from multiple querys when corresponding expression not belongs to same datatype also in this case we must use appropriate type conversion function.

Example

```
select deptno from emp
union
select dname from dept;
```

Error: expression must have same data type as corresponding expression

Solution

```
select deptno "deptno", to_char(null) "deptnames" from emp
union
Select to_number(null),dname from dept;
```

Output

deptno	deptnames
10	
20	
30	
	ACCOUNTING
	OPERATIONS
	RESEARCH
	SALES

Converting one datatype into another datatype is called "Conversions".

Oracle has 2 types of conversions.

1. **Implicit conversions**
2. **Explicit conversions**

1. **Implicit Conversions:**

A. In Oracle when an expression contains string representing pure number (eg: '1234') then oracle server automatically converts "String type" in "Number type".

Example: SQL> select sal+ '100' from emp;

B. In Oracle whenever we are passing number into character functions then oracle server automatically converts number type into character type.

Example

SQL> select length (1234) from dual;
Output: 4

C. In Oracle whenever we are passing DATE string into DATE functions then oracle server automatically converts "Data String" into "Date Type". But here passed parameter must be in "Default Date Format".

Example

SQL> select last_day ('12-aug-05') from dual;
Output: 31-aug-05

Implicit Conversion Table			
From	To	Assignment	Expression Evaluation
Varchar2 (or) char	Number	Yes	Yes
Varchar2 (or) char	Date	Yes	Yes
Number	Varchar2 (or) char	Yes	No
Date	Varchar2 (or) char	Yes	No

2. **Explicit Conversions:** Using "Explicit conversions functions" we can also convert one datatype into another datatype explicitly. Oracle having following explicit conversion function.

Decode

- It is a conversion function which is used to decoding the values.
- Decode function is also same as "if.... then ... else if" constructor in PL/SQL.
- Decode function internally used equality operator (=).

Syntax: Decode (columnname, value1, statement1, value2, statement2 ...);

Example

SQL> select decode (1, 2, 3, 4, 5, 6, 7, 8) from dual;
Output: 8 [In matching 1 with any pair not occurred then print unpaired value i.e. else part i.e. 8]

SQL> select decode (1, 2, 3, 1, 5, 6, 7) from dual;
Output: 5 [Match 1 with 5 pair]

SQL> select decode (1, 2, 3, 1, 5, 1, 7) from dual;
Output: 5 [Matched 1 with 5 & 7 pairs value but first pair value will return]

SQL> select decode (1, 2, 3, 4, 5, 6, 7) from dual;
Output: null [Not matched 1 with any pair]

Example

SQL> select ename, sal, deptno, decode (deptno, 10, 'ten', 20, 'twenty','others') from emp;

Output

ENAME	SAL	DEPTNO	DECODE
SMITH	800	20	twenty
ALLEN	1600	30	others
WARD	1250	30	others
CLARK	2450	10	ten

Decode is used to convert number into string or character into string conditionally.

Note: If we want to modify data conditionally within a table using "SQL" then we must use "Decode ()".

- ***Write a query to update "commission" of the employees in emp table based on following condition?***
  1. If job= 'CLERK' then update comm into 10% of sal.
  2. If job= 'SALESMAN' then update comm into 20% of sal.
  3. If job= 'ANALYST' then update comm is 30% of sal
  4. Else comm is 40% of sal.

Ans:

```
Update emp set comm=decode (
                    job, 'CLERK', sal*0.1,
                    'SALESMAN', sal*0.2,
                    'ANALYST', sal*0.3, sal*0.4
                    );
```

Pivoting

In all relational databases if you want to display aggregate values in tabular form and also rows converted into columns then we are using decode conversion function within GROUP BY, this type of reports are also called as pivot report.

Example: select job, sum (sal) from emp group by job;

JOB	SUM(SAL)
CLERK	5750
SALESMAN	4800
PRESIDENT	5200
MANAGER	8875
ANALYST	6400

Converting Into Pivoting Reports

Example

```
select job,
    sum(decode(deptno, 10, sal)) "deptno 10",
    sum(decode(deptno, 20, sal)) "deptno 20",
    sum(decode(deptno, 30, sal)) "depnto 30"
 from emp
group by job;
```

Output

JOB	DEPTNO 10	DEPTNO 20	DEPTNO 30
CLERK	1300	1900	950
SALESMAN			5600
PRESIDENT	5000		
MANAGER	2450	2975	2850
ANALYST		6000	

Example

```
select dname,
    sum(decode(job, 'CLERK', 1, 0)) "CLERKS",
    sum(decode(job, 'SALESMAN', 1, 0)) "SALESMAN",
    sum(decode(job, 'ANALYST', 1, 0)) "ANALYST"
 from emp e, dept d
 where e.deptno = d.deptno
 group by dname;
```

Output

DNAME	CLERKS	SALESMAN	ANALYST
ACCOUTING	1	0	0
RESEARCH	2	0	2
SALES	1	4	0

Oracle 8.0 introduced "Case statement" and also "Oracle 8i" introduced case conditional statement.
Case conditional statement is also called as "Searched Case Statement".
Case statement is also used to "Decoding" the values.
Case Statement performance is very high compare to decode conversion function.

Note: Decode conversion function internally used "Equality operator" (=), where as in "Case statement" we can also use all SQL operator (<,>, <=, >=, <>, like, is, etc...) explicitly.

Method 1: (Case statement)

Syntax:

```
case columnname
when value1 then
    statement1
when value2 then
    statement2
-------------------
-------------------
else
    statements
end;
```

Example

```
select ename, sal,deptno,
    case deptno
     when 10 then 'ten'
     when 20 then 'twenty'
     else
       'others'
    end
 from emp;
```

Output

ENAME	SAL	DEPTNO	CASE
CLARK	2850	10	ten
SCOTT	3400	20	twenty
JAMES	2250	30	others

Grouping_id()

This function is used in **ROLLUP, CUBE** querys. This function always accepts GROUP BY clause columns and also returns numbers.

Example

Grouping_id (deptno, job) ====> 1 2 3

Here 1 represent 1st grouping column subtotal and also 2nd represent 2nd grouping column subtotal and also 3rd represent grand total.

Example

```
select deptno, job, sum(sal), grouping_id (deptno, job) from emp
group by cube(deptno, job)
order by 1, 2;
```

Note: We can also use grouping_id() within case statement which is used to return messages to the subtotal, grand total.

Example

```
select deptno,job,sum(sal) as sumsal,
    case grouping_id(deptno, job)
     when 1 then
      'dept subtotal'
     when 2 then
      'job subtotal'
     when 3 then
      'grand total'
    end as subtotal
 from emp
group by cube(deptno, job)
order by 1, 2;
```

Method 2: (Case Conditional Statement (or) Searched Case Statement)

Syntax:

```
Case
when <column condition1> then
        <statement1>
when <column condition2> then
        <statement2>
-------------------------------
-------------------------------
else
        <statements>
end;
```

Example

```
select ename,sal,
    case
     when sal < 1000 then
        'Low Salary'
     when sal between 1000 and 3000 then
        'Medium Salary'
     when sal in (3100, 3200, 3500, 3800) then
        'Special Salary'
     else
        'Other Salary'
    end
 from emp;
```

Note: Decode conversion function is an oracle standard whereas case statement is an ANSI standard.

Oracle 11g introduced pivot function. Pivot function is used to convert rows into columns and also displays aggregate function value in tabular form. Pivot function performances very high compare to decode conversion function.

Syntax:

```
select * from (select col1, col2, col3..... from < table name >)
pivot(< aggregate function name() > for < column name > in (value1, value2,.....));
```

Example: 11G

```
select * from (select job, sal, deptno from emp)
pivot(sum(sal)
for deptno in(10 as deptno10, 20 as deptno20, 30 as deptno30));
```

Output:

JOB	Deptno10	Deptno20	Deptno30
CLERK	2000	3000	1430
SALESMAN			2800
PRESIDENT	5300		
MANAGER	2730	3075	2930
ANALYST		6200	

```
select * from (select job, deptno from emp)
pivot(count(*)
for deptno in(10 as deptno10, 20 as deptno20, 30 as deptno30));
```

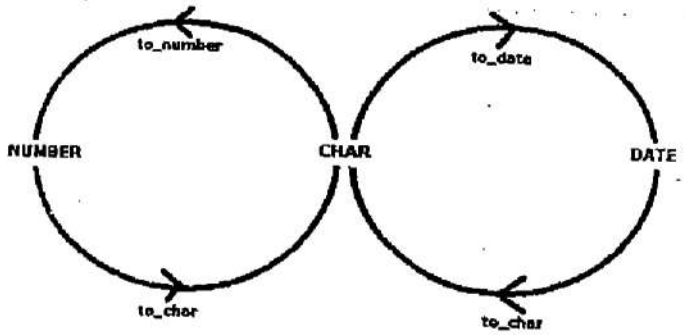
Output:

JOB	Deptno 10	Deptno 20	Deptno 30
CLERK	1	2	1
SALESMAN	0	0	4
PRESIDENT	1	0	0
MANAGER	1	1	1
ANALYST	0	2	0

Data Type Conversion Function:

Oracle also having following explicit conversion functions.

1. **to_number()**
2. **to_char()**
3. **to_date()**



Converting number into words ***

In oracle to_char having JSP format which takes Julian date and then spell out.

Example

select to_char(sysdate, 'jsp') from dual;

Output: two million four hundred fifty-eight thousand seven hundred six

If we want to view Julian date then we are using format **'J'** within **to_char** function.

Example

select to_char(sysdate, 'j') from dual;

output: 2458706

Julian date

In oracle Julian date is a number of days since jan1, 4712 BG Julian date always displays as number. These Julian dates are used in oracle to convert a **numbers** into **words** by using **to_char, to_date** functions.

If we want to convert any number into words then first we must convert given numbers into Julian date.

If we want to convert any number into **Julian date** then we must use format **'J'** with **"to_date"** function then only that Julian date converted into word by using **"to_char"** JSP format.

Example:

select to_char(to_date(1234, 'J'), 'JSP') from dual;

Output: One Thousand Two Hundred Thirty-Four

- **Write a query which is used to convert user entered number into word by using to_char, to_date()?**

Ans: Sql>select to_char(to_date(&no, 'J'), 'JSP') from dual;

Output:

Enter value for no: 786

Seven hundred eighty six

1. **to_number:-** This function is used to converting a "String" representing numeric value with format into numeric value without format.

Example

SQL> select '$35.5'+5 from dual;
Error: invalid number

Solution

SQL> select '35.5'+5 from dual;
O/P: 40.5

Example

SQL> select to_number ('$35.5') +5 from dual;
Error: Invalid number

SQL> select to_number ('35.5', '99.9') +5 from dual;
Error: Invalid number format model.

Solution

SQL> select to_number ('$35.5', '$99.9') +5 from dual;
O/P: 40.5

Note: Whenever we are using "**to_number()**" then we must use second parameter as same as first parameter format then only oracle server automatically convert string type into number type. In this second parameter we are using oracle predefined format element.

Not for other character

Example

SQL> select to_number ('a35.5', 'a99.9') +5 from dual;
Error: Invalid number format model

2. **to_char():** to_char is an over loading function which is used to convert "number type" into "character type" and also used to convert "date type" into "character type" or "date string".

Converting "Number type" into "Character type"

- ***WAQ which is used to substitute a message in place of null value in MGR columns?***

Ans: SQL> select nvl(mgr, 'no manager') from emp;
Error: Invalid number

Solution: SQL> select nvl(to_char(mgr), 'no manager') from emp;

MGR
7902
7698
7698
No manager
7698

Syntax: to_char (number, 'character format')

Oracle provided following predefined format elements for **to_number, to_char** function. These predefined format elements is used in second parameter of the **to_char, to_number** function.

Format elements	Meanings
9	Representing Number
G	Group Separator (,)
D	Decimal Indicator (.)
L	Local Currency
$	Dollar ($)
0	Leading zero
,	Comma
.	Decimal

Example: Group Seperator (,)

```
SQL> select to_char(1234567, '99g99g999') from dual;
O/P: 12,34,567
```

Note: Whenever we are using "**to_char**" first parameter exceeds then the second parameter format elements numbers of 9 then oracle server return # symbol.

Example

```
SQL> select to_char(28345, '9999') from dual;
O/P: #####
```

Combination of both g & d:

Example

```
SQL> select to_char(1234567, '99g99g999d99') from dual;
O/P: 12,34,567.00
```

L -> Local Currency: (Default currency dollar)

Example

```
SQL> select to_char (123, 'L999') from dual;
O/P: $123
```

NLS (National Language Support)

National language supports allows users to intract with the database in there native language and also it is use to allow application to run in different language environment. These NLS setting are handling by database administrator only.

Example

```
select to_char(sysdate, 'DAY MONTH'),
       to_char(sysdate, 'DAY MONTH', 'nls_date_language = french')
 from dual;
```

TO_CHAR()	TO_CHAR()
THURSDAY APRIL	JEUDI AVRIL

In oracle if you want to view predefined NLS parameter name and also those parameter default values then we are using "nls_session parameters" dictionary.

Example

SQL> desc nls_session_parameters;

NAME
PARAMETER
VALUE

SQL> select PARAMETER, VALUE from nls_session_parameters;

PARAMETER	VALUE
NLS_LANGUAGE	AMERICAN
NLS_CURRENCY	$
NLS_DATE_FORMAT	DD-MON-RR
NLS_DATE_LANGUAGE	AMERICAN

In oracle database administrator setting date format through NLS parameter by using following syntax:

Syntax: alter session set nls_date_format= 'any date format';

Example

SQL> select sysdate from dual;
O/P: 04-APR-19

SQL> alter session set nls_date_format= 'DD/MM/YYYY HH: MI: SS';

SQL> select sysdate from dual;
O/P: 04/04/2019 08:20:31

SQL> alter session set nls_date_format= 'DD-MON-YY';

Note: We can also use these predefined NLS parameter name within 3rd parameter of the "to_number", "to_char", "to_date" function. In these function 3rd parameter is an optional parameter.
Here 3rd parameter must be specified within "single quote" and also in 3rd parameter NLS parameter value is defined by using following syntax:

Syntax: nlsparametername = value
Syntax: to_char (number, 'format', '[nlsparameter]')

Example

SQL> select to_char(123, 'L999') from dual;
O/P: $123

Note: In oracle by default local currency is $, if you want to display other than this default local currency then we are using "nls_currency" parameter within 3rd parameter of the "to_char" function.

Example

SQL> select to_char(123, 'L999' , 'nls_currency=Rs') from dual;
O/P: Rs 123

SQL> select ename, to_char(sal, 'L99g99g999', 'nls_currency = Rs') from emp;

Output:

ENAME	TO_CHAR(SAL)
SMITH	Rs 800
ALLEN	Rs 1,400

Zero (0) -> Leading Zero:

Example

```
SQL> select to_char (123, '0999') from dual;
O/P: 0123
SQL> select to_char (123, '9990')) from dual;
O/P: 123
```

$ (Dollar Sign):

```
SQL> select to_char (123, '$999') from dual;
O/P: $123
```

Converting date type into character type

Syntax: to_char (date, 'format', '[nls parameter]')

Whenever we are using "to_char" function always first parameter must be oracle date type.

Example

```
SQL> select to_char(to_date('26/05/09', 'DD/MM/YY'), 'DAY/MONTH/YYYY') from dual;
O/P: TUESDAY  /MAY    /2009
```

Whenever we are using "to_char" function and also when we are using DAY format then oracle server internally automatically allocates maximum 9 bytes of memory in DAY field.

Whenever passed day having less than 9 bytes then oracle server automatically generates gapes.
To overcome this problem for filling gapes then we are using **FILL MODE (FM).**

Solution

```
SQL> select to_char(to_date('26/05/09', 'DD/MM/YY'), 'FMDAY/MONTH/YYYY') from dual;
O/P: TUESDAY/MAY/2009
```

3. **to_date ():** It is used to convert "Date String" into "Date Type".

Example

```
SQL> select '19-DEC-05'+1 from dual;
Error: Invalid number
```

Solution

```
SQL> select to_date ('19-DEC-05') +3 from dual;
Output: 22-DEC-05
```

Oracle having following null value functions

1. nvl()
2. nvl2()
3. nullif()
4. coalesce()

1. **nvl() :-** It is used to substitute (or) replace users value in place of null.

Syntax: nvl (exp1,exp2)

2. **nvl2() :-** Oracle 9i introduced nvl2() this functions excepts 3 parameters.

Syntax: nvl2 (exp1, exp2, exp3)

Here if expression 1 is null then it returns exp3, otherwise it returns exp2.

Example

SQL> select nvl2(null,20,30) from dual;
O/P: 30
SQL> select nvl2(10,20,30) from dual;
O/P: 20

3. **nullif() :-** Oracle 9i introduced nullif() this functions accepts 2 parameter.

Syntax: null if(exp1, exp2)

Here if exp1 is same as exp2 then it will returns null and also if exp1 is not same as exp2 then it will returns exp1.

SQL> select nullif(10,10) from dual;
Output: null

SQL> select nullif(10,20) from dual;
Output: 10

- ***Write a query to display the employees who are getting less than 2000 salarys from emp table without using where condition and also in final output return null value in place of more than 2000 salary within salary column by using nullif(), greatest() from emp table ?***

Ans : SQL> select ename, nullif(sal,greatest(2000,sal)) from emp;

ENAME	SAL
SMITH	800
ALLEN	1600
WARD	1250
JONES	
MARTIN	1250

4. **Coalesce():** Oracle 9i introduced coalesce() this function accepts number of expression, this functions returns first normal value within given expressions.

Syntax: coalesce (exp1, exp2,.....expn)

Example:

SQL> select coalesce (null, null, 30, null, 50) from dual;
Output : 30

Difference between nvl(), coalesce()

NVL is an oracle function where as coalesce() is a ANSI standard function, that's why coalesce function works with all relational databases, whenever coalesce having 2 parameter then coalesce and NVL() returns same results.

Example

SQL> select nvl (null, 20) from dual;
O/P: 20
SQL> select coalesce (null, 20) from dual;
O/P: 20
SQL> select nvl (10, 20) from dual;
O/P: 10
SQL> select coalesce (10, 20) from dual;
O/P: 10

Whenever we are using NVL() oracle server returns result if exp1, exp2 not belongs to same data type also i.e. whenever exp2 internally automatically converted into exp1 character data type then NVL() returns result , then those expression not belongs to same data type also. Where as in coalesce exp1, exp2 must belongs to same data type.

Example:

SQL> select nvl ('a', sysdate) from dual;
O/P: a
SQL> select coalesce ('a', sysdate) from dual;
Error: inconsistent datatype expected CHAR got DATE.

Transaction: It is a logical unit of work in between two points is called "Transaction" (or) a set of DML statements which is used to perform particular task is also called as transaction.

In oracle we are maintaining transaction by using following transaction command, these are:

1. **Commit**
2. **Savepoint**
3. **Rollback**

1. **Commit:** This command is used to save the transaction permanently in to database. In all relational databases by default all DDL command are automatically committed.
   In all relational databases before we are performing rollback transaction then we must commit the transaction at least one time in hard disk by using commit command.

Example

```
SQL>create table test(sno number(10));
SQL>insert into test values(.............);
SQL>commit;
SQL>select * from test;
```

SNO
1
2
3
4

```
SQL>delete from test;
     4 rows deleted
SQL>select * from test;
     no row selected
SQL> rollback;
SQL>select * from test;
```

SNO
1
2
3
4

2. **Savepoint:** It is a logical mark in between the transactions is called "Savepoint".

Syntax: savepoint savepointname;

Rollback to Particular Savepoint

Syntax: rollback to savepointname;

3. **Rollback:** This command is used to "UNDO" the transactions from memory.

Syntax: rollback;

Example

```
Sql> insert into emp(empno) values(1);
Sql> update emp set sal=sal+100 where ename= 'SMITH';
Sql> savepoint s1;
Sql> insert into emp(empno) values(2);
Sql> update emp set sal=sal+100 where ename= 'KING';
Sql> savepoint s2;
Sql> delete from emp where empno=2;
Sql> rollback to s1;
Sql> commit;
Sql> select * from emp;
```

Oracle 10g introduced regular expressions. Regular expressions are special type of notation which is used to search/match/replace strings. Regular expressions are also called posix notations.

Note: In oracle regular expression works with **CHAR, VARCHAR, VARCHAR2, CLOB** datatypes but regular expressions doesn't work with **LONG** datatype. Regular expression having so many Meta characters through these Meta characters we can search or match or replace strings. Meta characters are special characters having special meaning.

Oracle having following pre-defined regular expression functions. These are:

1) **Regexp_like()**
2) **Regexp_instr()**
3) **Regexp_substr()**
4) **Regexp_replace()**
5) **Regexp_count() (oracle 11g)**

Meta Characters:-

.	→	Matches any single character.
+	→	Matches one or more occurances of preceding char.
*	→	Matches zero or more occurances of preceding char.
?	→	Matches zero or one occurances of preceding char.
^	→	Begening of line anchar.
$	→	End of line anchar.
\	→	Escape char.
[]	→	Matches any char in the list.
[^...]	→	Negation -> i.e matches other than list of characters.
|	→	OR -> a|b -> matches either a or b.
{m}	→	Matches exactly m occurances.
{m,n}	→	Matches at least m to n occurances.
[[:upper:]]	→	Matches upper case characters.
[[:lower:]]	→	Matches lower case characters.
[[:alpha:]]	→	Matches alphabets.
\d	→	Matches digits
\s	→	Matches spaces
\w	→	Matches word
\n	→	Back references -> i.e \1 -> represent first subgroup. \2 -> represent second subgroup.
(....)	→	Subexpression or grouping.

1) **Regexp_like() :** This function is used to search strings based on regular expression pattern.

```
Syntax: regexp_like (colname, 'pattern');
```

- ***Write a query to display the employee whose ename contain 'AM' (or) 'AR' string from emp table by using like operator?***

  Ans: select * from emp where ename like '%AM%' or ename like '%AR%' ;

- ***Write a query to display the employee whose ename contain AM or AR strings from emp table by using regexp_like() ?***

Ans: *select * from emp where regexp_like (ename, 'AM|AR');*

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL
7521	WARD	SALESMAN	7698	22-FEB-81	1250
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250
7782	CLARK	MANAGER	7839	09-JUN-81	2450
7876	ADAMS	CLERK	7788	23-MAY-87	1100
7900	JAMES	CLERK	7698	03-DEC-81	950

- ***Write a query to display the employee whose ename starts with either A or B from emp table by using regexp_like() ?***

Ans: *select * from emp where regexp_like (ename, '^A|^B');*

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600
7698	BLAKE	MANAGER	7839	01-MAY-81	2850
7876	ADAMS	CLERK	7788	23-MAY-87	1100

- ***Write a query to display the employee whose ename end with either N or S from emp table by using regexp_like () ?***

Ans: *select * from emp where regexp_like (ename, 'N$|S$');*

- ***Write a query to display the employee whose ename 2nd letter is either M or L or A from emp table by using regexp_like () ?***

Ans: select * from emp where regexp_like(ename, '^.[MLA]');

- ***Write a query to display the employees whose ename 2nd letter is not a M, L, A from emp table by using regexp_like() ?***

Ans: SQL> select * from emp where regexp_like(ename, '^.[^MLA]');

Note: regexp_like() having optional 3rd parameter this 3rd parameter is a match optional. This match option has 2 Meta characters.

i. C (Case sensitive) → (Default)
ii. I (Case insensitive)

Syntax: regexp_like (colname, 'pattern', '[match option]')

- ***Write a query to display the employees whose ename starts with S from emp table by using regexp_like() ?***

Ans :
```
SQL> insert into emp(empno, ename) values(1, 'smith');
SQL> select * from emp;
SQL> select ename from emp where regexp_like(ename, '^S');
```

ENAME
SMITH
SCOTT

```
SQL> select * from emp where regexp_like(ename, '^S' , 'C');
```

ENAME
SMITH
SCOTT

```
SQL> select * from emp where regexp_like(ename, '^S' , 'i');
```

ENAME
SMITH
SCOTT
smith

```
SQL> create table test (phoneno varchar2(20));
SQL> insert into test values('123.456.789.4579');
SQL> insert into test values('345.567.985');
SQL> insert into test values('345.567.985.35');
SQL> commit;
SQL> select * from test;
```

Phoneno
123.456.789.4579
345.567.985
345.567.985.35

- ***Write a query to display those phoneno having 3 digits .3digits .end with 3 digits from the above table by using regexp_like() ?***

Ans: SQL> select * from test where regexp_like(phoneno, '\d{3}. \d{3}.\d{3}');

Phoneno
123.456.789.4579
345.567.985
345.567.985.35

Solution

```
SQL> select * from test where regexp_like(phoneno, '\d{3}. \d{3}.\d{3} $');
```

(or)

```
SQL> select * from test where regexp_like(phone, '[[:digit:]]{3}.[[:digit:]]{3}.[[:digit:]]{3} $');
```

Phoneno
345.567.985

Example

```
SQL> create table test1(ename,varchar2(10));
SQL> insert into test1 values('aaaabcc');
SQL> insert into test1 values('aabbbbc');
SQL> insert into test1 values('aacc');
SQL> insert into test1 values('a');
SQL> commit;
SQL> select * from test1;
```

ENAME
aaaabcc
aabbbbc
aacc
a

- ***Write a query to display the employees whose ename having at least second 'a' and zero or more 'b' and at least one 'c' from the above table by using regexp_like() ?***

Ans: SQL> select * from test1 regexp_like(ename, 'aa+b*c+');

ENAME
aaaabcc
aabbbbc
aacc

Note: If you want to escap special meaning of the meta character then we must use \ (backward) infront of the meta character.

```
SQL> insert into test1 values ('*');
1 row created.
SQL> select * from test1;
```

ENAME
aaaabcc
aabbbbc
aacc
a
*

```
SQL> select * from test1 where regexp_like (ename, '\*');
```

ENAME
*

```
SQL> create table test2;
SQL> insert into test2 values('1. abc');
SQL> insert into test2 values('2.  abc');
SQL> insert into test2 values('3.   abc');
SQL> insert into test2 values('4.abc');
```

ENAME
1. abc
2. abc
3. abc
4.abc

- ***Write a query to display the employees whose ename having following regular expression data by using above table ".one digit" ".one or more spaces abc"?***

Ans: SQL> select * from test2 where regexp_like(test2, '\d\.\s+abc');

2) Regexp_instr() :

This function always returns number of data type. This function returns position based on specified regular expression pattern, it is also same as either INSTR(). This function accepts accepts 5 parameters. This function is also same as INSTR() but this function having an optional 5th parameter. This parameter represents return position, this return position having 2 values either 0 or 1.

Syntax: regexp_instr(colname/string, 'pattern', searching position, no of occurrences from position, return position

Return position (0 or 1)

0 : (Default current position)
1 : (Next position of regular expression pattern)

Example:

```
SQL> select regexp_instr('ABC1DEF', '[[:digit:]]',1,1) from dual;
O/P: 4
SQL> select regexp_instr('ABC1DEF', '[[:digit:]]',1,1,0) from dual;
O/P: 4
SQL> select regexp_instr('ABC1DEF', '[[:digit:]]',1,1,1) from dual;
O/P: 5
```

Example:

```
SQL> create table test(mailid varchar2(20));
SQL> insert into test values('abc@gmail.com');
SQL> insert into test values('xyz@gmail.com');
SQL> insert into test values('pqr @gmail.com');
SQL> insert into test values('zzz @gmail.com');
SQL> select * from test;
```

MAILID
abc@gmail.com
xyz@gmail.com'
pqr @gmail.com
zzz @gmail.com

- ***Write a query to display valid mail id from the above table by using regexp_instr()?***

Ans: SQL> select * from test where regexp_instr(mailid, '@')>0 and regexp_instr(mailid, ' ')=0;

MAILID
abc@gmail.com
xyz@gmail.com'

3) Regexp_substr():

This function will extract portion of the string within given string based on regular expression pattern, it is also same as SUBSTR().

Example:

```
SQL> select regexp_substr('oracle12c', '[0-9]+') from dual;
O/P : 12
```

```
SQL> select regexp_substr('oracle12c', '[0-9]+.') from dual;
O/P : 12

SQL> select regexp_substr('100 abc xyz', '\w+',1,3) from dual;
O/P: xyz
SQL> select regexp_substr('100 abc xyz', '\w+',1,2) from dual;
O/P: abc
```

4) **Regexp_replace():** This function is used to replaces character by string by stringby using regular expression pattern.

Example:

```
SQL> select regexp_replace('abc1xyz' , '[[:digit:]]', 'oracle') from dual;
O/P: abcoraclexyz
```

Back reference and grouping

In oracle if you want to divide string into some group then we must specified group of characters within paranthesis and also these subgroup are control by using back reference. In regular expression back references is represented by using \n.Here n represented number, here \1 represents first group and also \2 represent second group.

Syntax: reexp_replace (colname/string, '(group1)(group2).....', 'backreference')

Example:

```
SQL> select regexp_replace('oracle12c', '([[:alpha:]]+)([[:digit:]]+.)', '\1 \2') from dual;
O/P: oracle 12c
SQL> select regexp_replace('oracle12c', '([[:alpha:]]+)([[:digit:]]+.)', '\2 \1') from dual;
O/P:12c oracle
```

5) **Regexp_count():** This function always return no. data type. This function count no. of occurances of the specified character based on regular expression pattern.

Syntax: regexp_count (columnname, 'pattern');

Example :

```
SQL> select regexp_count('abc1xyz1pqr1', '[[:digit:]]') from dual;
Output: 3
```

Like operator escape function:

In oracle like operator having 2 special characters, these are:

i. %
ii. _ (underscore)

These special characters are also called as wild card character. These special characters have special meaning. Whenever these wild card characters are available within a table column and also when we are try to retrieve database on these wild card character, then oracle server returns wrong result.

To overcome this problem for escaping these wild card character special meaning, then ANSI/ISO SQL provided escape function, this escape function used along with like operator only.

Syntax: select * from tablename where colname like 'pattern' escape 'escape character';

Here escape character length must be 1 byte.

- ***Write a query to display the employees whose ename starts with "S_" from emp table by using like operator?***

Ans:
```
SQL> insert into emp(empno,ename) values(1, 'S_MITH');
SQL> select * from emp;
SQL> select * from emp where ename like 'S_%';
```

ENAME
SMITH
SCOTT
S_MITH

Construction:

For escaping wild card character special meaning then first we must use any SQL character after SQL function and also we must specified which wild card character special meaning, if you want to escape that character before we must specified escape character within pattern.

- SQL> select * from emp whose ename like 'S1_%' escape '1';

Execution:

Whenever we are submitting like operator SQL function querys then oracle server identify wild card character after SQL character and also oracle server automatically escapes this wild card character special meaning.

Solution:

SQL> select * from emp where ename like 'S?_%' escape '?';
Output: S_MITH

S?_ :- Here after escape character underscore treated as underscore not treated as wild card characters.

- ***Write a query to display the employees whose ename starts with "S_" from emp table by using regexp_like function?***

Ans:
SQL> select * from emp where regexp_like (ename, '^s_');
Output: S_MITH

- ***Write a query to display the employees whose ename starts with "S_ _" from emp table by using regexp_like function?***

Ans:
```
SQL> insert into emp(empno,ename) values(2, 'S_ _MITH');
SQL> select * from emp;
SQL> select * from emp where ename like 'S_ _%';
```

ENAME
SMITH
SCOTT
S_MITH
S_ _MITH

Solution

- SQL> select * from emp where ename like 'S?_ ?_%' escape '?';
  Output: S_ _MITH

S?_?_ :- Here after escape character underscore treated as underscore, not treated as wild card character.

In all Relational Databases we can also stores hierarchical data. If you want to store hierarchical data then relational table must have minimum three columns and also in these three columns 2 columns must belongs to same data type and also having logical relation.
In oracle if we want to retrieve hierarchical data from a relational table then we are using following clauses these are.

1. Level
2. Start with
3. Connect by

1. **Level:** Level is a PSEUDO column which automatically assigns numbers to level in a tree structure. Levels are also starting with Level number "1" (one).

2. **Start with:** Using "Start with" clause specifies starting condition within tree structure.

   Syntax: start with condition

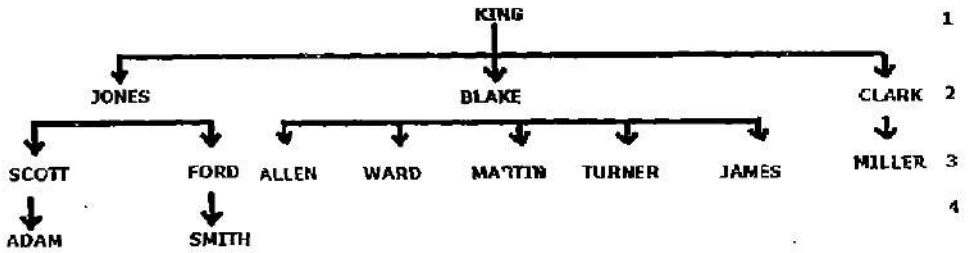
3. **Connect By:** Using "Connect BY" clause we must specify relationship between hierarchical columns using "Prior" operator.

   Syntax: connect by prior ParentColumnName=ChildColumnName;
   Syntax:

```
select level,ColumnName
from tablename
where condition
start with condition
connect by prior
ParentColumnName = ChildColumnName;
```

- ***Write an hierarchical query to display the employees who are working under other employees route node onwards from emp table?***

Ans:
```
select level, ename from emp
Start with mgr is null
Connect by prior empno=mgr;
```



Note: Oracle 9i, introduced "sys_connect_by_path()" which returns path of the hierarchy within tree structure. This function accepts two parameters.

Syntax: sys_connect_by_path (ColumnName, 'DelimiterName');

Example

```
select level, sys_connect_by_path(ename, '->')
 from emp
 Start with mgr is null
Connect by prior empno = mgr;
```

LEVEL	ENAME
1	->KING
2	->KING->JONES
3	->KING->JONES->SCOTT
4	->KING->JONES->SCOTT->ADAMS
3	->KING->JONES->FORD
4	->KING->JONES->FORD->SMITH
2	->KING->BLAKE
3	->KING->BLAKE->ALLEN
3	->KING->BLAKE->WARD
3	->KING->BLAKE->MARTIN
3	->KING->BLAKE->TURNER
3	->KING->BLAKE->JAMES
2	->KING->CLARK
3	->KING->CLARK->MILLER

- ***Write a hierarchical query to display the employees who are working under "BLAKE" from emp table?***

Ans:
```
select level, sys_connect_by_path (ename, '->') from emp
start with ename='BLAKE' connect by prior empno=mgr;

LEVEL  ENAME
-------------
1->BLAKE
2->BLAKE->ALLEN
2->BLAKE->WARD
2->BLAKE->MARTIN
2->BLAKE->TURNER
```

Example

```
select level, sys_connect_by_path(ename, '->')
from emp
start with ename= 'BLAKE'
connect by empno= prior mgr;
```

Output:

```
LEVEL NAME
---- -------
1-> BLAKE
2->BLAKE->KING
```

Prior

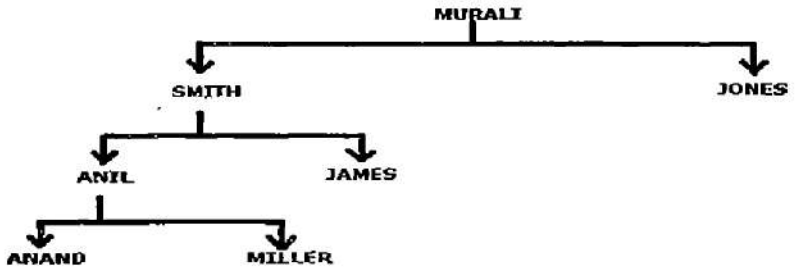
Prior is a "Unary Operator" used along with "Connect by" clause, whenever we are using prior operator in front of child column (Empno) then oracle server uses top-bottom search within tree structure.
Whereas whenever we are using prior operator in front of the parent columns (mgr) then oracle server uses bottom-top search within tree structure.

Execution

Whenever we are submitting H.Q then Oracle server searching data based on "Start with" clause within relational table i.e. it will take a value from the "Prior" operator column and then search that value is available within another hieratical column in a relational table. If those values are available then corresponding column data will be retrieve from third column.

In oracle we can also store tree structure data in relational table. In this case that table having minimum 3 columns and also in these 3 columns 2 columns must belongs to same data type and also those 2 columns having related data.

- ***Create a hierarchical table based on the following tree structure?***



Ans:
```
SQL> create table test (ename varchar2 (10), empno number (10), mgr number (10));
SQL> insert into test values ('anand', 1001, 1003);
SQL> insert into test values ('miller, 1002, 1003);
SQL> insert into test values ('anil', 1003, 1005);
SQL> insert into test values ('james', 1004, 1005);
SQL> insert into test values ('smith', 1005, 1007);
SQL> insert into test values ('jones', 1006, 1007);
SQL> insert into test values ('murali, 1007, null);
Sql> select * from test;
```

Output

ENAME	EMPNO	MGR
ANAND	1001	1003
MILLER	1002	1003
ANIL	1003	1005
JAMES	1004	1005
SMITH	1005	1007
JONES	1006	1007
MURALI	1007	

Note: In oracle we can also use "Order By" clause within hierarchical queries but in this case oracle server shorting the data and also it changes the structure of hierarchical. To overcome this problem Oracle introduced "Order Siblings By" clause which is used to shorting data each and every level within tree structure and also it doesn't change structure of the hierarchical.

Syntax: order siblings by columnname [asc/desc];

Example:

```
SQL>  select level, lpad('  ',level*2)|| ename from test
      start with mgr is null
      connect by prior empno=mgr
      order by ename;
```

LEVEL	ENAME
4	anand
3	anil
3	james
2	jones
4	miller
1	murali
2	smith

```
SQL>  select level, lpad('  ',level*2)|| ename from test
      start with mgr is null
      connect by prior empno=mgr
      order siblings by ename;
```

LEVEL	ENAME
1	murali
2	jones
2	smith
3	anil
3	james
4	anand
4	miller

- ***Write a query to display the employee doesn't have any subordinates from the above table by using multiple row subqueries?***

Ans: select * from test where empno not in (select nvl (mgr, 0) from test);

ENAME	EMPNO	MGR
jones	1006	1007
anand	1001	1003
miller	1002	1003
james	1004	1005

- ***Write a query to display the manager from the above table by using multiple row subqueries?***

Ans: select * from test where empno in (select mgr from test);

ENAME	EMPNO	MGR
anil	1003	1005
smith	1005	1007
murali	1007	

Normalization is a scientific process which is used to decomposing a table into number of tables. This process automatically reduces duplicate data and also automatically avoids insertion, updating and deletion problems.

In design phase of the SDLC (Software Design Life Cycle) database designer design logical model of the database. In these logical model only designers uses normalization process by using normal forms.

In 1970 E.F.Codd written a paper "relational model of data for large shared data banks". In this paper only E.F.Codd introduced normalization process by using "normal forms".

These Normal Forms are:

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. Boyce-Codd Normal Form (BCNF)
5. Fourth Normal Form
6. Fifth Normal Form

1. **First Normal Form (1NF):**

If a table is in first normal form then in that table data in each column should be "Atomic (single)" i.e. the data doesn't contains multiple value seperated with comma (,) within single cell & also identifieng a record uniquely by using a key.

ITEM TABLE (NOT IN 1NF)

ITEM NAME	COLOR	PRICE	TAX
Marker	BLACK RED	20	0.2
Pen	BLUE GREEN	30	0.3

1 NF TABLE →

CANDIDATE KEY

ITEM TABLE

ITEM NAME	COLOR	PRICE	TAX
MARKER	BLACK	20	0.2
MARKER	RED	20	0.2
PEN	BLUE	30	0.3
PEN	GREEN	30	0.3

Process:

Identifying a "Repeating columns groups" and then those column group put it into separate table in more "atomic form" this table is called first normal form table, by default first normal form table is a child table because in this table one column having duplicate data, by default tat column is a foreign key.

ELECTRONIC SHOP

Order Form

Order no :
Order Date :
Cust Name :
Address :
Phone No :
Item Name : LCD TV / Refrigirator / Computer
Amount :

OK

Primary Key

ORDER MASTER TABLE

Oder No	Order Date	Cust Name	Address	Phone No
1	-	-	-	-

Foreign Key

1NF TABLE (Order Detailed Table)

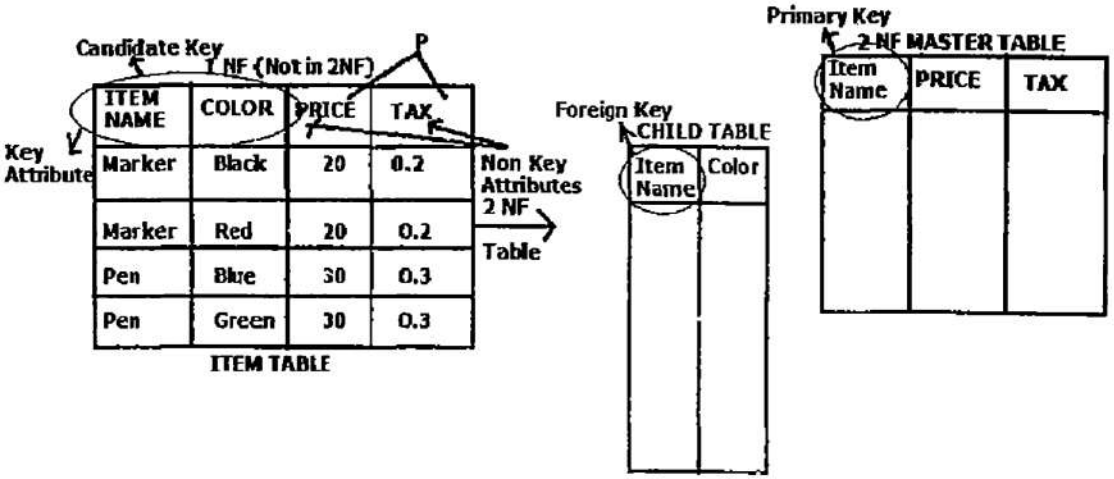
Order No	Item Name	Amount
1	LCD TV	30000
2	Refrigirator	40000
3	Computer	50000

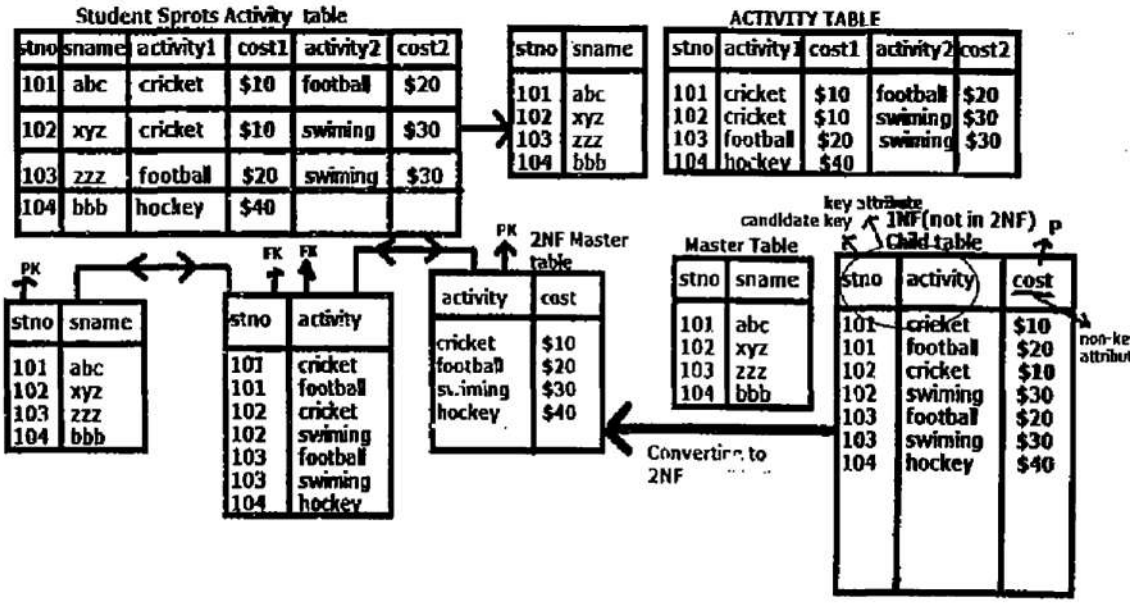
2. **Second Normal Form (2NF):**

If a table is in first normal form and also all non key attributes are fully functionally dependent on total candidate key.
Generally first normal forms deals with atonacity where as second normal forms deals with relationship between key and non key attributes.
If a table is in First Normal Form (1NF) and also that tables having any partial non key attributes then that table not in Second Normal Forms (2NF).

Process:

Identifying partial non key attributes which depends on partial key attributes then those all attributes put into separate table, this table is called Second Normal Forms (2NF) table, by default this table is a master table. In this table only all non key attributes are fully functionally dependent on primary key.





Ecode	Projcode	Dept	Dept head	Hours
E101	P1	Systems	D1	4
E102	P1	Sales	D2	5
E103	P1	Admin	D3	7
E101	P2	Systems	D1	9
E102	P2	Sales	D2	10
E104	P2	Research	D4	12
E102	P3	Sales	D2	8
E104	P3	Research	D4	6

Phase 1:

1. Ecode ---(<---FD----)----→ Hours(Wrong) E101--→ 4,9
2. Projcode---(<---FD---)----→ Hours(Wrong) P1--→ 4,5,7
3. Ecode+Projcode ---→ Hours(Correct) F -> Fully Functional Dependent

Phase 2:

1. Ecode----(<---FD---)----→ Dept(Correct) E101--→ Systems
2. Projcode----- (<---FD---) ---→ Dept (Wrong) P1--→ systems, sales, admin Once partial dependencies is satisfied we want to go for "Fully Functional Dependent".

Phase 3:

1. Ecode----(<---FD---)--→ Depthead(Correct)
2. Projcode--→ Depthead(Wrong)

Primary key

2NF Table(Master Table)

Ecode	Dept	Depthead
E101	Systems	D1
E102	Sales	D2
E103	Admin	D3
E104	Research	D4

CHILD TABLE

Ecode	Projcode	Hours
E101	P1	4
E102	P1	5
E103	P1	7
E101	P2	9
E102	P2	10
E104	P2	12
E102	P3	8
E104	P3	6

Before Normalization process in the above research having Insertion, Updation and Deletion problems.

<u>Insertion Problem:</u> In the above resource table when we are trying to insert particular department employee then we must assign project code. If project code is not available then we need to supply null values for the project code field. This is called Insertion Problem.

<u>Updation Problem</u>: In the above resource table ecode, dept, depthead attribute values are repeated. Whenever an employee transfer from one dept to another dept then we need to modify all these 3 attribute values correctly, otherwise incontinently problem occurred, this is called updating problem.

<u>Deletion Problem</u>: In the above resource table when we are trying to delete an employee record then automatically department details also deleted. This is called deletion problem.
Whenever we are using normalization process then automatically Insertion, Updation, Deletion problems are automatically avoided.

Without using normalization process

ENAME	SAL	DNAME	DEPTHEAD	LOC
SMITH	3000	IT	Murali	Hyd
ALLEN	4000	HR	abc	mumbai
WARD	5000	IT	Murali	Hyd
JONES	6000	HR	abc	mumbai
MILLER	7000	IT	Murali	Hyd
FORD	8000	IT	Murali	Hyd

Data Redundancy Problems:

1. Disc space wasted
2. Data inconsistency
3. DML operations become slow

Using Normalization Process

PR

Deptid	Dname	Depthead	Loc
1	IT	Murali	Hyd
2	HR	abc	mumbai

FR

empid	ename	sal	Deptid
1	SMITH	3000	1
2	ALLEN	4000	2
3	WARD	5000	1
4	JONES	6000	2
5	MILLER	7000	1
6	FORD	8000	1

A Table is in First Normal Form (1NF) :

1. Data in each column should be atomic. There is no multiple values separated with comma (,).
2. Table does not contain repeating column groups.
3. Identifying a record uniquely using a "Primary Key".

Non- Atom columns(emp)

DName	Ename
IT	SMITH, ALLEN, WARD
HR	FORD

Problem

It is not possible to select, insert, update, delete a Single employee

Repeating Column Groups

Dname	ename1	ename2	ename3
IT	SMITH	ALLEN	WARD
HR	FORD		

Problem

When we are adding more than "3" employees then we must change structure of the table.

When we are inserting less than "3" employees then disk space wasted.

Solution:

FR

MASTER TABLE

DEPTID	DNAME
1	IT
2	HR

FR

CHILD TABLE(1NF)

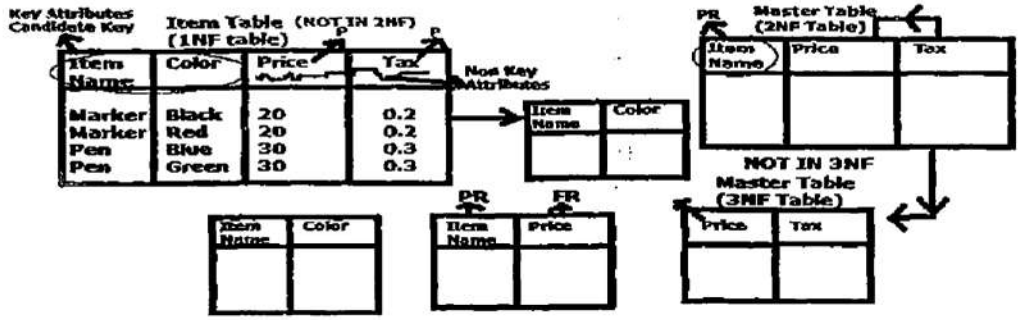
DEPTID	ENAME
1	SMITH
1	ALLEN
1	WARD
2	FORD

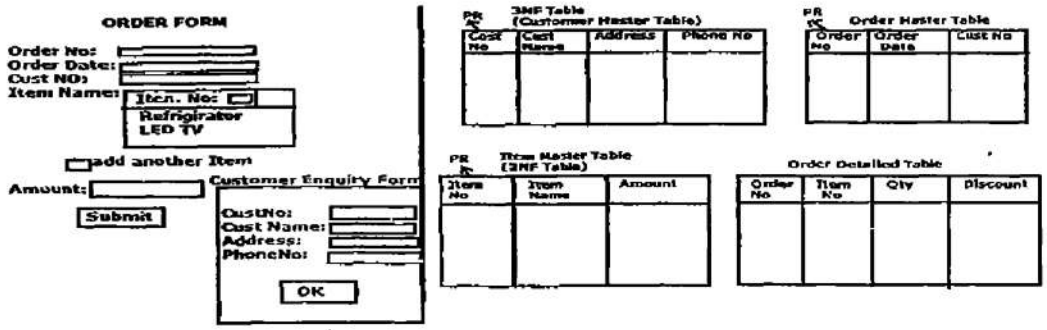
3. Third Normal Form (3NF):

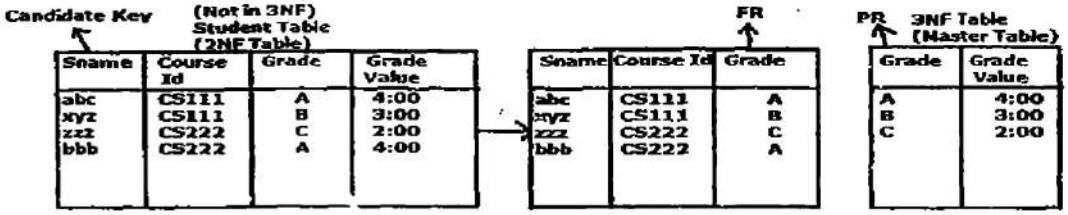
If a table is in 2NF and also all non-key attributes are only dependent on primary key. If a table is in second normal form (2NF) and also in that table any non-key attributes which depends on another non-key attributes then that table not in third normal form (3NF).

Process:

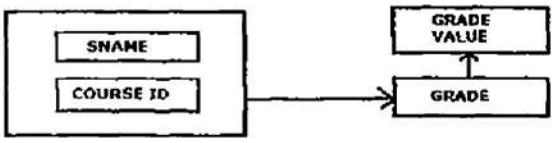
Identify non-key attributes which depends on another non-key attributes then those all non-key put into separate table. This table is called 3NF table, by default this table is in master table. In this table all non-key attributes are only dependent on primary key.







Logical Diagram:



Note: In database design any non-key attributes which depends on another non-key attributes are called transitive dependency. In 3NF table there is no transitive dependency.

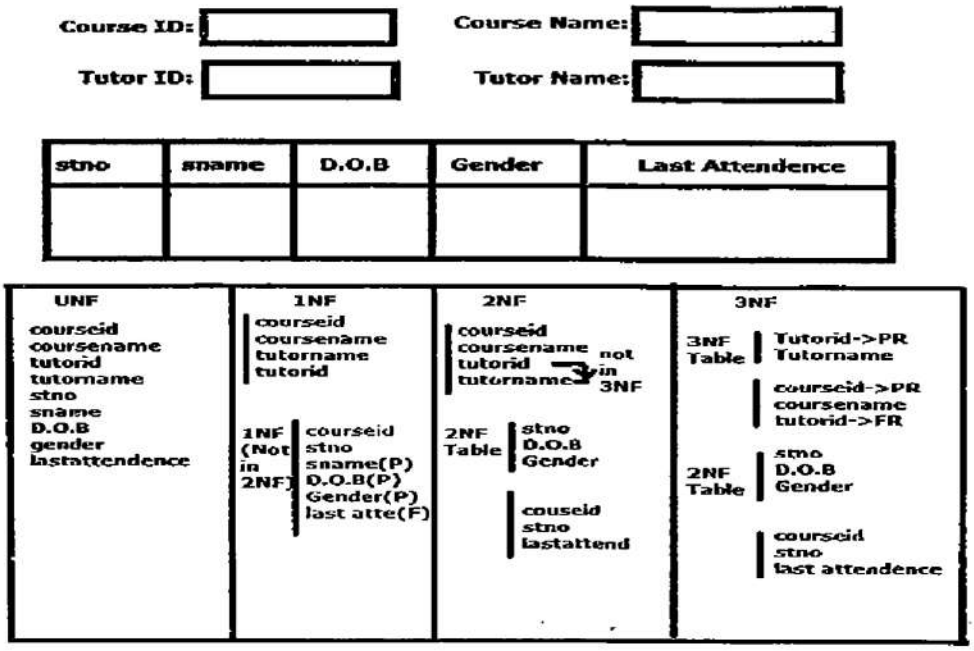
1NF: Remove repeating columns.

2NF: Remove partial attribute.

3NF: Remove non-key attributes which are not dependent on key (or) Remove non-key attributes which are dependent on another non-key attributes.

CLUSTERS

- Cluster is a database object which contains group of tables together and shares same data blocks.
- Clusters are used to improve performance of the joins that's why clusters are created by DBA.
- Generally clusters table must have a common column name, this common column is also called as cluster key.
- In all relational databases whenever we are submitting inner join or outer join then database server first checks from clause tables are available in cluster or not. If those tables are available in cluster then database server very fastly retrieve data from cluster tables.
- Generally, clusters are created by database administrator at the time of table creation.



In oracle when we are creating cluster by using following 3 step process. These are.....

Step 1: Create a Cluster
Step 2: Create an Index on Cluster
Step 3: Create Cluster Tables.

Step 1: Create a cluster:
Clusters are created based on common column. This common column is also called as "Cluster Key".

Syntax: create cluster clustername (CommonColumnName datatype(size));

Step 2: Create an Index on Cluster:

Syntax: create index indexname on cluster clustername;

Step 3: Create Cluster Tables:

Syntax: create tablename (column1 datatype(size), column2 datatype(size),....) cluster clustername (CommonColumnName);

Example:

```
Sql>  create cluster emp_dept (deptno number (10));
Sql>  create index abc on cluster emp_dept;
Sql>  create table emp1 (empno number (10),
      ename varchar2 (10), sal number (10),
      deptno number (10)) cluster emp_dept (deptno);
Sql>  create table dept1 (deptno number (10),
      dname varchar2 (10), loc varchar2 (10)) cluster emp_dept (deptno);
Sql>  desc emp1;
Sql>  desc dept1;

Sql>  insert into emp1 values (1, 'abc', 2000, 10);
Sql>  select * from emp1;
```

EMPNO	ENAME	SAL	DEPTNO
1	abc	2000	10

```
Sql>  insert into dept1 values (10, 'xyz', 'hyd');
Sql>  select * from dept1;
```

DEPTNO	DNAME	LOC
10	xyz	hyd

Note: In all database system cluster tables having same rowid.

Example:

```
Sql> select rowid from emp1;
Sql> select rowid from dep1;
```

Note: Generally, we cannot drop cluster if cluster having tables. To overcome this problem Oracle 8.0 introduced "Including tables" clause which is used to drop cluster along with tables.

Syntax: drop cluster clustername including tables;

Example: SQL> drop cluster emp_dept including tables;
Cluster dropped;

In Oracle all clusters information stored under "user_clusters" data dictionary. In all relational databases using Clusters we can achieve physical data independence, because whenever we are adding clusters table structure do not effects but only performance is effected when we are using "joins".

Functional Dependency (FD):

If any given two tuples in a relation "r" then if X (an attribute set) agrees then Y (another attribute set) cannot disagree then only X->Y is called "Functional Dependency".

X->Y (here X is also called a "Determinant" and Y is also called as "Dependent")

Here Y is functionally dependent on X (or) X functionally determines Y.

Super key, Candidate key

In database design first designers defines super keys from a table and then only designers selects candidate key from those super keys.

Super Key: A column (or) combination of columns which uniquely indentifying a record in a table is called "Super Key".

Candidate Key: A minimal "Super Key" which uniquely identifying a record in a table is called "Candidate Key". (or) A "Super Key" which is a subset of another "Super Key" then those combination of columns are not a "Candidate Key".

Example:

empno	ename	skill id	skill	voter id
1	murali	1	Orade	v1
1	murali	2	Sysbase	v1
1	murali	3	Ingress	v1
2	abc	4	DB2	v2
2	abc	1	Orade	v2
3	xyz	5	Informix	v3
4	zzz			v4
5	bbb	6	SQL server	v5
5	bbb	4	DB2	v5

DATABASE DESIGNERS

Superkeys:
1. empno+skill
2. empno+ename+skill
3. empno+voterid+skill
4. empno+skill
5. ename+voterid+skill
6. voterid+skill
7. empno+ename+voterid+skill

Not a Super Key:
Eg:
1. empno+ename
2. empno+voterid
3. ename+voterid

Candidate Keys:
1. Empno+skill
2. Ename+skill
3. Voterid+skill

4. **BCNF**

- If a table is in BCNF then every determinant is a "Candidate Key".
- If a table having more than one composite candidate keys and also those candidate keys are overlapped and also one candidate key non-key attribuite which depends on another composite candidate key non-key attribute then only we are using BCNF process.
- Generally, in database design if a table having multiple composite candidate keys then deletion problem occurred. To overcome this problem database designer uses BCNF process.
- Generally, 2nd and 3rd normal forms deals with relationship between key and non key attributes. Whereas BCNF deals with relationship between non-key attributes within composite candidate keys itself.

Process:

Identify non-key attribute from a candidate key which depends on another composite key non-key attributes then those all attributes are put into separate table, these table is also called as "BCNF table". In this table only every determinant behaves like a candidate key and also this table reduces duplicate data. These tables are also behaves like a master table. This BCNF table doesn't contain non-key attribute.

Non key attributes

Profcode	Dept	H.O.D	Hours
P1	Physics	H1	3
P1	Computer	H2	5
P2	Chemistry	H3	6
P2	Botany	H4	7
P3	Zoology	H5	8

FD
1. profcode+dept——>Hours

2. profcode+H.O.D——>Hours
1. Profcode+dept——>candidate1
2. Profcode+H.O.D——>candidate2

Convert this table into BCNF

BCN TABLE

Dept	H.O.D
Physics	H1
Computer	H2
Chemistry	H3
Botany	H4
Zoology	H5

TABLE

Profcode	Dept	Hours
P1	Physics	3
P1	Computer	5
P2	Chemistry	6
P2	Botany	7
P3	Zoology	8

FD
1. Dept——>H.O.D
Determinent
Candidate key

FD
2. H.O.D——>Dept
Determinent
Candidate Key

Note: If a table contains single composite candidate key then BCNF is also same as 3NF.

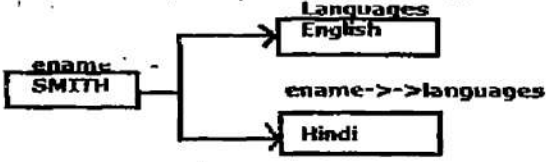
Multi-Valued Dependency (MVD -> ->):

In database design one column same value match with multiple values in another column within a same table is called multi valued dependency. These attributes are also called multi value attributes. Multi valued dependency represented by using double arrow marks (-> ->).

Single Value Multiple Columns

ename	Languages
SMITH	English
SMITH	Hindi

Representation(Multivalued Dependency)



5. **Fourth Normal Form (4NF):**

- If a table is in 4NF then that table doesn't contain more than one independent multivalue attributes.
- If a table contain more than 2 attributes and also identifying a record uniquely by using combination of all these 3 attributes & also one attributes set of values which depends on another attributes set of values and also some attribute set of values which depends on another attribute set of values and also some attributes are not logically related then only we are using 4NF process.
- When a table having more than one independent attribute then that table contain null value problems and also that table automatically having more duplicate data. To overcome these problem for reducing these duplicate data then only database designer uses 4NF process.

Process:

Identify independent multi valued attributes and then those attribute are put it into separate tables, these tables automatically reduces duplicate data and also those table automatically avoided null value problems.

Assumption: Each employee has multiple projects and also each employee having multiple skills.

EMPTable

empid	ProjectID	Skill
1111	RailwayResv.	JSP
1111	RailwayResv.	Oracle
1111	RailwayResv.	.net
1111	LibraryMgmt.	PHP
1111	LibraryMgmt.	Sql Server

→

4NF Table

empid	ProjectID
1111	RailwayResv.
1111	LibraryMgmt.

4NF Table

empid	Skill
1111	JSP
1111	Oracle
1111	.net
1111	PHP
1111	Sql Server

Before 4NF process in the above resource table whenever an employee got a new skill then we need to supply null values for the project id and also when an employee got new project then we need to supply null values for this skills. These null value problems are automatically avoided in 4NF process tables and also these tables reduces duplicate data.

6. **Fifth Normal Form (5NF):**

- If a table is in 5NF then the table cannot be decomposed into further relations.
- Generally if a table have more than one multi-valued attributes and also identifying a record uniquely by using combination of all these attributes and also all attributes set of value depends on one another then only we are using 5NF.
- 5NF is also called as "Projection-Joined Normal Form" because if possible we can also decomposing a table into no. of tables but when we are joining those tables resultant record must be available on resource table.
- Generally in 4NF resource table some attributes are not logically related where as in 5NF resource table all attributes are logically related.

Example:

Product Table

Buyer	Product	Company
B1	Shirt	Levi's
B1	Jeans	Arrow
B1	Shirt	Pepe

5th Normal Form Tables

Buyer	Product

Product	Company

Buyer	Company

Example:

```
SQL> create table stu_dept_sub(sname varchar2(10),
        dname varchar2(10),
        subject varchar2(10),
        primary key(sname,dname,subject));
SQL> insert into stu_dept_sub values(.........);
SQL> select * from stu_dept_sub;
```

ENAME	DNAME	SUBJECT
Allen	CS	Java
Allen	Manager	Marketing
Martin	Chemeistry	Bioche
Martin	Manager	Marketing
Smith	Cs	C++
Ward	Cs	oracle

Convert this table into following two tables:

1. stu_dept
2. stu_sub

1. stu_dept:

```
SQL> create table stud_dept(sname varchar2(10),
        dname varchar2(10),
        primary key(sname,dname));
SQL> insert into stud_dept values(.............);
SQL> select * from stud_dept;
```

SNAME	DNAME
Allen	Cs
Allen	Management
Martin	Chemistry
Martin	Management
Smith	Cs
Ward	Cs

2. stu_sub:

```
SQL> create table stud_sub(sname varchar2(10),
subject varchar2(10),
primary key(sname,subject));
SQL> insert into stud_sub values(.............);
SQL> select * from stud_sub;
```

SNAME	SUBJECT
Allen	Jave
Allen	Marketing
Martin	Bioche
Martin	Marketing
Smith	C++
Ward	Oracle

Testing: (By using joins)

SQL> select t1.sname, t1.dname, t2.subject from stu_dept t1.stu_sub t2 where t1.sname=t2.sname;

SNAME	DNAME	SUBJECT
Allen	Management	Java
Allen	Cs	Java
Allen	Management	Marketing
Allen	Cs	Marketing
Martin	Management	Bioche
Martin	Chemistry	Bioche
Martin	Management	Marketing
Martin	Chemistry	Marketing
Smith	Cs	C++
Ward	Cs	Oracle

Whenever we are joining this table oracle server return more no. of rows then the resource table. To overcome this problem we must use 5NF process.

Solution (By using 5NF): [Create another table dept_sub]

```
SQL> create table dept_sub(dname varchar2(10),
subject varchar2(10),
primary key(dname,subject));
SQL> insert into dept_sub values(.............);
SQL> select * from dept_sub;
```

DNAME	SUBJECT
Chemistry	Bioche
Cs	Java
Cs	C++
Cs	Oracle
Management	Marketing

Testing (by using joins)

```
SQL> select t1.sname,t1.dname,
     t2.subject from stu_dept t1,
     stu_sub t2.dept_sub t3
     where t1.sname=t2.sname and
     t1.dname=t3.dname;
```

ENAME	DNAME	SUBJECT
Allen	CS	Java
Allen	Management	Marketing
Martin	Chemeistry	Bioche
Martin	Management	Marketing
Smith	Cs	C++
Ward	Cs	Oracle

Codd Rules

In 1970 E.F Codd introduced 12 rules for all relational database products these rules are also called as Codd rules. These are...

Rule 1: Information rule:

According to this rule information is to the represented as data store in a cell.

Rule 2: Guaranteed access rule:

In relational data model according to this rule if you want to retrieve unique identification of the record from a table then we are using combination of "tablename + attribute name + primary key value".

Example: select * from emp where empno=7566;

Rule 3: Systematic treatment of null values:

According to this rule in relational database null is an unknown value, one null value is not same as another null value and also null value is not same as any other values and also any arithmetic operation perform with null again it will becomes null.

Rule 4: Active online catalog based on relational model:

According to this rule in all relational databases metadata (data about data) information stored in database same like a relational table this relational table is also called as read only table or data dictionaries. This metadata is nothing but column information, datatype information, constraint into, indexes information.

Rule 5: Data sub language rule:

If you want to operate relational database product then one well define language required this language is nothing but SQL language. This language must have sub languages, through this sub languages we can control relational database product.

Rule 6: View updating rule:

According to this rule in relational database we can also perform DML operation through simple view to base table but we cannot perform DML operations through complex view to base table.

Rule 7: High level insertion, updation, deletion:

According to this rule we can also perform operation set of data at a time by using set operators.

Rule 8: Physical data independence:

Whenever we are modifying internal level then we are adding indexes in internal level then the structure of the table will not be affected in conceptual level.

Rule 9: Logical data independence:

Whenever we are performing modifications in conceptual level then we are not required to change external level is called logical data independence.

Example: In relational databases whenever we are adding a new column in a table within conceptual level then structure of the view will not be affected in external level.

Rule 10: Integrity independence:

According to this rule in relational database we are maintaining proper data by using check constraints.

Rule 11: Distribution subversion rule:

According to this rule in relational database product local, remote server having same command.

Rule 12: Non subversion rule:

According to this rule 2 versions are not same for same database object i.e. in all relational database products whenever we are copying a table from another table constraints cannot bypass i.e. whenever we are creating a table from another table then constraints are never copied.

Difference between DBMS (Database) & RDBMS (Relational Database):

If any database products which satisfied less than 6 CODD rules then those software are also called as DBMS software.

Example: FoxPro, dbase111+

If any database product which supports more than 6 Codd rules then those database are also called as relational database.

Example: Oracle, SQL server, MySql, Teradata,Db2.....

Generally in DBMS product, we are establishing relationship between tables by using program, where as in relational database we are establishing relationship between tables by using foreign key constraint.

Locking is a mechanism which is used to prevent unauthorized access for our resource.

All relational databases system has 2 types of locks.

1. Row Level Locks.
2. Table Level Locks

1. **Row Level Locks:** Oracle 6.0 introduced "Row Level Locks". In this method we are locking set of rows in a table. In row level locks we are locking set of rows by using "for update" clause in all databases. This clause is used only in "Select statement".

Syntax: select * from tablename where condition for update [no wait];

Whenever we are performing locks then another user query the data but they cannot perform DML operations and also in all relational databases whenever we are using "Commit" or "Rollback" command then automatically locks are released.

SQL> conn scott/tiger;	SQL> conn murali/murali;
SQL> select * from emp where deptno=10 for update; SQL> commit;[for releasing lock]	SQL> update scott.emp set sal=sal+100 where deptno=10; [we cannot perform DML operations]

No Wait: No wait is an optional clause used along with "for update" clause. Whenever we are using "no wait" optional clause then oracle server automatically returns control to the current session, if another user not releasing locks also, In this case oracle server returns an error. [ORA-0054: resource busy acquire with NOWAIT specified]

SQL> conn scott/tiger	SQL> conn murali/murali
SQL> select * from emp where deptno=10 for update no wait; ORA-0054: resource busy acquire SQL>_	SQL> select * from scott.emp where deptno=10 for update;

Default Locks: In all database systems whenever we are using DML operations then automatically database servers internally uses default lock. These lock are also called as "Exclusive Lock". These lock also automatically released when we are using commit or rollback command.

scott/tiger	murali/murali
SQL> update scott.emp set sal=sal+100 where deptno=10; SQL> commit; [for releasing locks]	SQL> update scott.emp set sal=sal+100 where deptno=10; [we could not perform DML operations because of "Default Locks"]

Dead Locks: In oracle when 2 or more session try to access same resource at same time when already those sessions are locked each other then automatically dead lock is occurred.

Dead lock is a special type of lock occurred in relational databases when we are try to perform read, write operation at a time. In oracle whenever deadlock occur then oracle server returns an error [ORA-00060 dead lock detected while waiting for resource and also in this case we cannot perform any DML operations and these deadlock also automatically released when we are using commit or rollback.

SQL> scott/tiger	SQL> scott/tiger
SQL> update emp set sal=sal+100 where deptno=10; 3 rows updated SQL> update emp set sal=sal+100 where deptno=20; ORA-0060: dead lock detected SQL> commit; [for releasing locks]	SQL> update emp set sal=sal+100 where deptno=20; 3 rows updated SQL> update emp set sal=sal+100 where deptno=100; [we cannot perform DML operation because of internal lock]

Table Level Locks: In this method we are locking a table. In all relational databases table level lock are handle by Database Administrators. These lock also commit or rollback command. Oracle having 2 types of table level locks.

1. Share lock
2. Exclusive lock

1. **Share Lock:** When we are using share lock then another user query the data but they cannot perform DML operations and also at a time number of users can lock the resource.

Syntax: lock table tablename in share mode;

Here, also whenever we are using "Commit" or "Rollback" then only automatically locks are released.

sql> scott/tiger	sql> murali/murali
SQL> lock table emp in share mode; SQL> commit; [for releasing lock]	SQL> select * from scott.emp; SQL> lock table scott.emp in share mode; SQL> update scott.emp set sal=sal+100; [we cannot perform DML operations]

2. **Exclusive lock:** When we are using "exclusive locks" then another user query the data but they cannot perform DML operations and also at a time only one user lock the resources.

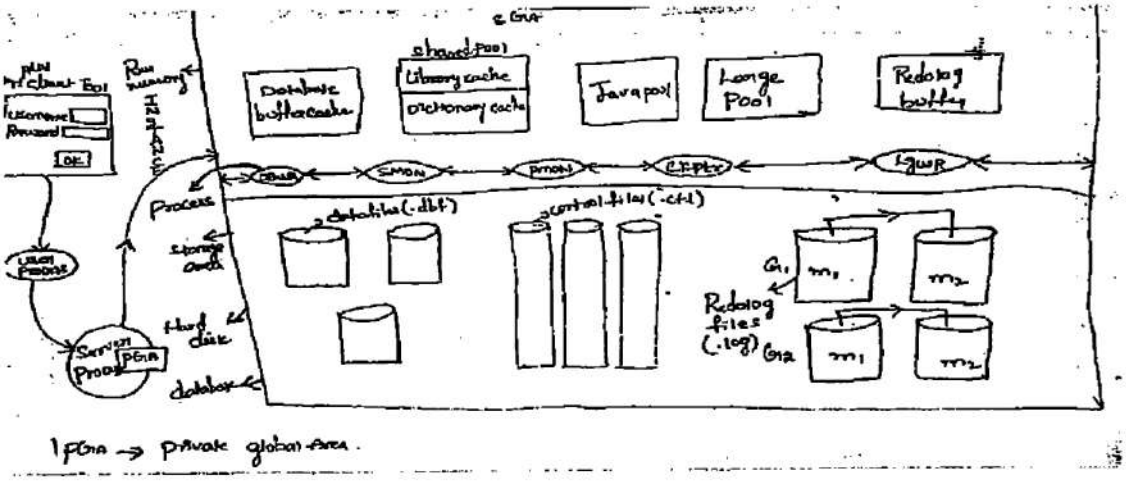
Syntax: lock table tablename in exclusive mode;

scott/tiger	murali/murali
SQL> lock table emp in exclusive mode; SQL> commit;[for releasing locks]	SQL> select * from scott.emp; SQL> lock table scott.emp in share mode;

Note: In all database systems whenever we are using cursor locking mechanism then internally database servers uses "Exclusive locks".

Oracle server mainly consists of two parts:

1. Storage Area
2. Instance



1. **Storage Area:** Whenever we are installing oracle server automatically three physical files are created in hard disk these files are called as Storage Area (or) Physical database.
   a. Data files (.dbf)
   b. Control files (.ctl)
   c. Redo log files (.log)

   All databases having two types of structures
   a. Logical Structure
   b. Physical Structure

Logical Structure:
A structure which is not visible in operating system is called "Logical Structure".
Logical structure contains database objects like table, views, synonyms, clusters, sequences.
Logical structure is handled by either developers (or) database administrators.

Physical Structure:
A structure which is visible in operating system is called Physical structure. Physical structure is handled by database administrator only.Physical structure have a three types of files.
   a. data files (.dbf)
   b. control files (.ctl)
   c. redolog files (.log)

a. **Data Files:** Whenever we are creating database objects logically then those object are physically stored in data files. In oracle all data files information stored under "dba_data_files" data dictionary.
   If we want to path of the data files then we used:-

   Example:

```
Sql> conn sys as dba
Sql> Enter password: sys
Sql> desc dba_data_files;
Sql> select file_name from dba_data_files;
```

b. Control Files:
   - Control files controls all other files within storage area these files extension is ".ctl".
   - Control files also stores physical database information these fiels are used by database administrator in backup and recovery process.
   - In Oracle, all control files information stored under "v$control file" data dictionary.

Example:

```
Sql> desc v$ control file;
Sql> select name from v$control file;
```

c. **Redolog Files:** Redolog files stores committed data from the redolog buffer. Redolog files also used by database administrator in backup and recovery process.
   - In Oracle, all log files Information stored under "v$logfile" data dictionary.

Example:

```
Sql> desc v$logfile;
Sql> select member from v$logfile;
```

2. Instance:

Whenever we are performing any operations within database those all operations also first performed in Physical memory area. This Physical Memory area is also called as "Instance". This Instance is available in RAM memory area this instance having two parts.

1. SGA
2. Processes

SGA: SGA is also called as System Global Area (or) Shared Global Area. This SGA memory area having set of buffers these are:

1. Database buffer cache
2. Shared pool
3. Java pool
4. Large pool
5. Redolog buffer

- Whenever we are submitting SQL/PLSQL code then the code is automatically stored in library cache within shared pool. Library code always reduces parsing (syntax & semantic checking's). Library cache also stores cache values defined in sequence database object.
- Shared Pool also having dictionary cache which executes "DCL" related objects and also checks username and passwords given by the client connect to the server.
- Whenever we are requesting table information by using select statement then server process always checks requested table is available in database buffer cache. If requested table is not available in buffer cache then "DBWR" process checks requested table is available in data files within storage area. If requested table is available in data files then copy of the table is transferred into database buffer cache. Then only server process fetching that table information from database buffer cache into client tool.
- Java Pool executes java related objects where as large pool performs large number of operations and also redo log buffer stored new Information for the transaction.

b. Control Files:
   - Control files controls all other files within storage area these files extension is ".ctl".
   - Control files also stores physical database information these fiels are used by database administrator in backup and recovery process.
   - In Oracle, all control files information stored under "v$control file" data dictionary.

Example:

```
Sql> desc v$ control file;
Sql> select name from v$control file;
```

c. **Redolog Files:** Redolog files stores committed data from the redolog buffer. Redolog files also used by database administrator in backup and recovery process.
   - In Oracle, all log files Information stored under "v$logfile" data dictionary.

Example:

```
Sql> desc v$logfile;
Sql> select member from v$logfile;
```

2. Instance:

Whenever we are performing any operations within database those all operations also first performed in Physical memory area. This Physical Memory area is also called as "Instance". This Instance is available in RAM memory area this instance having two parts.

1. SGA
2. Processes

SGA: SGA is also called as System Global Area (or) Shared Global Area. This SGA memory area having set of buffers these are:

1. Database buffer cache
2. Shared pool
3. Java pool
4. Large pool
5. Redolog buffer

- Whenever we are submitting SQL/PLSQL code then the code is automatically stored in library cache within shared pool. Library code always reduces parsing (syntax & semantic checking's). Library cache also stores cache values defined in sequence database object.
- Shared Pool also having dictionary cache which executes "DCL" related objects and also checks username and passwords given by the client connect to the server.
- Whenever we are requesting table information by using select statement then server process always checks requested table is available in database buffer cache. If requested table is not available in buffer cache then "DBWR" process checks requested table is available in data files within storage area. If requested table is available in data files then copy of the table is transferred into database buffer cache. Then only server process fetching that table information from database buffer cache into client tool.
- Java Pool executes java related objects where as large pool performs large number of operations and also redo log buffer stored new information for the transaction.

Example:

```
Sql> conn sys as sysdba
Enter the Password: sys
Sql> create table b1(sno number(10);
Sql> desc v$database;
Sql> select current_scn from v$database;
Output:
        CURRENT_SCN
        ------------
        1012963

Sql> insert into b1 values(50);
Sql> commit;
Sql> select count(*) from b1;
Output: 1
Sql> select count(*) from b1 as of scn 1012963;
Output: 0
```

4. **PMON:** Process monitor cleans up user processes if users does not disconnect properly from the server.
5. **SMON:** System monitors process recovery all processes. Whenever problems are occurred in a process. That's why system monitor is also called as "Instance Recovery". PGA: (Private Global Area) In Oracle, server process contains a memory area which uniquely indentified by each client connect to the server. This memory area is also called as "Private Global Area". In pl/sql all connections are executed within PGA memory area.

Oracle 9i introduced flashback query.
Flashback query are handled by Database administrator only, flashback queries along allows content of the table to be retrieve with reference to specific point of time by using "as of" clause that is flashback queries retrieves clause that is flashback queries retrieves accidental data after committing the transaction also.

Flashback queries internally uses undo file that is flashback queries retrieve old data before committing the transaction.

Oracle provided following two methods for flashback query:

Method 1: Using timestamp
Method 2: Using SCN (System Change Number)

Timestamp:

Oracle 9i introduced timestamp datatype. Timestamp datatype stores date & time with 6 fractions of second.

Syntax: ColumnName timestamp

If we want to view this function of see then we must insert data by using systime stamp function.

Example: 1

```
SQL> create table test(col1 date);
SQL> insert into test values(sysdate);
SQL> select * from test;
```

Col1
05-MAY-19

Example: 2

```
SQL> create table test1(col1 timestamp);
SQL> desc test1;
```

Column name	Type
Col1	Timestamp(6)

```
SQL> insert into test1 values(to_timestamp('20-DEC-05 08:35:48:49546798', 'DD-MON-YY
HH:MI:SS:FF'));
SQL> select * from test;
```

Col1
20-DEC-05 08:35:48:495468

Note: In oracle if you want to insert our own fraction of second by using "to_timestamp" then we must use "FF" format or if you want to insert system date and fraction of second then we must use "systimestamp" function.

Example: 3

```
SQL> create table test2(col1 timestamp);
SQL> insert into test2 values(systimestamp);
SQL> select * from test2;
```

Example: 4

```
SQL> create table test3(col1 timestamp);
SQL> insert into test3 values(sysdate);
SQL> select * from test3;
```

Col1
06-MAY-19 08:10:54:000000 AM

Method 1: Flashback Query by using timestamp method

Syntax: select * from tablename as of timestamp to_timestamp();

Note: If you want to calculate particular part of time then oracle 9i introduced interval function through this function we can add or substract days,hours,minutes,seconds.Whenever we are using interval function we are specifying number within single quotes.

Example:

```
SQL> create table test(sno number(10));
SQL> insert into test values(.........);
SQL> commit;
SQL> select * from test;
```

Sno
1
2
3
4

```
SQL> delete from test;
o/p: 4 row deleted
SQL> commit;
SQL> select * from test;
o/p:  no row selected
```

Flashback Query:

```
SQL> select * from test as of timestamp(to_timestamp('06-05-2019 08:22:40', 'DD-MM-YYYY HH24:MI:SS');
```

Sno
1
2
3
4

Note: If you want to dump data from undo file into actual table then we are using following syntax.

Syntax: insert into tablename select statement;

```
SQL> insert into test select * from test as of timestamp(to_timestamp('06-05-2019 08:22:40', 'DD-MM-YYYY HH24:MI:SS');
SQL> commit;
SQL> select * from test;
```

Process: Oracle instance also having set of background process these are:

1. DBWR (Database Writer)
2. LGWR (Log Writer)
3. CKPTR (Check Pointer)
4. SMON (System Monitor)
5. PMON (Process Monitor)

1. DBWR **(Database Writer):** Database process always fetches data from database buffer cache and store the data permanently into data file.

2. LGWR **(Log Writer):** Log writer fetches data from redolog buffer into redolog files. Generally whenever we are using DML transaction new value for the transaction also stored in redolog buffer. Whenever user using commit or 1/3 fill of redolog buffer then LGWR process fetches data from redolog buffer into redolog file, these redolog files are used by database administrator in backup & recovery process.

3. **CKPTR (Check Pointer):** Whenever we are performing transactions then DBWR process stored those transaction into data files, in this case check pointer automatically creates a unique identification number for the transaction in data files, control files, redolog files. These numbers is also called as SYSTEM_CHANGE_NUMBER (SCN). Using these SCN number also we can retrieve accidental data by using flashback query. In oracle if you want to view this SCN number then we are using "current_scn" property from "v$database" dictionary.

Example:

```
SQL> conn sys as sysdba;
Enter password: sys
SQL> create table y1(sno number(10));
SQL> desc v$database;
SQL> select current_scn from v$database;
```

Current_scn
2936219

Method 2: Flashback Query by using SCN (System Change Number)

Syntax: *select * from tablename as of scn scnnumber;*

Example:

```
SQL> conn sys as sysdba;
Enter password: sys
SQL> create table dept1 as select * from scott.dept;
SQL> select * from dept1;
```

DEPTNO	DNAME	LOC
10	Accounting	New York
20	Research	Dallas

```
SQL> desc v$database;
SQL> select current_scn from v$database;
```

```
SQL> delete from dept1;
        4 rows deleted
SQL> commit;
SQL> select * from dept1;
        No rows selected
```

Flashback Query:

```
SQL> select * from dept1 as of scn 2936498;
```

DEPTNO	DNAME	LOC
10	Accounting	New York
20	Research	Dallas

```
SQL> select * from dept1;
No rows selected
```

Solution:

```
SQL> insert into dept1 select * from dept1 as of scn 2936498;
4 rows created
```

DEPTNO	DNAME	LOC
10	Accounting	New York
20	Research	Dallas

4. **SMON (System Monitor):** SMON process is also called as instance recovery process because whenever any problems occurs in any background process then SMON process automatically identify that problem and also automatically recovery that process.

5. **PMON (Process Monitor):** PMON process identifies user process those processes are not disconnected properly from oracle database and also PMON process automatically destroys those user processes.

PGA: PGA memory area is also called Private Global Area. This memory area is available in server process.PGA memory area use more performance than the SGA memory area. Whenever we are using order by clause then the shorting is performed in PGA memory area and also PLSQL collections are executed within PGA memory area.
PGA memory area uniquely identifies each client connect to the oracle server because PGA memory area internally stores clients information.

Storage Area: Whenever we are installing oracle server then automatically 3 types of files are created with in hardisk. These files are also called as storage area or physical database. These files are:

1. Data files (.dbf)
2. Control files (.ctl)
3. Redolog files (.log)

These 3 files are also called as physical structure of the database. These files also handled by database administrator.

Data files: These files extension is .dbf in oracle all tables are permanently store in data files.
In oracle all data files information stored under "dba_data_files data" dictionary. If you want to view path of the data files then we are using "file_name from dba_data_files" data dictionary.

```
SQL> conn sys as sysdba;
     Enter password: sys
SQL> desc dba_data_files;
SQL> select file_name from dba_data_files;
```

Oracle internal logical storage structure:

Oracle internal logical storage structure contains :

1. Tablespace
2. Segment
3. Extent
4. Block

1. **Tablespace:**

Oracle will identify physical file with help of a logical pointer is called table space.Generally oracle database is nothing but collection of datafiles physically or collection of tablespace logically.
Generally whenever we are installing oracle then automatically different kind of data store in different data file i.e. in oracle application specific data stored in "users.dbf" files.
This datafile is control by a logical pointer users tablespaces and also all data dictionary information separately stored in "system.dbf" file and also this data file store sys user, system user, dual table. This datafile also controlled by database administrator by using logical pointer system tablespace.
In oracle whenever we are installing on server then automatically 5table spaces arecreated,if you want to view all these tablespace then we are using "user_tablespaces" data dictionary.

Example:

```
SQL> conn sys as sysdba;
     Enter password: sys
SQL> desc user_tablespaces;
SQL> select tablespace_name from user_tablespaces;
```

TABLESPACE_NAME
USERS
SYSTEM
UNDOTBS
SYSAUX
TEMP

In oracle all users information stored under dba_users data dictionary.

Example:

```
SQL> desc dba_users;
SQL> select USERNAME, DEFAULT_TABLESPACE from dba_users;
```

USERNAME	DEFAULT_TABLESPACE
SYS	SYSTEM
SYSTEM	SYSTEM
SCOTT	USERS
MURALI	USERS

```
SQL> update scott.emp set sal=sal+100;
SQL> alter tablespace users read only;
SQL> update scott.emp set sal=sal+100;
Error: file cannot be modified at this time
        ORA-01110: data file 4: 'C:\APP\SHAILENDRA\ORADATA\ORCL1\USERS01.DBF'

SQL> alter tablespace users read write;
SQL> update scott.emp set sal=sal+100;
        14 rows updated
```

Relationship between datafiles and tablespaces:-

In oracle database is nothing but collection of data files physically or collection of tablespaces logically generally in oracle database different kinds of data seperatly store in different datafiles. These datafiles are access by using a logical pointer.

Example:

Old transactions are separately stored in undo data files this old data is accessed by a logical pointer undo tablespace and also temporary data separately stored in temp data files. This data files store temp data like joins, set operators, order by clause and also special features data separately stored in SYSAUX data file. This data files access by using a logical pointer SYSAUX tablespace, if you want to view relationship between tablespaces and data files then we are using dba_datafiles data dictionary.

Example:

```
SQL> conn sys as sysdba;
      Enter password: sys
SQL> desc dba_datafile;
SQL> select file_name, tablespace_name from dba_data_file;
```

Tablespace is nothing but collection of datafiles one datafiles belongs to one tablespace only.Whenever we are installing oracle server then five tablespaces are created.In oracle database administrator also create these one table space by using following syntax.

Syntax: *create tablespace tablespacename datafile 'path of datafile' size any no;*

We can also drop user defined tablespaces by using following syntax.

Syntax: *drop tablespace tablespacename including contents and datafiles;*

Example:

```
Goto C:\> create a new folder and name as abc
SQL> conn sys as sysdba;
Enter password:sys
SQL> create tablespace abc datafile 'C:\abc\abc01.dbf' size 5m;
Tablespace created
SQL> alter tablespace abc add datafile 'C:\abc\abc02.dbf' size 5m;
SQL> desc dba_data_files;
SQL> select tablespace_name, file_name from dba_data_file;
```

Segment: In oracle segment is nothing but collection of extended that form in a database object like tables, synonyms, indexes. In oracle whenever we are creating a table then automatically segment is created. In oracle all segments info stored under "user_segment" data dictionary.

Example:

```
SQL> conn murali/murali;
SQL> select * fro tab;
No rows selected
SQL> desc_user_segments;
SQL> select segment_name from user_segments;
No rows selected
SQL> create table a1(sno number(10));
SQL> select * from tab;
SQL> select segment_name from user_segments;
```

SEGMENT_NAME
A1

Rowid format: Oracle 8i introduced physical rowid is an extended rowid format, this extended rowid format having 18 characters, these encoding characters are A-Z, a-z, +, /. Rowid is a physical address of the row in a table and also it stores 10 bytes, this extended rowid having 4 parts these are:

1. Data object number
2. Tablespace – Relative datafile number
3. Block number
4. Row number

1. **Data object number:** In rowid format first 6 characters represent data object number i.e. it is nothing but segment and also it uses 32 bits.

2. **Tablespace – Relative datafile number:** In rowid format next 3 characters represents tablespace relative datafile number, it uses 10 bits.

3. **Block number:** In rowid format next 6 characters represents row number within row directory under a block it uses 16 bits.

18 characters -> 6+3+6+3
80 bits -> 32+10+22+16

Example:

SQL> select rowid, ename from emp where ename= 'SMITH';

ROWID	ENAME
AAAR3sAAEAAAACXAAA	SMITH

1. Data object number (AAAR3s)
2. Tablespace – Relative datafile number (AAE)
3. Block number (AAAACX)
4. Row number (AAA)

Oracle 8.0, introduced nested table. A table within another table is also called as "Nested Table". Basically, nested table is a user defined datatype which is used to store number of data items in a single unit. Generally, if we want to store number of data items in a single unit then we are using Index by table in PL/SQL, but these tables are not allowed to store these index by table permanently in oracle database.

To overcome this problem oracle 8.0 introduced extension of the Index by table called "Nested Table" which stores number of data items and also these tables are allowed to store permanently into oracle database using SQL.

In oracle we are creating nested table by using following 3 step process:

Step 1: Create An Object Type.
Step 2: Create Nested Table Type
Step 3: Create Actual Table.

Step 1: Create an Object Type:

Before we are creating user defined datatype then first we must create object type which is a collection of predefined datatype object type is also same as structures in "C" language".

Syntax: create or replace type typename as object (attributename1 datatype(size), attributename2 datatype(size),.....);

Step 2: Create a Nested Table Type:

Based on object type we must create our own datatype, this user defined datatype is also known as nested table type.

Syntax: create or replace type typename as table of objecttypename;

Step 3: Create an Actual Table:

Syntax: create table tablename (col1 datatype(size),col2 datatype(size).....,
coln NestedTableTypeName) nested table coln store as anyname;

Example:

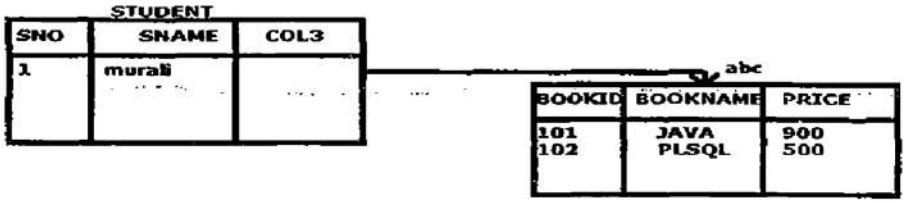
```
SQL> create or replace type obj1 as object(
        bookid number(10), bookname varchar2(10), price number(10));
SQL> create or replace type xyz as table of obj1;
SQL> create table student(sno number(10), sname varchar2(10), col3 xyz)
        nested table col3 store as zzzz;
        Table Created.
SQL> desc student;
```

NAME	TYPE
SNO	NUMBER(10)
SNAME	VARCHAR2(10)
COL3	xyz

If we want to store data into nested table then we are using user defined datatype name within insert statement.

Insertion:

SQL> insert into student values(1, 'murali', xyz(obj1(101, 'java',900), obj1(102, 'plsql',500)));
1 row created.
SQL> select * from student;



Note: We can also retrieve modify, delete nested table data separately by using table operator i.e. in place of table name we are using following syntax in select, update, delete statements.

Syntax: table (select NestedTableColumnName from relational tablename);

Example: SQL> select * from table (select col3 from student);

BOOKID	BOOKNAME	PRICE
101	JAVA	900
102	PLSQL	500

Example: SQL> update table(select col3 from student) set price=1000 where bookid=101;
Ouput:

BOOKID	BOOKNAME	PRICE
101	JAVA	1000
102	PLSQL	500

PARTITIONS

At the time of table creation a table can be decomposed into number of portions is called "partition table".
Partition tables are used in data ware housing applications.
Partition tables are created by database administrator and used in backup and recovery process.
Partition tables are used to improve performance of the applications in backup and recovery process.
Partition tables are created on very large data bases. Partition tables are created based on key column. This key column is also called as "partition key".

Oracle has 3 types of partitions.

1. Range partition
2. List partition
3. Hash partition

1. Range Partition: In this partition we are partitioning tables based on range of values.

```
Syntax: create table tablename(col1 datatype(size), col2 datatype(size),....)
        partition by range(keycolumnname)
        (partition partitionname values less than(value),
        ..............................................
        ..............................................
        partition partitionname values less than(max value));
```

If we want to view particular portion then we are using following syntax:

```
Syntax: select * from tablename partition (partitionname1, partitionname2, ...);
```

Example:

```
sql> create table test(sno number(10), name varchar2(10), sal number(10))
     partition by range(sal)(partition p1 values less than(1000),
     partition p2 values less than(2000),
     partition p3 values less than(max value));
Table created
Sql> insert into test values(&no, '&name', &sal);
Sql> select * from test partition(p1);
```

Output:

```
SNO  NAME  SAL
-------------------
 1   ABC   500
```

In oracle if you want to retrieve particular partition then we are using following syntax.

```
Syntax: select * from tablename partition (partitionname)
```

2. List Partition: Oracle 9i, introduced list partition in this partition we are partitioning tables based on list of values.

```
Syntax: create table tablename(col1 datatype(size), col2 datatype(size),.............)
        partition by list(keycolname)
        (partition partitionname1 values(value1, value2,.............),
        partition partitionname values(default));
```

Note: Generally, if we want to create partition based on character datatype column then we are using "List Partition".

Example

```
Sql> create table test(sno number(10), name varchar2(10))
     partition by list(name) (partition p1 values('India', 'Nepal'),
     partition p2 values('us', 'uk', 'canada')
     partition p3 values(default));
Sql> insert into test values (&sno, '&name');
Sql> select * from test;
```

Output:

```
SNO  NAME
----------
1    AUS
2    JAPAN
3    INDIA
```

```
Sql> select * from test partition(p1);
```

Output:

```
SNO  NAME
----------
1    AUS
2    JAPAN
```

3. **Hash Partition:** In this method whenever we are specifying no. of partitions in oracle server internally automatically creates partitions based on "Hash Algorithm".

```
Syntax: create table tablename (col1 datatype (size), col2 datatype (size), .....)
        partition by hash(keycolname)
        partition anynumber;
```

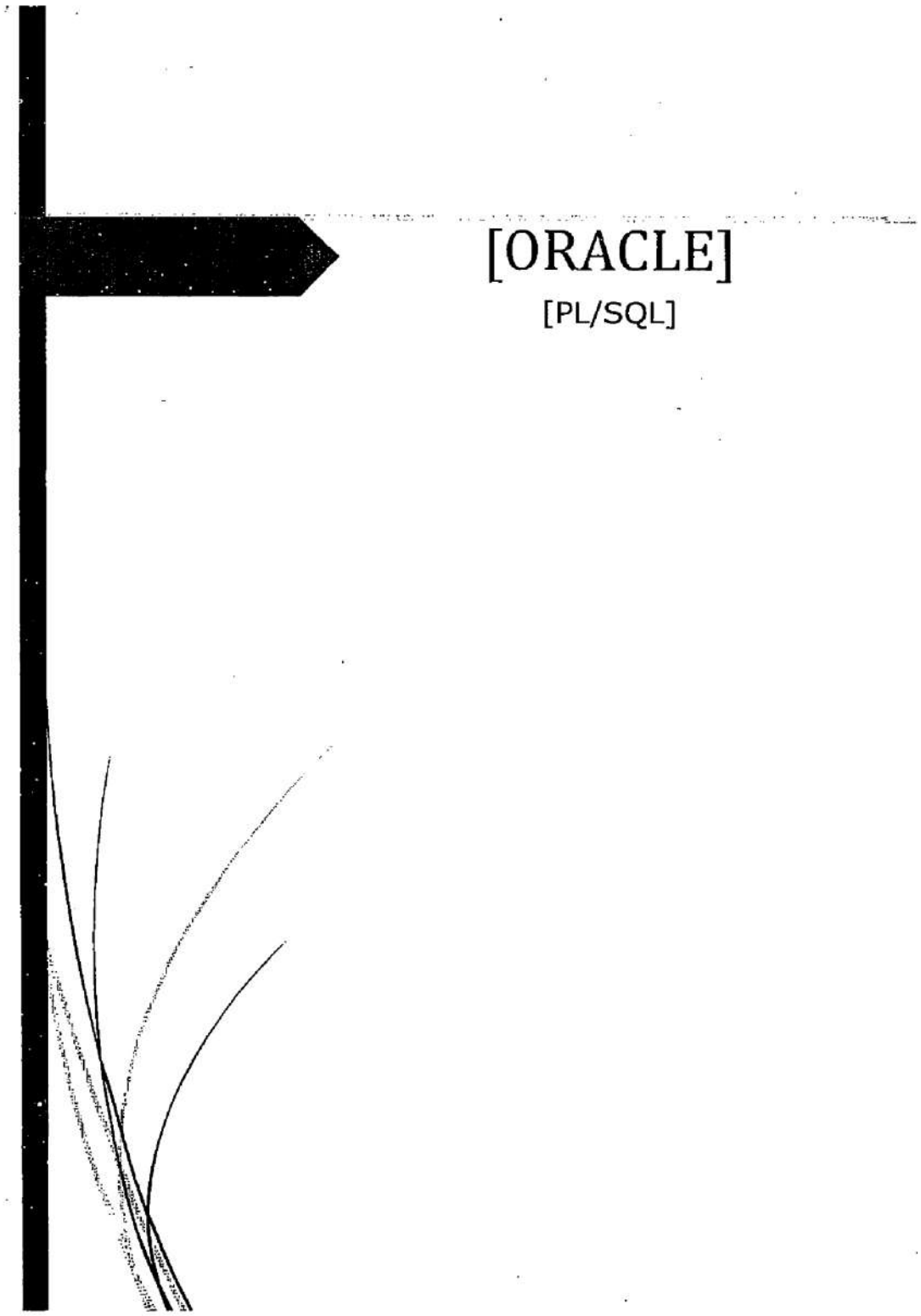
Example

```
Sql> create table test (sno number (10), sal number (10)) partition by hash (sal) partition 5;
        Table created
Sql> insert into values test(&sno, &sal);
```

In oracle if we want to view all views partitions information then we are using "user_tab_partitions" data dictionary.

Example

```
Sql> desc user_tab_partitions;
Sql> select partition_name from user_tab_partitions where table_name='TEST';
```


[ORACLE]
[PL/SQL]

PL/SQL

1. **PL/SQL Introduction:**
   - Select.....Into Clause
   - Variable Attribute (%Type, %Rowtype)
   - Bind Variables
   - Conditional, Control Statements

2. **Cursors:**
   - Explicit cursor & Explicit cursor attributes
   - Explicit Cursor Life Cycle
   - Cursor..... For Loop
   - Parameterized Cursor
   - Implicit Cursors & Implicit Cursor Attributes.
   - Functions, expressions are used in explicit cursor
   - Update, delete, statements are used in cursors (without using where current of, for update clauses)

3. **Exceptions:**
   - Predefined Exceptions
   - User Defined Exceptions
   - Unnamed Exceptions
   - Exception Propagation.
   - Raise_Application_Error()
   - Error Trapping Functions (SQLCODE, SQLERRM)

4. **Sub Programs:**

   Stored Procedures
   - Parameter Modes (In, Out, In Out)
   - No Copy compiler hint
   - Autonomous Transactions
   - Authid Current_User (Definer Rights)

   Stored Functions
   - DML statements are used in functions
   - Select.....into clause used in functions
   - When to use procedure, when to use functions
   - Wm_concat

5. **Packages:**
   - Global Variables
   - Serially_Reusable Pragma
   - Overloading Procedures
   - Forward Declaration

6. **Types Are Used In Packages:**
   - Pl/Sql Record
   - Index By Table(Or) PL/SQL Table(Or) Associative Array
   - Nested Table
   - Varray
   - Refcursor

7. **Bulk Bind:**
   - Bulk Collect Clause
   - Forall Statements
   - Indices Of Clause(10g)
   - Bulk Exceptions Handling Through Sql%Bulk_Exceptions
   - SQL%Bulk_Rowcount()
8. **Ref Cursor:**
   - Strong Ref Cursor
   - Weak Ref Cursor
   - Sys_Refcursor
   - Passing Sys_Refcursor as Parameter to the Stored Procedure.
9. **Local Subprograms:**
   - Local Procedures
   - Local Functions
   - Passing ref cursor as parameter into the local subprograms
10. **UTL_File Package:**
11. **SQL Loader:**
    - Flat Files, Control Files, Bad Files, Discard Files, Log Files.
12. **Triggers:**
    - Row Level Triggers
    - Applications Of Row Level Trigger (Auto Increment Concept)
    - Trigger timing (before / after)
    - Statement Level Triggers
    - Trigger Execution Order
    - Follows Clause (Oracle 11g)
    - Compound Triggers (Oracle 11g)
    - Mutating Error
    - System Level Triggers (or) DDL Triggers
13. **Avoiding Mutating Error Using Compound Trigger**
14. **Dynamic SQL**
15. **Large Objects (Lobs):**
    - Internal Large Objects (CLOB, BLOB)
    - External Large Objects (BFILE)
16. **Where "current of", "for update of" clauses are used in explicit cursors.**
17. **Member Procedures, Member Functions**
18. **11g Features, oracle 12c features**

PL/SQL is a Procedural Language Extension for SQL. Oracle 6.0 introduced PL/SQL.

1. Oracle 6.0--→ PL/SQL 1.0
2. Oracle 7.0--→ PL/SQL 2.0
3. Oracle 8.0--→ PL/SQL 8.0
4. .....
5. .....
6. Oracle 11.2--→ PL/SQL 11.2

Basically, PL/SQL is a Block Structured Programming Language. When we are submitting PL/SQL blocks into the Oracle server then all Procedural statements are executed within PL/SQL engine and also all SQL statements are separately executed by using SQL engine.

Block Structure

```
declare [Optional]
-> variable declaration, cursors, user defined exceptions

begin [Mandatory]
-> DML, TCL
-> select..... into clause
-> conditional, control statements;

exception [Optional]
-> Handling runtime errors

end; [Mandatory]
```

PL/SQL has two types of blocks.

1. Anonymous blocks
2. Named blocks

BLOCKS

ANONYMOUS BLOCKS

```
Eg: declare
    -------
    begin
    -------
    end;
```

1. These blocks doesnot have any name.
2. These blocks are not stored in database.
3. These blocks are not allowed to call in client applications.

NAMED BLOCKS

Eg: Procedures, Functions;

1. These blocks having a name.
2. These blocks are automatically permanently stored in database.
3. These blocks are allowed to call in client applications.

Declaring a Variable

Syntax: variablename datatype(size);

Example

```
declare
a number(10);
b varchar2(10);
```

Storing a value into variable

Using assignment operator (:=) we can store a value in variable.

Syntax: variablename := value;

Example

```
declare
a number(10);
begin
a:=50;
end;
/
```

Display a message (or) variable name

Syntax: dbms_output.put_line ('message');
(or)
Syntax: dbms_output.put_line (variablename);

dbms_output → PACKAGE NAME
put_line → PROCEDURE NAME

Example

```
Sql> set serveroutput on;
Sql> begin
     dbms_output.put_line('welcome');
     end;
     /
Output: welcome
```

Example

```
declare
a number(10);      */ (or) a number(10):=50; [it is also possible]  /*
begin
a:=50;
dbms_output.put_line(a);
end;
/
```

Select.....into clause is used to retrieve data from table and store it into PL/SQL variable.
Select.....into clause always returns "Single record (or) single value" at a time.

Syntax:

```
Select <column name1>, <column name2>,.....
into <variable name1>, <variable name2>,.....
from <table name> where condition;
```

(Where condition must return a single record)

Select... into clause is used in executable section of the PL/SQL block.

- **Write PL/SQL program for user entered empno then display name of the employee and his salary from emp table?**

Ans:

```
declare
 v_ename varchar2(20);
 v_sal   number(10);
begin
 select ename, sal into v_ename, v_sal from emp where empno = &empno;
 dbms_output.put_line(v_ename ||' '|| v_sal);
end;
/
```

Output:

```
Enter value for empno: 7369
SMITH  800
```

Note: In PL/SQL when a variable contains "not null" (or) "constant" clause then we must assign the value when we are declaring the variable in declare section of the PL/SQL block.

Syntax: Variablename datatype(size) not null := value;
Syntax: variablename constant datatype(size) := value;

Example:

```
declare
 a number(10) not null := 50;
 b constant number(10) := 9;
begin
 dbms_output.put_line(a);
 dbms_output.put_line(b);
end;
/
```

Output: 50 9

Note: We can also use "default" clause in place of "assignment operator" when we are assigning the value in declare section of the PL/SQL block.

Example

```
declare
 a number default 50;
begin
 dbms_output.put_line(a);
end;
/
```

Output: 50

➢ **Write a PL/SQL program maximum salary from emp table and store it into PL/SQL variable and display max salary?**

Ans:

```
declare
 v_sal number(10);
begin
 select max(sal) into v_sal from emp;
 dbms_output.put_line('maximum salary:' || v_sal);
end;
/
```

Output: maximum salary: 5000

Example

```
declare
 a number(10);
 b number(10);
 c number(10);
begin
 a := 90;
 b := 30;
 c := greatest(a, b);      c := max(a,b) [error: group function cannot used in PL/SQL expression]
dbms_output.put_line(c);
end;
/
```

Output: 90

Note: In PL/SQL expressions we are not allowed to use **"group functions"**, **"decode conversion function"**, but we are allowed to use **"number functions"**, **"character functions"**, **"delete functions"** and **"date conversion functions"** in PL/SQL expressions.

Example 1

```
declare
 a varchar2(10);
begin
 a := upper('shailendra');
 dbms_output.put_line(a);
end;
/
```

Output: SHAILENDRA

Example 2

```
declare
 a date;
begin
 a := next_date('12 - aug - 15') + 1;
 dbms_output.put_line(a);
end;
/
```

Output: 13-AUG-15

Variable Attributes (Anchor Notations)

Variable attributes are used in place of data types in variable declaration. Whenever we are using variable attributes, Oracle server automatically allocates memory for the variables based on the corresponding column data type in a table.

PL/SQL having two types of variable attributes.

1. **Column level Attributes**
2. **Row level Attributes**

1. **Column Level Attributes:** In this methods we are defining attributes for individual columns. Column Level attributes are represented by using "**%type**".

Syntax: variablename <table name> . <column name> %type;

Whenever we are using "**column level attributes**" oracle server automatically allocates memory for the variables as same as corresponding column datatype in a table.

Example

```
declare
 v_ename    emp.ename%type;
 v_sal  emp.sal %type;
 v_hiredate    emp.hiredate %type;
begin
 select ename, sal, hiredate into v_ename, v_sal, v_hiredate from emp
  where empno = &no;
 dbms_output.put_line(v_ename || ' ' || v_sal || ' ' || v_hiredate);
end;
/
```

Output:

```
Enter value for no: 7902
FORD   3000    03-DEC-81
```

2. **Row Level Attributes:** In this methods a single variable can represent all different datatypes in a row within a table. This variable is also called as **"record type variable"**. It is also same as structures in **"C"** language. Row level attributes are represented by **"%rowtype"**.

Syntax: variablename <table name> %rowtype;

Example

```
declare
 i  emp%rowtype;
begin
 select ename, sal, hiredate into i.ename, i.sal, i.hiredate from emp
  where empno = &no;
 dbms_output.put_line (i.ename ||' '|| i.sal ||' '|| i.hiredate);
end;
/
```

Output:

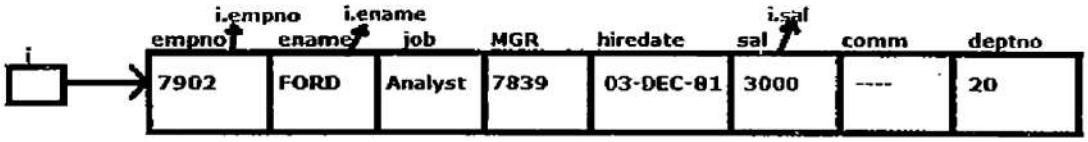
```
Enter value for no: 7902
FORD    3000    03-DEC-81
```

Example:

```
declare
 i    emp%rowtype;
begin
 select * into i from emp where empno = &empno;
 dbms_output.put_line(i.ename ||' '|| i.sal ||' '|| i.hiredate);
end;
/
```

Output:

```
Enter value for empno: 7902
FORD    3000    03-DEC-81
```



1. if
2. if-else
3. elsif

1. If:

Syntax: (olden day syntax style)

```
if condition then
stmts;
end if;
```

2. if-else:

Syntax:

```
if condition then
stmts;
else
stmts;
end if;
```

3. elsif: To check more number of conditions then we are using "elsif".

Syntax:

```
if condition1 then
        stmts;
elsif condition2 then
        stmts;
elsif condition3 then
        stmts;
else
        stmts;
end if;
```

Example

```
declare
 v_deptno number(10);
begin
 select deptno into v_deptno from dept where deptno = &deptno;
 if v_deptno = 10 then
  dbms_output.put_line('ten');
 elsif v_deptno = 20 then
  dbms_output.put_line('twenty');
 elsif v_deptno = 30 then
  dbms_output.put_line('thirty');
 else
  dbms_output.put_line('others');
 end if;
end;
/
```

Output:

```
Enter value for deptno: 20
twenty
sql> /
Enter value for deptno: 40
Others
Sql> /
Enter value for deptno: 90
Error: ORA-01403: no data found.
```

Note 1: When a PL/SQL blocks having "Select into" clause and also if requested data not available in a table through this clause then Oracle server returns an error "ORA-01403: no data found".

Note 2: When a PL/SQL block having pure DML statements and also through those statements if requested data not available in a table then Oracle server does not return any error message.
To handle these types of blocks we are using "Implicit Cursor Attributes".

Example

```
begin
delete from emp where ename= 'welcome';
end;
/
PL/SQL procedure successfully completed.
```

Note 3: In PL/SQL blocks having "select into" clause and also if "select into" clause try to return multiple records at a time (or) try to return multiple values from a single columns at a time then Oracle server returns an error "ORA-1422: Exact fetch returns more than requested number of rows".

Example

```
declare
 i   emp%rowtype;
begin
 select * into i from emp where deptno = 10;
 dbms_output.put_line (i.ename || '  ' || i.sal ||' '|| i.job);
end;
/
Error: ORA-1422: Exact fetch returns more than requested number of rows.
```

Case Statement: Oracle 8.0, introduced case statement and also Oracle 8i introduced case conditional statement. This statement is also called as "Searched Case".

Syntax:

```
case variablename
when value1 then
        stmts;
when value2 then
        stmts;
else stmt n;
end case;
```

Example

```
declare
 v_deptno number(10);
begin
 select deptno into v_deptno from emp where deptno = &deptno;
 case v_deptno
  when 10 then
   dbms_output.put_line('ten');
  when 20 then
   dbms_output.put_line('twenty');
  else
   dbms_output.put_line('others');
 end case;
end;
/
```

Output: Enter value for deptno=10
Ten

Case Conditional Statement (OR) Searched Case (Oracle 8i):

Syntax:

```
case
when condition1 then
        stmts;
when condition2 then
        stmts;
else
        stmts;
end case;
```

Example:

```
case
when v_deptno=10 then
dbms_output.put_line('ten');
when v_deptno between 20 and 30 then
dbms_output.put_line('twenty and thirty');
else
dbms_output.put_line('others');
end case;
end;
/
```

PL/SQL has following 3 types of loop. These are...

1) **Simple Loop**
2) **While Loop**
3) **For Loop**

1. **Simple Loop:** This loop is also called as "Infinite loop". Here body of the loop statements is executed repeatedly.

Syntax:

```
loop
stmts;
end loop;
```

Example:

```
begin
 loop
  dbms_output.put_line('welcome');
 end loop;
end;
/
```

If we want to exit from "Infinite loop" then we are using Oracle provided predefined methods.

Method 1: (Default Method)

Syntax: exit when <true condition>;

Example:

```
declare
 n number(10) := 1;
begin
 loop
  dbms_output.put_line(n);
  exit when n >= 10;
  n := n + 1;
 end loop;
end;
/
```

output:

1
2
3
4
5
6
7
8
9
10

Method 2: (Using IF)

Syntax:

```
if true condition then
exit;
end if;
```

Example:

```
declare
 n number(10) := 1;
begin
 loop
  dbms_output.put_line(n);
  if n >= 10 then
   exit;
  end if;
  n := n + 1;
 end loop;
end;
/
```

1
2
3
4
5
6
7
8
9
10

2. **While Loop:** Here body of the loop statements are executed repeatedly until condition is false. In "While Loop" whenever condition is true then only loop body is executed.

Syntax:

```
while (condition)
loop
stmts;
end loop;
```

Example:

```
declare
 n number(10) := 1;
begin
 while (n <= 10)
loop
  dbms_output.put_line(n);
  n := n + 1;
 end loop;
end;
```

1
2
3
4
5
6
7
8
9
10

3. For Loop:

Syntax:

```
for IndexVariableName in lowerbond .. upperbond
loop
stmts;
end loop;
```

Example:

```
declare
 n number(10);
begin
 for n in 1 .. 10
loop
  dbms_output.put_line(n);
 end loop;
end;
```

Example:

```
declare
 n number(10);
begin
 for n in reverse 1 .. 10
loop
   dbms_output.put_line(n);
 end loop;
end;
```

Note: For loop index variable internally behaves like an "Integer" variable that's why when we are using "**for loop**" we are not required to declare variable in declare section. Generally PL/SQL for loop is also called as numeric "**for loop**".

Example:

```
begin
 for n in 1 .. 10
loop
   dbms_output.put_line (n);
 end loop;
end;
```

Example:

```
Sql> create table test (sno number(10));
Sql> begin
 for n in 1 .. 10
loop
   insert into test values (n);
 end loop
end;
```

Oracle server having 2 engine, these are ...

1) **SQL engine**
2) **PL/SQL engine**

When we are submitting PL/SQL block into oracle server then all SQL statements are executed within SQL engine and also all procedural statements are separately executed within PL/SQL engine.

- **Write a PL/SQL program which is used to retrieve total salary from emp table and then store that total salary into another table?**

Ans:

```
sql> create table target(totalsal number(10));

sql> declare
       v_sal number(10);
       begin
       select sum(sal) into v_sal from emp;
       insert into target values(v_sal);
       end;
       /

Sql> select * from target;
```

Bind variable is a session variable (whenever necessary we can create) created at "host" environment that's why these variables are also called as "host variables".

Bind variables are used in SQL, PL/SQL, Dynamic SQL language that's why these variables are also called as "Non PL/SQL variables". We can also use these variables in PL/SQL to execute when subprograms having OUT, INOUT parameters.

In oracle we are creating bind variables by using following 3 steps process.

Step 1: (Creating a bind variable)

```
Syntax:-        variable <variable name> <data type>;
```

Step 2: (using bind variable)
When we are using bind variable then we must use colon (:) operator in front of the bind variable name.

```
Syntax:- :<bind variable name>;
```

Step 3: (display value from bind variable)
If you want to display value from bind variable then we must use print command at a SQL prompt at following syntax.

```
Syntax:- print <bind variable name>;
```

Example:

```
sql> variable g number;
sql> declare
     a number(10):= 900;
     begin
     :g := a/2;
     end;
     /
PL/SQL procedure successfully completed.

Sql> print g; (or)
Sql> print :g;
```

Output:

```
G
-----
450
```

PL/SQL Data types and Variables:

1. It supports all SQL databases (Scalar Data Types) + Boolean Data types.
2. Composite Data types
3. Ref Objects
4. Large Objects (LOBS) -> CLOB, BLOB, BFILE
5. Bind Variables (or) Non PL/SQL variables

Cursor is a private SQL memory area which is used to process multiple records and also this is a "record by record" process. All database systems having two types of static cursor, these are...

1) **Implicit cursor**
2) **Explicit cursor**

Explicit Cursor: (Static Cursor)

For SQL statements returns multiple records is called "**explicit cursors**" and also this is a "**record by record**" process. Explicit cursor memory area is also called as "**active set area**".

Explicit Cursor Life Cycle:

1. **Declare**
2. **Open**
3. **Fetch**
4. **Close**

1. **Declare:** In "Declare" section of the PL/SQL block we are defining the cursor using following syntax.

```
Syntax: cursor <cursor name> is select * from <table name> where condition; [group by, having... etc.]
```

Example:

```
Sql> declare
cursor c1 is select * from emp where sal > 2000 ;
```

2. **Open:** In all databases whenever we are opening the cursor then only database servers retrieve data from table into cursor memory area because in all database systems when we are opening the cursors then only cursor "Select" statements are executed.

```
Syntax: open <cursor name>;
```

This statement is used in "Executable Section" of the PL/SQL block.

Note: Whenever we are opening the cursor "Implicit" cursor pointer always points to the "First Record" in the cursor.

3. **Fetch:** (Fetching data from cursor memory area)
Using fetch statement we are fetching data from cursor memory area into PL/SQL variables.

```
Syntax: fetch <cursor name> into <variable name1>, <variable name2>,.....;
```

4. **Close:** Whenever we are closing the cursor all the resources allocated from cursor memory area is automatically released.

```
Syntax: close <cursor name>;
```

Example

```
declare
 cursor c1 is select ename, sal from emp;
 v_ename varchar2 (20);
 v_sal   number (10);
begin
 open c1;
 fetch c1 into v_ename, v_sal;
 dbms_output.put_line(v_ename || ' ' || v_sal);
 fetch c1 into v_ename, v_sal;
 dbms_output.put_line('My Second Employee name is :' || v_ename);
 fetch c1 into v_ename, v_sal;
 dbms_output.put_line(v_ename || 'high salary');
 close c1;
end;
/
```

Output

```
Smith    3000
My Second Employee Name Is: Allen
Ward     High Salary
```

Explicit Cursor Attributes

Every explicit cursor having "4" attributes. These are

1. **%notfound**
2. **%found**
3. **%isopen**
4. **%rowcount**

When we are using these attributes in PL/SQL block then we must specify cursor name along with these attributes.

Syntax: cursorname % attributename;

Except "**%rowcount**" all other cursor attributes returns "**Boolean value**" either "**true (or) false**", whereas "**%rowcount**" attributes always returns "**number**" datatype.

1. **%notfound:**

This attribute always returns Boolean value either true (or) false. Whenever "Fetch" statement does not fetches any row from cursor memory area then only "%notfound" attribute returns true.
If a Fetch statement returns at least one row from cursor memory area then "%notfound" returns false.

Syntax: cursorname%notfound;

➢ **Write PL/SQL explicit cursor program to display all employee name and their salary from emp table using %notfound attribute?**

Ans:

```
declare
 cursor c1 is select ename, sal from emp;
 v_ename varchar2(20);
 v_sal   number(10);
begin
 open c1;
 loop
  fetch c1 into v_ename, v_sal;
  exit when c1%notfound;
  dbms_output.put_line(v_ename ||' '|| v_sal);
 end loop;
 close c1;
end;
/
```

➢ **Write a PL/SQL cursor program to display total salary from emp table without using sum functions?**

Ans:

```
declare
 cursor c1 is select sal from emp;
 v_sal number(10);
 n   number(10) := 0;
begin
 open c1;
 loop
  fetch c1 into v_sal;
  exit when c1%notfound;
  n := n + v_sal;
 end loop;
 dbms_output.put_line('Total salary is :' || ' ' || n);
 close c1;
end;
/
```

Output: Total salary is: 29225.

Note: When a resource table have NULL value column and also when we are performing summation based on that column then we must use "NVL()".

Example n:= n+(nvl(v_sal,0));

- **Write PL/SQL program to display first 5 highest salary employees from emp table using "%rowcount" attribute?**

Ans:

```
declare
 cursor c1 is select ename, sal from emp;
 v_ename varchar2(20);
 v_sal   number(10);
begin
 open c1;
 loop
  fetch c1 into v_ename, v_sal;
  dbms_output.put_line(v_ename || ' ' || v_sal);
  exit when c1%rowcount >= 5;
 end loop;
 close c1;
end;
/
```

- **Write PL/SQL program to display even number of records from emp table using "%rowcount" attributes?**

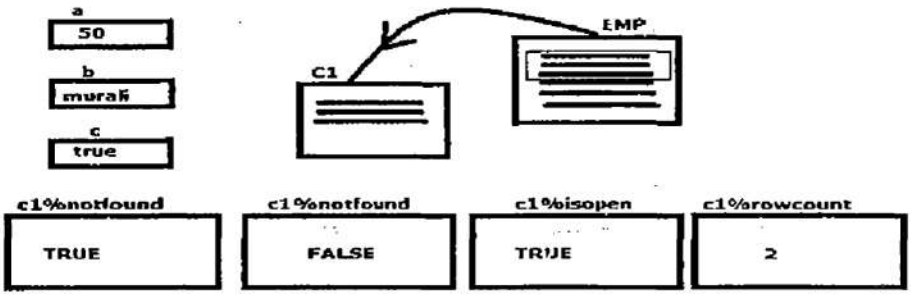
Ans:

```
declare
 cursor c1 is select ename, sal from emp;
 v_ename varchar2(20);
 v_sal   number(10);
begin
 open c1;
 loop
  fetch c1 into v_ename, v_sal;
  exit when c1%notfound;
  if mod(c1%rowcount, 2) = 0 then
   dbms_output.put_line(v_ename || '  '|| v_sal);
  end if;
 end loop;
 close c1;
end;
```

In Oracle, whenever we are creating an explicit cursor then oracle server internally automatically creates "4" memory locations along with cursor memory area. These memory locations behave like a variable. These variables also stores only <u>one value at a time</u>. These variables are identified through "Explicit Cursor Attributes".

<u>Example</u>

```
declare
 a number(10);
 b varchar2(10);
 c  boolean;
begin
 a := 50;
 b := 'murali';
 c := 'true';
end;
```



2. **%found:** This attribute also returns Boolean value either "true" (or) "false". It returns true when fetch statement fetches at least one row from cursor memory area, whereas it returns false when fetch statement does not fetches any row from cursor memory area.

Syntax: cursorname%found;

Example

```
declare
 cursor c1 is select * from emp where ename = '&ename';
 i emp%rowtype;
begin
 open c1;
 fetch c1 into i;
 if c1%found then
   dbms_output.put_line('Requested employee:'||' '||i.ename ||' '|| i.sal);
 else
   dbms_output.put_line(' Your employee does not exist');
 end if;
 close c1;
end;
```

Output

```
Enter value for ename: SMITH
Requested employee: SMITH 3000
Sql>/
Enter value for ename: abc
Your employee does not exist
```

➢ **Write a PL/SQL cursor program which display all employees and their salaries from emp table using %found attributes?**

Ans:

```
declare
 cursor c1 is select * from emp;
 i emp%rowtype;
begin
 open c1;
 fetch c1 into i;
 while (c1%found)
loop
   dbms_output.put_line(i.ename || ' ' || i.sal);
   fetch c1 into i;
 end loop;
 close c1;
end;
```

3. **%rowcount:** This attribute always returns number datatype i.e. it counts number of records fetches from the cursor memory area.

Syntax: cursorname%rowcount;

Example

```
declare
 cursor c1 is select ename, sal from emp;
 v_ename varchar2(10);
 v_sal   number(10);
begin
 open c1;
 fetch c1 into v_ename, v_sal;
 dbms_output.put_line(v_ename || ' ' || v_sal);
 fetch c1 into v_ename, v_sal;
 dbms_output.put_line(v_ename || ' ' || v_sal);
 dbms_output.put_line('Number of records fetched from cursor is :' || ' ' || c1%rowcount);
 close c1;
end;
/
```

Output

```
SMITH 1000
ALLEN 1600
Number of records fetched from cursor is: 2
```

➢ **Write a explicit cursor program to display 5th record from emp table using "%rowcount" attribute?**

Ans:

```
declare
 cursor c1 is select * from emp;
 i emp%rowtype;
begin
 open c1;
 loop
  fetch c1 into i;
  exit when c1%notfound;
  if c1%rowcount = 5 then
   dbms_output.put_line(i.ename || ' ' || i.sal);
  end if;
 end loop;
 close c1;
end;
/
```

Output: MARTIN 1250

Note: By using cursor we can also transfer data from one Oracle table into another Oracle table, arrays and operating system files.

- **Write a PL/SQL cursor program to transfer ename, sal who are getting more than 2000 salary from emp table into another table?**

Ans:

```
sql> create table target(sno number(10), ename varchar2(20), sal number(10));

sql> declare
  cursor c1 is select * from emp where sal > 2000;
  i emp%rowtype;
  n number(10);
begin
  open c1;
  loop
    fetch c1 into i;
    exit when c1%notfound;
    n := c1%rowcount;
    insert into target values (n, i.ename, i.sal);
  end loop;
  close c1;
end;
/
```

Output: select * from target;

Using cursor "for loop" we are eliminating explicit cursor life cycle. Here we are no need to use **"open, fetch, close"** statements explicitly. Whenever we are using cursor **"for loop"** internally oracle server only open the cursor then fetch data from the cursor and close the cursor automatically.

Syntax:

```
for <index variable name> in <cursor name>
loop
statements;
end loop;
```

- Data transferred from cursor to index variable

Note: In cursor "for loop" index variable internally behaves like a record type variable (%rowtype).
This cursor **"for loop"** is used in "executable section" of the PL/SQL block. Cursor **"for loop"** is also called as "shortcut method" of the cursor.

- **Write a PL/SQL cursor program to display all employee names and their salaries from emp table using cursor "For loop"?**

Ans:

```
declare
 cursor c1 is select * from emp;
begin
 for i in c1
 loop
   dbms_output.put_line(i.ename || ' ' || i.sal);
 end loop;
end;
/
```

Note: we can also eliminate **"declare section"** of the cursor by using cursor **"for loop"**. In this case we must specify cursor select statement in place of cursor name within cursor "for loop".

Syntax:

```
for <index variable name> in (select statement)
loop
statements;
end loop;
```

Example

```
begin
 for i in (select * from emp)
loop
   dbms_output.put_line (i.ename ||' '|| i.sal);
 end loop;
end;
/
```

Using cursor "for loop" we are eliminating explicit cursor life cycle. Here we are no need to use "**open, fetch, close**" statements explicitly. Whenever we are using cursor "**for loop**" internally oracle server only open the cursor then fetch data from the cursor and close the cursor automatically.

Syntax:

```
for <index variable name> in <cursor name>
loop
statements;
end loop;
```

- Data transferred from cursor to index variable

Note: In cursor "for loop" index variable internally behaves like a record type variable (%rowtype).
This cursor "**for loop**" is used in "executable section" of the PL/SQL block. Cursor "**for loop**" is also called as "shortcut method" of the cursor.

- **Write a PL/SQL cursor program to display all employee names and their salaries from emp table using cursor "For loop"?**

Ans:

```
declare
 cursor c1 is select * from emp;
begin
 for i in c1
 loop
   dbms_output.put_line(i.ename || ' ' || i.sal);
 end loop;
end;
/
```

Note: we can also eliminate "**declare section**" of the cursor by using cursor "**for loop**". In this case we must specify cursor select statement in place of cursor name within cursor "**for loop**".

Syntax:

```
for <index variable name> in (select statement)
loop
statements;
end loop;
```

Example

```
begin
 for i in (select * from emp)
loop
   dbms_output.put_line (i.ename ||' '|| i.sal);
 end loop;
end;
/
```

➢ **Write a cursor program to display 5th record from emp table by using cursor "For loop"?**

Ans:

```
declare
 cursor c1 is select * from emp;
begin
 for i in c1
 loop
   if c1%rowcount = 5 then
     dbms_output.put_line (i.ename || ' ' || i.sal);
   end if;
 end loop;
end;
/
```

Output: MARTIN 1250

➢ **Write a PL/SQL cursor program which displays total salary from emp table without using sum function with using cursor "For loop"?**

Ans:

```
declare
 cursor c1 is select * from emp;
 n number(10) := 0;
begin
 for i in c1
 loop
   n := n + i.sal;
 end loop;
 dbms_output.put_line('Total salary is :' || n);
end;
/
```

Output: Total salary is: 29250

Example:

```
declare
 cursor c1 is select * from emp;
begin
 for i in c1
 loop
   if i.sal > 2000 then
     dbms_output.put_line(i.ename ||' '|| i.sal);
   else
     dbms_output.put_line(i.ename ||' '|| 'Low salary');
   end if;
 end loop;
end;
/
```

Output:

In Oracle, we can also pass parameters in to the explicit cursor same like a procedure in parameters, this type of cursor is also called as "Parameterized Cursor".

In "Parameterized Cursors" we are defined formal parameter when we are declaring a cursor, whereas we are passing actual parameter when we are opening the cursor.

Note: In Oracle, whenever we are defining formal parameters in **"cursors, procedures, functions"** then we are not allowed to use datatype "size" in formal parameter declaration.

Syntax 1:

```
cursor <cursor name> (<parameter name>  <data type>) is
select * from <table name> where <column name> = <parameter name>;
```

Syntax 2: open <cursor name> (actual parameter);

➢ **Write a PL/SQL parameterized cursor program for passing deptno is a parameter that display employee details from emp table based on passed deptno?**

Ans:

```
declare
 cursor c1(p_deptno number) is
select * from emp where deptno = p_deptno;
 i emp%rowtype;
begin
 open c1(10);
 loop
  fetch c1 into i;
  exit when c1%notfound;
  dbms_output.put_line(i.ename || ' ' || i.sal || '  ' || i.deptno);
 end loop;
 close c1;
end;
/
```

Output:

```
CLARK   3550   10
KING    6200   10
MILLER  3100   10
```

➢ **Write a PL/SQL cursor program using parameterized cursor for passing job as a parameter from emp table then display the employees working as CLERK and ANALYST and also display final output statically by following format?**

```
Employees working as CLERK
        SMITH
        ADAMS
        JAMES
        MILLER
Employees working as ANALYST
        SCOTT
        FORD
```

Ans:

```
declare
 cursor c1(p_job varchar2) is
select * from emp where job = p_job;
 i emp%rowtype;
begin
 open c1('CLERK');
 dbms_output.put_line('Employees working as Clerks');
 loop
  fetch c1 into i;
  exit when c1%notfound;
  dbms_output.put_line(i.ename);
 end loop;
 close c1;

 open c1('ANALYST');
 dbms_output.put_line('Employees working as Analysts');
 loop
  fetch c1 into i;
  exit when c1%notfound;
  dbms_output.put_line(i.ename);
 end loop;
 close c1;
end;
/
```

Note 1: Before we are reopening the cursor we must close the cursor properly otherwise Oracle server returns an error. "ORA-06511: Cursor is already open".

Note 2: When we are not opening the cursor but we are trying to perform operations on the cursor then Oracle server returns an error "ORA-1001: Invalid Cursor".

Example: (Converting to Parameterized cursor Shortcut method of Cursor)

```
declare
 cursor c1 (p_deptno number) is
select * from emp where deptno = p_deptno;
begin
 for i in c1(10)
loop
   dbms_output.put_line (i.ename || ' ' || i.sal || ' ' || i.deptno);
end loop;
end;
/
```

Output:

```
CLARK   3550   10
KING    6200   10
MILLER  3100   10
```

Applying same technique to another example:

```
declare
 cursor c1(p_job   varchar2) is select * from emp where job = p_job;
begin
 dbms_output.put_line ('Employees working as Clerks');
 for i in c1('CLERK')
loop
  dbms_output.put_line (i.ename);
 end loop;
 dbms_output.put_line ('Employees working as Analysts');
 for i in c1('ANALYST')
loop
  dbms_output.put_line (i.ename);
 end loop;
end;
/
```

Output:

```
Employees working as Clerks
        SMITH
        ADAMS
        JAMES
        MILLER
Employees working as Analysts
        SCOTT
        FORD
```

Note: In oracle we can also defined more than one cursor in single PL/SQL block and also we are allow to pass one cursor value into another cursor, whenever we are passing one cursor value into another cursor then receiving cursor must be in "parameterized cursor".

Generally in relational databases when we are implementing master-detailed reports then only we are defining multiple cursors.

- **Write a PL/SQL cursor program to retrieve all deptno from dept table by using explicit cursor and also pass these deptno from this explicit cursor into another parameterized cursor which returns employee details from emp table based on passed deptno?**

Ans:

```
declare
 cursor c1 is select deptno from dept;
 cursor c2 (p_deptno number) is select * from emp where deptno = p_deptno;
begin
 for i in c1
 loop
  dbms_output.put_line ('My Department Number is:' || ' ' || i.deptno);
  for j in c2 (i.deptno)
 loop
   dbms_output.put_line (j.ename || ' ' || j.sal || ' ' || j.deptno);
  end loop;
 end loop;
end;
/
```

Output:

```
My Department Number Is: 10
        CLARK  2450   10
        KING   5000   10
        MILLER 1300   10
My Department Number Is: 20
        SMITH  800    20
        JONES  2975   20
        SCOTT  3000   20
        ADAMS  1100   20
        FORD   3000   20
My Department Number Is: 30
        ALLEN  1600   30
        WARD   1250   30
        MARTIN 1250   30
        BLAKE  2850   30
        TURNER 1500   30
        JAMES  950    30
My Department Number Is: 40
```

Note: In parameterized cursor we can also pass default values by using "default" clause (or) by using := operator.

Syntax: parametername datatype default (or) := defaultvalues;

Example:

```
declare
 cursor c1 (p_deptno number default 30) is select * from emp where deptno = p_deptno;
                                (or)
 cursor c1 (p_deptno number := 30) is select * from emp where deptno = p_deptno;
begin
 for i in c1
loop
  dbms_output.put_line(i.ename ||' '|| i.sal ||' '|| i.deptno);
 end loop;
end;
/
```

Output:

```
ALLEN  1600  30
WARD   1250  30
MARTIN 1250  30
BLAKE  2850  30
TURNER 1500  30
JAMES  950   30
```

In PL/SQL we can also used oracle predefined function or expressions in explicit cursor select statement. In this case we must create alias name for those functions (or) expressions in cursor select statement and also we must declare cursor record type variable (**%rowtype**) in declare section of the PL/SQL block.

Syntax: <variable name> <cursor name> %rowtype;

➢ **Write a PL/SQL cursor program which display total salary from emp table using "sum()"?**

Ans:

```
declare
 cursor c1 is select sum(sal) a from emp;
 i c1%rowtype;
begin
 open c1;
 fetch c1 into i;
 dbms_output.put_line ('Total salary is:' || ' ' || i.a);
 close c1;
end;
/
```

Output: Total salary is: 40425

➢ **Write a PL/SQL cursor program using parameterized cursor for passing deptno as parameter that display total number of employees, total salary, minimum salary, maximum salary of that deptno from emp table?**

Ans:

```
declare
 cursor c1(p_deptno number) is
  select count(*) a, sum(sal) b, min(sal) c, max(sal) d
   from emp
   where deptno = p_deptno;
 i c1%rowtype;
begin
 open c1(&deptno);
 fetch c1 into i;
 dbms_output.put_line('Numbers of employees are :' || ' ' || i.a);
 dbms_output.put_line('Total Salary is :' || ' ' || i.b);
 dbms_output.put_line('Minimum Salary is :' || ' ' || i.c);
 dbms_output.put_line('Maximum Salary is :' || ' ' || i.d);
 close c1;
end;
/
```

Output:

```
Enter value for deptno: 10;
Numbers of employees are : 3
Total Salary is : 10855
Minimum Salary is : 2600
Maximum Salary is : 5400
```

- **Write a PL/SQL cursor program which is used to modify salaries of the employees in emp table based on following conditions:**

  1. If job = 'CLERK' then increment sal --> 100
  2. if job = 'SALESMAN' then decrement sal --> 200

Ans:

```
declare
 cursor c1 is select * from emp;
 i emp%rowtype;
begin
 open c1;
 loop
  fetch c1 into i;
  exit when c1%notfound;
  if i.job = 'CLERK' then
   update emp set sal = sal + 100 where empno = i.empno;
  elsif i.job = 'SALESMAN' then
   update emp set sal = sal - 200 where empno = i.empno;
  end if;
 end loop;
 close c1;
end;
/
```

Output: select * from emp;

For SQL statements returns single record is called implicit cursor. Implicit cursor memory area is also called as context area.

In oracle whenever PL/SQL block having "**select...into**" clause (or) pure **DML** statements then oracle server internally automatically allocates a memory area, this memory area is also called as "**SQL area**" (or) "**context area**" (or) "**implicit area**".

This memory area returns a single record when PL/SQL block contains "Select.... into" clause. This memory area also returns multiple records when PL/SQL block contains pure DML statements, but these multiple records does not controlled by user explicitly i.e. these all records processed at a time by SQL engine.

Example

```
declare
 v_ename varchar2(10);
 v_sal  number(10);
begin
 select ename, sal into v_ename, v_sal from emp where empno = &no;
 dbms_output.put_line(v_ename || ' ' || v_sal);
end;
/
```

Along with this memory area Oracle server internally automatically created "4" memory locations. This memory location behaves like a variable. The variables are identified through "Implicit Cursor Attributes". These Implicit Cursor Attributes are:

1) **sql%notfound**
2) **sql%found**
3) **sql%isopen**
4) **sql%rowcount**

In all databases always "**sql%isopen**" returns false, whereas "**sql%notfound**", "**sql%found**" attributes returns Boolean value either true (or) false and also "**sql%rowcount**" attribute always returns number datatype.

Example

```
begin
 delete from emp where ename = 'welcome';
 if sql%found then
   dbms_output.put_line('u r record is deleted');
 end if;
 if sql%notfound then
   dbms_output.put_line('u r record does not exist');
 end if;
end;
/
```

Output: u r record does not exist

Example

```
begin
 update emp set sal = sal + 100 where job = 'CLERK';
 dbms_output.put_line('Affected number of clerks are :' || ' ' || sql%rowcount);
end;
/
```

Output: Affected numbers of clerks are: 4

Exception is an error occurred during runtime, whenever runtime error is occurred use an appropriate exception name in exception handles under exception section within PL/SQL block.

Oracle has 3 types of "Exceptions".

1. **Predefined exceptions**
2. **User defined exceptions**
3. **Unnamed exceptions**

1. **Predefined exceptions**

Oracle provided 20 predefined exception names for regularly "Occurred runtime errors". Whenever run time errors occurred use appropriate predefined exception name in exception handler under exception section of the PL/SQL block.

```
Syntax:
        when <predefined exception name1> then
        statements;
        when <predefined exception name2> then
        statements;
        ----------------------------
        ----------------------------
        when others then
        statements;
```

Predefined Exception Names

1. **no_data_found** (ORA-1403)
2. **too_many_rows** (ORA-1422)
3. **zero_divide** (ORA-1476)
4. **invalid_cursor** (ORA-1001)
5. **cursor_already_open** (ORA-6511)
6. **dup_val_on_index** (ORA-0001)
7. **invalid_number** (ORA-1722)***
8. **value_error** (ORA-6502)***

1) **no_data_found:** When a PL/SQL blocks having "select....into" clause and also if requested data not available in a table through this clause then Oracle server returns an error.
ORA-1403: no data found. For handling this error then Oracle provided "**no_data_found**" exception name.

Example:

```
declare
 v_ename varchar2 (20);
 v_sal   number (10);
begin
 select ename, sal into v_ename, v_sal from emp where empno = &no;
 dbms_output.put_line (v_sal);
exception
 when no_data_found then
  dbms_output.put_line ('your employee does not exist');
end;
/
```

Output:

```
Enter value for no: 7902
WARD 3000
Enter value for no: 1111
Your employee does not exist
```

2) **too_many_rows:** In PL/SQL blocks whenever "select.....into" clause try to return multiple records from a table (or) try to return multiple values at a time from a single column then Oracle server returns an error.
ORA-1422: exact fetch returns more than requested number of rows.
For handling this error Oracle provided "**too_many_rows**" predefined exception name.

Example:

```
declare
 v_sal number (10);
begin
 select sal into v_sal from emp;
 dbms_output.put_line (v_sal);
exception
 when too_many_rows then
  dbms_output.put_line ('not to return multiple rows');
end;
/
```

Output: not to return multiple rows.

3) **Zero_divide:** In PL/SQL when we are trying to perform division with zero then oracle server returns an error "ORA-1476: division is equal to zero". For handling this error we are using "Zero_divide" exception name.

Example:

```
begin
 dbms_output.put_line(5 / 0);
exception
 when zero_divide then
  dbms_output.put_line('not to perform division with zero');
end;
/
```

Ouptut: not to perform division with zero.

4) **Invalid_cursor:** In Oracle, when we are not "open" the cursor, but we try to perform operations on the cursor then Oracle server returns an error "ORA-1001: Invalid cursor". For handling this error Oracle provided "invalid_cursor" exception name.

Example:

```
declare
 cursor c1 is select * from emp;
 i emp%rowtype;
begin
 loop
  fetch c1 into i;
  exit when c1%notfound;
  dbms_output.put_line (i.ename || ' ' || i.sal);
 end loop;
 close c1;
exception
 when invalid_cursor then
  dbms_output.put_line ('first we must open the cursor');
end;
/
```

Output: first we must open the cursor.

5) **Cursor_already_open:** In Oracle, before we are reopening the cursor then we must close the cursor properly otherwise Oracle server returns an error "ORA-6511: cursor already open".
For handling this error we are using "**cursor_already_open**" exception name.

Example:

```
declare
 cursor c1 is select * from emp;
 i emp%rowtype;
begin
 open c1;
 loop
  fetch c1 into i;
  exit when c1%notfound;
  dbms_output.put_line(i.ename || ' ' || i.sal);
 end loop;
 open c1;
exception
 when cursor_already_open then
  dbms_output.put_line('first close the cursor to reopen the cursor');
end;
/
```

Output: first close the cursor to reopen the cursor.

6) **Dup_val_on_index:** In Oracle, when we try to insert duplicate values into "**Primary key**" (or) "**unique key**" then Oracle server returns an error "ORA-0001: Unique constraint violated".
For handling this error oracle provided "dup_val_on_index" exception name.

Example

```
begin
 insert into emp (empno) values (7369);
exception
 when dup_val_on_index then
  dbms_output.put_line ('not to insert duplicate values');
end;
/
```

Output: not to insert duplicate values.

7) **Invalid_number, Value_error:** In oracle when we are try to convert "**string type**" in to "**number type**" (or) try to convert "**date string**" into "**date type**" then Oracle server returns two types of errors. These are ...

(i) Invalid number
(ii) Value error

(i) **Invalid_Number:** When a PL/SQL block contains SQL statements and also those statements try to convert "string type" into "number type" (or) "date string" into "date type" then Oracle server returns an error "ORA-1722: Invalid number" for handling this error Oracle provided "**invalid_number**" exception name.

Example

```
begin
 insert into emp (empno, ename, sal) values (1, 'shailendra', 'abc');
end;
/
```

Output: ERROR: ORA-1722: invalid number

Solution:

```
begin
 insert into emp (empno, ename, sal) values (1, 'shailendra', 'abc');
exception
 when invalid_number then
  dbms_output.put_line ('insert proper data only');
end;
/
```

Output: insert proper data only

(ii) **Value_Error:** When a PL/SQL block contains procedural statements and also those statements try to convert "string type" into "number type" then Oracle server returns an error.
ERROR: ORA-6502: numeric or value error: character to number conversion error.
For handling this error then we are using "value_error" exception name.

Example:

```
declare
 z number(10);
begin
 z := '&x' + '&y';
 dbms_output.put_line(z);
exception
 when value_error then
  dbms_output.put_line ('enter proper data only');
end;
/
```

Output:

```
Enter value for x: a
Enter value for y: b
Enter proper data only
```

Note: In Oracle, when we are try to store more data then the "varchar2" datatype size specified at variable declaration then also Oracle server returns an error.
Error: "ORA-6502: number (or) value error: character string buffer too small".
For handling this error also then we are using "value_error" exception name.

Example:

```
declare
 z varchar2(3);
begin
 z := 'abcd';
 dbms_output.put_line (z);
exception
 when value_error then
  dbms_output.put_line ('Invalid string length');
end;
/
```

Output: Invalid string length

Note: When we try to store more number of digits than the maximum size specified number data type then also Oracle server returns an error "ORA-6502: numeric (or) value error: number precision too large."
For handling this error also we are using "value_error" exception name.

Example

```
declare
 a number(3);
begin
 a := 99999;
 dbms_output.put_line(a);
exception
 when value_error then
  dbms_output.put_line('not to store more data');
end;
/
```

Output: not to store more data.

In PL/SQL exceptions also occurred in declare section, executable section, exception section.

Whenever exceptions were occurred in executable section then those exceptions are handled either in Inner blocks (or) in Outer block. Where as when exceptions are occurred in declare section (or) in exception section then those exceptions must be handled in "outer blocks" only, this is called "PL/SQL exception propagation".

Example

```
declare
 z varchar2(3) := 'abcd';
begin
 dbms_output.put_line(z);
exception
 when value_error then
  dbms_output.put_line('Invalid string length');
end;
/
```

Output: ERROR: ORA-6502: number or value error: character string buffer too small.

Solution

```
begin
 declare
  z varchar2(3) := 'abcd';
 begin
  dbms_output.put_line (z);
 exception
  when value_error then
   dbms_output.put_line ('Invalid string length');
 end;
exception
 when value_error then
  dbms_output.put_line ('Invalid string length handled using Outer blocks only');
end;
/
```

Output: Invalid string length handled using Outer blocks only.

Solution: 2

```
begin
 declare
  z varchar2(3) := 'abcd';
 begin
  dbms_output.put_line (z);
 end;
exception
 when value_error then
  dbms_output.put_line ('Invalid string length handled using Outer blocks only');
end;
/
```

Output: Invalid string length handled using Outer blocks only.

In oracle we can also create our own exception name and also raise that exception explicitly whenever necessary, these types of exceptions are also called as "user defined exceptions".
In all databases, if we want to return messages based on client business rules then only we are using user defined exceptions.

In PL/SQL we are handling user defined exception by using following 3 steps process.

Step 1: Declare
Step 2: Raise
Step 3: Handling Exception

Step 1: Declare: In declare section of the PL/SQL block we can create our own exception name by using exception predefined type.

```
Syntax: <user defined name> exception;
```

Example

```
sql> declare
     a exception;
```

Step 2: Raise: In PL/SQL we must raise user defined exception either in executable section or in exception section by using "Raise" statement.

```
Syntax: raise <user defined exception name>;
```

Step 3: Handling Exception: We can also handle user defined exceptions same like a predefined exceptions by using exception handler under exception section of the PL/SQL block.

```
Syntax:
        when <user defined exception name1> then
        stmts;
        when <user defined exception name2> then
        stmts;
        ------------------------------
        ------------------------------
        when others then
        stmts;
```

➢ **Write PL/SQL program raise a user defined exception on Wednesday?**

Ans:

```
declare
 a exception;
begin
 if to_char (sysdate, 'DY') = 'WED' then
  raise a;
 end if;
exception
 when a then
  dbms_output.put_line ('my exception raised on Wednesday');
end;
/
```

Output: my exception raised on Wednesday

Example

```
declare
 a exception;
 v_sal number(10);
begin
 select sal into v_sal from emp where empno = 7369;
 if v_sal > 2000 then
  raise a;
 else
  update emp set sal = sal + 100 where empno = 7369;
 end if;
exception
 when a then
  dbms_output.put_line('Salary already high');
end;
/
```

Output: Salary already high

- **Write a PL/SQL program using following table whenever user entered empno is more than 5 or less than 1 (<1) then raise a user defined exception and also whenever user entered empno is available in a table then display name of the employee from that table based on user entered empno?**

```
sql> create table test (empno number(10), ename varchar2(10));
sql> insert into test values(.....);
sql> select * from test;
```

EMPNO	ENAME
1	SMITH
2	ALLEN
3	WARD
4	JONES
5	MARTIN

Ans:

```
declare
 a exception;
 v_ename varchar2(10);
 v_empno number(10);
begin
 v_empno := &empno;
 if v_empno > 5 or v_empno < 1 then
  raise a;
 else
  select ename into v_ename from test where empno = v_empno;
  dbms_output.put_line (v_ename);
 end if;
exception
 when a then
  dbms_output.put_line ('u r empno is out of range');
end;
/
```

Output:

```
Enter value for empno: 8
u r empno is out of range
SQL> /
Enter value for empno: 2
ALLEN
```

Note: In oracle whenever requested data not available in a table by using explicit cursor program then oracle server does not return any message. To overcome this problem we must use "**user defined exception**" in an explicit cursor.

Example:

```
declare
 cursor c1 is select * from emp where deptno = &deptno;
 i emp%rowtype;
 a exception;
begin
 open c1;
 loop
  fetch c1 into i;
  exit when c1%notfound;
  dbms_output.put_line(i.ename || ' ' || i.sal || ' ' || i.deptno);
 end loop;
 if c1%rowcount = 0 then
  raise a;
 end if;
 close c1;
exception
 when a then
  dbms_output.put_line('u r deptno does not exist');
end;
/
```

Output:

```
Enter value for deptno: 10
.............................
.............................
Sql>/
Enter value for deptno: 90
U r deptno does not exist
```

If we want to handle other than Oracle 20 predefined exception name errors then we are using **"Unnamed Exceptions"**. In this method we are creating our own exception name and associate that exception name with appropriate error number by using "EXCEPTION_INIT()".

Syntax: pragma exception_init (<user defined exception name>, <error number>);

This function is used in declare section of the PL/SQL block. Here "pragma" is a "Compiler Directive" i.e. whenever we are using this pragma then Oracle server internally associates error number with exception name at compile time.

Example: 1

```
begin
 insert into emp (empno, ename) values (null, 'shailendra');
end;
/
```

Output: Error: ORA-01400: cannot insert NULL into EMPNO

Solution:

```
declare
 a exception;
 pragma exception_init(a, -1400);
begin
 insert into emp (empno, ename) values (null, 'shailendra');
exception
 when a then
  dbms_output.put_line ('not to insert null values');
end;
/
```

Output: not to insert null values

Example: 2

```
Begin
delete from dept where deptno = 10;
end;
/
```

Solution:

```
declare
 a exception;
 pragma exception_init (a, -2292);
begin
 delete from dept where deptno = 10;
exception
 when a then
  dbms_output.put_line ('not to delete master records');
end;
/
```

Output: not to delete master records

➢ **Write a PL/SQL program to handle -2291 error number by using exception_init() based on emp, dept table?**

Ans:

```
declare
 a exception;
 pragma exception_init(a, -2291);
begin
 insert into emp (empno, ename, deptno) values (1, 'abc', 50);
exception
 when a then
  dbms_output.put_line ('not to insert other than primary key values');
end;
/
```

Output: not to insert other than primary key values

Raise_Application_Error()

Raise_application_error is an oracle predefined exception procedure is available in **"dbms_standard"** package (sql> desc dbms_standard).

In oracle if you want to display user defined exception messages in more descriptive form then only we are using this procedure i.e. if you want to display user defined exception messages as same as oracle error displayed format then we must use **"raise_application_error"** procedure.

Raise_application_error is used either in executable section or in exception section of the PL/SQL block. This procedure accepts two parameters.

```
Syntax: raise_application_error (<error number>, <message>);
```

Error number → between -20000 to -20999
Message →up to 512 characters

Example:

```
declare
 a exception;
begin
 if to_char(sysdate, 'DY') = 'FRI' then
  raise a;
 end if;
exception
 when a then
  raise_application_error(-20234, 'My exception raised on Friday');
end;
/
```

Output: ORA-20234: my exception raised on Friday

Note: Raise_application_error is an exception procedure that's why when condition is true it raise a message and also stop the execution and also control does not goes to remaining program.

Whereas "put_line" is normal procedure whenever condition is true it display message and also control goes to remaining program.

Difference between dbms_output.put_line & raise_application_error()

Put_line procedure is a normal procedure whenever we are raising a message conditionally then it raise a message when condition is true and also control goes to remaining program,

Whereas "**raise_application_error**" is an exception procedure that's why when condition is true it raise a message and also control doesn't goes to remaining PL/SQL block.

Example 1: (put_line)

```
begin
 if to_char(sysdate, 'DY') = 'FRI' then
   dbms_output.put_line('my exception raised on Friday');
   dbms_output.put_line('tomorrow there is no class');
 end if;
end;
/
```

Output:

my exception raised on Friday
tomorrow there is no class

Example 2 : (raise_application_error)

```
begin
 if to_char(sysdate, 'DY') = 'FRI' then
   raise_exception_error(-20324, 'my exception raised on Friday');
   dbms_output.put_line('tomorrow there is no class');
 end if;
end;
/
```

Output: ORA-20324: my exception raised on Friday

Note: Generally, "raise_application_error" procedure used in triggers because whenever condition is true it raises a message and also it stops invalid data entry according to condition because this is an exception procedure. Whereas "**dbms_output.put_line**" procedure doesn't stop invalid data entry into our table when condition is true also.

Example:

```
begin
 raise_application_error (-20123, 'first error message');
exception
 when others then
   raise_application_error (-20456, 'second error message');
end;
/
```

Output: ORA-20456: second error message.

In PL/SQL block whenever we are using more than one "**raise_application_error**" procedure then oracle server returns last "**raise_application_error**" procedure message, because "**raise_application_error**" procedure internally having an optional 3rd parameter.

This parameter returns Boolean value by default this parameter returns "**false**", that's why it replaces all previous "**raise_application_error**" procedure messages.

To overcome this problem if you want to display all above "**raise_application_error**" message we must use Boolean value "**true**" with 3rd parameter.

Syntax: raise_application_error (<error number>, <message>, <Boolean>);

Example:

```
begin
 raise_application_error (-20123, 'first error message');
exception
 when others then
   raise_application_error (-20456, 'second error message', false);
end;
/
```

Output: ORA-20456: second error message.

Solution:

```
begin
 raise_application_error (-20123, 'first error message');
exception
 when others then
   raise_application_error (-20456, 'second error message', true);
end;
/
```

Output:

ORA-20456: second error message.
ORA-20123: first error message.

In PL/SQL if you want to catch oracle error number, error number with error messages then oracle provided 2 error trapping function in PL/SQL these are:

1) SQLCODE
2) SQLERRM

These two predefined functions are used in "when others then" clause (default handler) or we can also use within our own exception handler.

In Oracle, if we want to know which run time error is occurred for a particular PL/SQL block at execution time then we must use SQLCODE().

Example:

```
declare
 v_sal number(10);
begin
 select sal into v_sal from emp;
 dbms_output.put_line (v_sal);
exception
 when others then
   dbms_output.put_line (SQLCODE);
   dbms_output.put_line (SQLERRM);
end;
/
```

Output:

```
-1422
ORA-01422: exact fetch returns more than requested number of rows
```

Note: Generally if you want to find out which error is occur at runtime then we must use SQLCODE() and also by using SQLCODE() we are allow to handle exception without using predefined exception name.

In PL/SQL in place of default message by using SQLCODE() we are allow to collect possibility exception and by using these exception we are allow to handle exceptions within exception section by using following 3 steps.

Step 1:

```
declare
 v_sal number(10);
begin
 select sal into v_sal from emp where deptno = &deptno;
 dbms_output.put_line (v_sal);
exception
 when no_data_found then
   dbms_output.put_line ('u r deptno does not exist');
 when others then
   dbms_output.put_line ('any error occurs');
end;
/
```

Output:

```
Enter value for deptno: 90
U r deptno does not exist.
Sql> /
Enter value for deptno: 10
Any error occurs.
```

Step 2 (Collecting error number)

```
declare
 v_sal number (10);
begin
 select sal into v_sal from emp where deptno = &deptno;
 dbms_output.put_line (v_sal);
exception
 when no_data_found then
  dbms_output.put_line ('u r deptno does not exist');
 when others then
  dbms_output.put_line (SQLCODE);
end;
/
```

Step 3 (Handling Exception)

```
declare
 v_sal number (10);
begin
 select sal into v_sal from emp where deptno = &deptno;
 dbms_output.put_line (v_sal);
exception
 when no_data_found then
  dbms_output.put_line ('u r deptno does not exist');
 when others then
  if SQLCODE = -1422 then
   dbms_output.put_line ('not to return multiple records');
  elsif SQLCODE = -1722 then
   dbms_output.put_line ('enter proper data only');
  end if;
end;
/
```

Output:

```
Enter value for deptno: 90
U r deptno does not exist.
Sql> /
Enter value for deptno: 10
Not to return multiple records.
Sql> /
Enter value for deptno: a
Enter proper data only.
```

Note: In Oracle, we are not allowed to use SQLCODE, SQLERRM functions directly in DML statements. If we want to use these functions in DML then first we must declare variables in declare section of PL/SQL block and then assign values into variables and then only those variables are used in DML statements.

➢ **Write a PL/SQL program which stores error number, error number with error message of a PL/SQL block into another table? *****

Ans:

```
sql> create table target(errno  number(10), errmsg  varchar2 (200));

declare
 v_errno  number(10);
 v_errmsg  varchar2(200);
 v_sal  number(10);
begin
 select sal into v_sal from emp;
exception
 when others then
  v_errno := SQLCODE;
  v_errmsg := SQLERRM;
  insert into target values (v_errno,  v_errmsg);
end;
/

sql> select * from target;
```

SQLCODE Return Values

SQLCODE return values:	Meaning
0	No Errors
Negative (-ve)	Oracle errors
100	No data found
1	User Defined Exceptions

Generally, SQLCODE() returns numbers where as SQLERRM() returns error number with error messages.

Example:

```
declare
 a exception;
begin
 raise a;
exception
 when a then
  dbms_output.put_line (sqlcode);
  dbms_output.put_line (sqlerrm);
end;
/
```

Output: 1
User Defined Exception

Note: In Oracle, if we want to view special meaning of the SQLCODE return values then we must pass SQLCODE (or) SQLCODE return value into SQLERRM function within executable section of the PL/SQL block.

Example:

```
begin
 dbms_output.put_line (SQLERRM (SQLCODE));
 dbms_output.put_line (SQLERRM (100));
 dbms_output.put_line (SQLERRM (1));
 dbms_output.put_line (SQLERRM (-1722));
end;
/
```

Output:

```
ORA-0000: normal, successful completion
ORA-01403: no data found
User Defined Exception
ORA-01722: Invalid Number
```

Tracing Error Line Number in PL/SQL ***

In oracle if you want to find out line number where the error is occur in PL/SQL block then we must use **"format_error_backtrace"** from **"dbms_utility"** package. This function is used in default handler under exception section of the PL/SQL block.

Syntax: dbms_utility.format_error_backtrace();

Example

```
declare
 a number(10);
 b number(10);
 c number(10);
begin
 a := 5;
 b := 0;
 c := a / b;
 dbms_output.put_line(c);
exception
 when others then
  dbms_output.put_line (dbms_utility.format_error_backtrace);
end;
/
```

Output: ORA-06512: at line 8

1. **Exception Raised in Executable section: *****

When exception is raised in executable section those exceptions are handled either in Inner block (or) in Outer block.

Method 1: Handled Using Inner block

Example:

```
declare
 a exception;
begin
 raise a;
exception
 when a then
  dbms_output.put_line ('Handled using Inner block');
end;
/
```

Output: Handled using Inner block

Method 2: Handled by using Outer block

Example:

```
declare
 a exception;
begin
 begin
  raise a;
 end;
exception
 when a then
  dbms_output.put_line ('Handled by using Outer block');
end;
/
```

Output: Handled by using Outer block.

2. **Exception Occur in Exception section**: ***

In PL/SQL when exceptions are occurring in exception section those exceptions are handled in outer blocks only.

Q) Handle following PL/SQL block by using outer block?

Example

```
declare
 z1 exception;
 z2 exception;
begin
 raise z1;
exception
 when z1 then
  dbms_output.put_line ('Z1 is handled');
  raise z2;
end;
/
```

Output:

```
Z1 is handled
ERROR: unhandled user_defined exception
```

Solution

```
declare
 z1 exception;
 z2 exception;
begin
 begin
  raise z1;
 exception
  when z1 then
   dbms_output.put_line ('Z1 is handled');
   raise z2;
 end;
exception
 when z2 then
  dbms_output.put_line ('Z2 is handled');
end;
/
```

Output:

```
Z1 is handled
Z2 is handled
```

Note: in oracle we can also raise predefined exception explicitly by using raise statement.

Syntax: raise <predefined exception name>;

Note: In oracle whenever we are using explicit cursor if requested data not available in a table then oracle server does not return any error message.

To overcome this problem for handling this block we are using either user defined exception or by using predefined exception "**no_data_found**" by using raise statement.

➢ **Write a PL/SQL explicit program if user enter deptno does not exist in emp table then handled exception by using no_data_found through raise statement?**

Ans:

```
declare
 cursor c1 is select * from emp where deptno = &deptno;
 i emp%rowtype;
begin
 open c1;
 loop
  fetch c1 into i;
  exit when c1%notfound;
  dbms_output.put_line (i.ename || ' ' || i.sal || ' ' || i.deptno);
 end loop;
 if c1%rowcount = 0 then
  raise no_data_found;
 end if;
 close c1;
exception
 when no_data_found then
  dbms_output.put_line (' u r requested deptno is not available in emp table');
end;
/
```

Output:

```
Enter value for deptno: 90
u r requested deptno is not available in emp table.
```

Sub programs are name PL/SQL block which is used to solve particular task. Oracle has two types of sub programs.

1) **Procedures** →**May or may not return value**
2) **Functions** →**Must return a value**

Stored Procedure:

Procedure is a named PL/SQL block which is used to solve particular task and also procedure may or may not return a values.

In Oracle, whenever we are using create or replace keyword in front of the procedure then those procedures are internally automatically, permanently stored in database, that's why these procedures are also called as "Stored Procedures".

Generally, procedures are used to improve performance of the application because in all database procedures internally having one time compilation. By default one time compilation program units improves performance of the application.

In oracle every Procedure having two parts these are:

1) **Procedure Specification**
2) **Procedure Body**

In Procedure specification we are specifying name of the procedure and type of the parameters
Whereas a Procedure body we are solving actual task.

Syntax:

```
Create [or replace] procedure <procedure name> (formal parameters)
is/as
variable declarations, cursor, user defined exception;
begin
...........
...........
[exception]
...........
...........
end [procedure name];
```

Formal Parameters

Syntax: <parameter name> [mode] datatype;

Different kind of modes: IN, OUT, IN OUT

To view errors:

Syntax: show errors;

Executing a Procedure:

Method 1: (Using Bind Variable)

Syntax: exec <procedure name> (actual parameters);

Method 2: (Using Anonymous block)
Syntax:

```
begin
procedurename (actual parameters);
end;
/
```

Method 3: (Using Call Statement)
Syntax: call procedurename (actual parameters);

- **Write a PL/SQL stored procedure program for passing empno as a parameter that display name of the employee and his salary from emp table?**

Ans:

```
create or replace procedure p1 (p_empno 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;
/
```

Execution:

Method 1: (Using Bind Variable)

```
sql> exec p1(7902);
FORD 4000
```

Method 2: (Using Anonymous block)

```
sql> begin
     p1(7902);
     end;
     /
FORD 4000
```

Method 3: (Using Call Statement)

```
sql> call p1(7902);
FORD  4000
```

In Oracle, all stored Procedures information stored under "user_procedures", "user_source" data dictionaries. If we want to view code of the procedure then we are using "user_source" data dictionary.

Example:

```
sql> desc user_source;
sql> select text from user_source where name= 'P1';
```

➢ **Write a PL/SQL stored procedure program for passing deptno as a parameter then display employee details from emp table based on passed deptno?**

Ans:

```
create or replace procedure p1 (p_deptno number) is
 cursor c1 is select * from emp where deptno = p_deptno;
 i emp%rowtype;
begin
 open c1;
 loop
  fetch c1 into i;
  exit when c1%notfound;
  dbms_output.put_line (i.ename || ' ' || i.sal || ' ' || i.deptno);
 end loop;
 close c1;
end;
```

Output:

```
sql> exec p1(10);
CLARK 3750 10
KING 6400 10
MILLER 3700 10
```

Procedure parameters:

Procedure parameter is a name which is used to pass values into the procedure and also return values from the procedure.

Every procedure has two types of parameters these are ...

1) **Formal Parameters**
2) **Actual Parameters**

1. **Formal Parameters:**
   Formal Parameters must be specified in procedure specification.
   Formal Parameters specifies name of the parameter, type of the parameter, mode of the parameter.

Syntax: parametername [mode] datatype;

Note: In Oracle, whenever we are defining formal parameters in CURSORS, PROCEDURES, FUNCTIONS then we are not allowed to specify datatype size in formal parameter declaration.

MODE: Mode specifies purpose of the parameter. Oracle procedures have three types of modes.

a) **IN**
b) **OUT**
c) **IN OUT**

a) **IN mode:** In Oracle, by default parameter mode is "IN" mode. IN mode is used to pass values into procedure body. IN mode behaves like a constant in procedure body. In this parameter we cannot assign new value in procedure body. This parameter behaves like a read only value.

Syntax: parametername in datatype;

➢ **Write a PL/SQL stored procedure program to insert a record into dept table by using "IN" parameters?**

Ans:

```
create or replace procedure p1 (p_deptno in number, p_dname in varchar2, p_loc in varchar2)
is
begin
   insert into dept values (p_deptno, p_dname, p_loc);
   dbms_output.put_line ('u r record inserted through procedures');
end;
/
```

Output:

```
sql> exec p1(1, 'a', 'b');
U r record inserted through procedures.
```

b) **OUT mode:*** OUT parameter is used to return values from the procedure body. OUT parameter internally behaves like an "uninitialized" variable in procedure body. Here explicitly we must specify OUT keyword.

Syntax: parametername out datatype;

Example:

```
create or replace procedure p1(a in number, b out number)
is
begin
  b := a * a;
end;
```

In Oracle, when a sub program having OUT, IN OUT parameters then those type of sub programs are executed using following '2' methods.

Method 1: Using Bind variable
Method 2: Using Anonymous block

Method 1: Using Bind Variable:

```
Sql> variable x number;
Sql> exec p1(8, :x);
Sql> print x;
```

Output: 64

Method 2: Using Anonymous block:

```
declare
  x number(10);
begin
  p1(8, x);
  dbms_output.put_line(x);
end;
/
```

Output: 64

- **Write a PL/SQL stored procedure program for passing ename as IN parameter then return salary of the employee using OUT parameter from emp table?**

Ans:

```
create or replace procedure p1(p_ename in varchar2, p_sal out number)
is
begin
 select sal into p_sal from emp where ename = p_ename;
end;
/
```

Execution:

Method 1: Using Bind Variable

```
Sql> variable a number;
Sql> exec p1('SMITH', :a);
Sql> print a;
Output: 2400
```

Method 2: Using Anonymous block

```
declare
b number(10);
begin
p1('SMITH', b);
dbms_output.put_line (b);
end;
/
```

Output: 2400

- **Write a PL/SQL stored procedure for passing deptno as IN parameter then return dname, loc using OUT parameter from dept table?**

Ans:

```
create or replace procedure p1(p_deptno in number, p_dname out varchar2, p_loc out varchar2)
is
begin
select dname, loc into p_dname, p_loc from dept where deptno=p_deptno;
end;
/
```

Execution:

Method 1: Using Bind Variable

```
Sql> variable x varchar2(10);
Sql> variable y varchar2(10);
Sql> exec p1(10, :x, :y);
Sql> print  x  y;
```

Output:

X	Y
ACCOUNTING	NEW YORK

Method 2: Using Anonymous block

```
declare
x varchar2(10);
y varchar2(10);
begin
p1(30,x,y);
dbms_output.put_line(x||' '||y);
end;
/
```

Output: SALES CHICAGO

- **Write a PL/SQL stored procedure program for passing deptno as IN parameter then return number of employees number in that deptno by using OUT parameter from emp table?**

Ans:

```
create or replace procedure p1(p_deptno in number, p_count out number)
is
begin
select count(*) into p_count from emp where deptno=p_deptno;
end;
/
```

Execution: Using Bind variable

```
Sql> variable a number;
Sql> exec p1(10,:a);
Sql> print a;
        A
----------
        3
```

Ans:

```
create or replace procedure p1(p_deptno in number, p_count out number)
is
begin
   select count(*) into p_count from emp where deptno=p_deptno;
end;
/
```

Execution: (By using bind variable)

```
Sql> variable a number;
Sql> exec p1(10,:a);
Sql> print a;
```

Output: 3

c) **IN OUT Mode:** This mode is used to pass values into procedure and also return values from the procedure, because it is the combination of IN, OUT parameter. IN OUT parameter internally behaves like a constant initialized variable in a procedure body.

Syntax: parametername in out datatype;

Example:

```
create or replace procedure p1(a in out number)
is
begin
a := a*a;
end;
/
```

Execution:

Method 1: (Using Bind variable)

```
Sql> variable x number;
Sql> exec :x := 8;   → initialized variable
Sql> exec p1(:x);
Sql> print x;
```

Output : 64

Method 2: (Using anonymous block)

```
declare
x number(10):=&x;
begin
p1(x);
dbms_output.put_line(x);
end;
/
```

Output:

```
Enter value for x: 4
16
```

➢ **Write a PL/SQL stored procedure for passing empno as a parameter then return salary for the emp by using IN OUT parameter from emp table?**

Ans:

```
create or replace procedure p1(a in out number)
as
begin
select sal into a from emp where empno=a;
end;
/
```

Execution:

Method 1: Using Bind variable

```
Sql> variable a number;
Sql> exec :a:=7902;
Sql> exec p1(:a);
Sql> print a;
```

Output:

```
         A
---------
      4000
```

Method 2: Using Anonymous block

```
declare
b number(10):= &no;
begin
p1(b);
dbms_output.put_line(b);
end;
/
```

Note: In oracle database if you want to pass default values into oracle then we must use either **default** (or) := operator within procedure, function **IN** parameter.

Syntax: parametername in datatype default (or) := defaultvalue;

Example:

```
create or replace procedure p1(p_deptno in number,
                               p_dname  in varchar2,
                               p_loc  in varchar2 default 'hyd')
is
begin
 insert into dept values (p_deptno, p_dname, p_loc);
end;
/
```

Execution:

```
Sql> exec p1(1, 'software');
Sql> select * from dept;
```

Oracle Procedure "IN" Parameter Execution methods

In Oracle, procedure IN parameter having three types of execution methods. These are ...

1) **Positional Notations**
2) **Named Notations (=>)**
3) **Mixed Notations**

Example:

```
create or replace procedure p1(p_deptno in number, p_dname in varchar2, p_loc in varchar2)
is
begin
insert into dept values(p_deptno,p_dname,p_loc);
end;
/
```

Execution:

Using Positional Notation:

```
Sql> exec p1(5, 'x', 'y');
```

Named notations:

```
Sql> exec p1(p_dname => 'x', p_loc => 'y', p_deptno => 8);
```

Mixed Notations: It is the combination of Positional, Named notations. Generally after Positional these can be all Named notations but after Named there cannot be Positional.

Autonomous Transactions ***

Autonomous transactions are independent transaction used in anonymous blocks, procedure, functions and triggers.

Generally we can also use autonomous transactions in child procedure. "Autonomous procedures" are independent procedures these procedures are never affected from the main transaction COMMIT, ROLLBACK command.

If you want to make a procedure autonomous then we are using Autonomous_transaction pragma, commit.

Syntax: pragma autonomous_transaction;

This pragma is used in declare section of the procedure (or) declare section of the anonymous block (or) declare section of the triggers.

Syntax:

```
create or replace procedure procedurename(formal parameter)
is/as
pragma autonomous_transaction;
begin
-----------------
-----------------
Commit/rollback;
[Exception]
-----------------
-----------------
end [procedurename];
```

In oracle whenever we are calling a procedure within calling program then oracle server will not create separate transaction for procedure i.e. procedure transactions are affects within calling environment i.e. in this case procedure transaction are affected in main transaction.

To overcome this problem oracle 8.1.6 introduced autonomous transaction in procedure. When a procedure is autonomous then oracle server creates separate transactions for procedure i.e. procedure commit & rollback command are never affected within calling program and also calling program transaction never affected within procedure.

Example 1: Using Autonomous Transactions

```
sql> create table test(name varchar2(10));

sql> create or replace procedure p1
is
pragma autonomous_transaction;
begin
insert into test values('india');
commit;
end;
/
Procedure created
```

Calling Program

```
begin
insert into test values('hyd'):
insert into test values('mumbai');
p1;
rollback;
end;
/

Sql> select * from test;
Output:
        NAME
        -------
          india
```

Example 2: Without Using Autonomous Transactions

```
Sql> delete from test;
     1 row deleted
Sql> select * from test;
     No row selected

sql> create or replace procedure p1
     is
     begin
     insert into test values('india');
     commit;
     end;
     /
```

Calling Program

```
begin
insert into test values('hyd');
insert into test values('mumbai');
p1;
rollback;
end;
/
Sql> select * from test;
```

Output:

```
NAME
----------
hyd
mumbai
india
```

In Oracle, when a procedure having **commit** and also when we are calling this procedure in a PL/SQL block then this procedure commit not only saves procedure transactions but also saves all the above procedure transactions.

To overcome this problem Oracle 8.1.6 introduced autonomous transactions. When we are using this transaction in child procedures then autonomous procedure transactions are never affected in main transactions.

Autonomous Transactions in Anonymous block

SESSION 1	SESSION 2

Main Transaction
sql> create table test(sno number (10));
sql> insert into test values(1);
sql> insert into test values(2);
sql> select * from test;
SNO
--- 	
1
2
Child Transaction(autonomous transaction)
sql> declare
pragma autonomou_transaction;
begin
for i in 3..10
loop
insert into test values(i);
end loop;
commit;
end;
/
sql> select * from test;
SNO
--- 	
1
2
3
4
5
6
7
8
9
10
sql> rollback;
sql> select * from test;
SNO
--- 	
3
4
5
6
7
8
9
10
	sql> select * from test;
	No rows selected
	
	sql> select * from test;
	SNO
	--- 	
	3
	4
	5
	6
	7
	8
	9
	10

In oracle if you want to view procedure permission from one user to another user then we are using execute object privilege by using following syntax.

Syntax: grant execute on procedurename to username;

In oracle whenever we are executing a procedure by default oracle server execute that procedure with the privileges of the owner.

Example: 1

sql> conn scott/tiger [owner (or) definer]	sql> conn murali/murali [invoker]
sql> create table test(name varchar2(10));	sql> create table test(name varchar2(10));

sql> create or replace procedure p1(a in varchar2)
is
begin
insert into test values(a);
commit;
end;	

sql> grant execute on p1 to murali; sql> select * from test; NAME ..... hyd	sql> exec scott.p1('hyd'); sql> select * from test; no row selected

In oracle by default oracle server execute procedure with the privilege of owner. To overcome this problem if you want to execute procedure with the privileges of invoker then oracle 8i introduced "authid current_user" within procedure, function. This clause is used in specification of the procedure.

Syntax:

```
create or replace procedure procedurename (formal parameters)
authid current_user
is/as
.................
begin
.................
[exception]
.................
end[procedurename];
```

Example: 2

sql> conn scott/tiger [owner (or) definer]	sql> conn murali/murali [invoker]

sql> create table test(name varchar2(10));
sql> create or replace procedure p1(a in varchar2)
authid current_user
is
begin
insert into test values(a);
commit;
end;
sql> grant execute on p1 to murali;
sql> select * from test;
no row selected
	sql> create table test(name varchar2(10));
	sql> exec scott.p1('holiday);
	sql> select * from test;
	NAME
	.....
	holiday
	
	<u>When we are copying a table</u>
	Sql> drop table test;
	Sql> exec scott.p1('holiday');
	Error: table or view does not exist.

Oracle 12C introduced accessible by clause in procedure for providing security of our procedure from the calling program. Accessible by clause is used in specification of the procedure.

Syntax:

```
create or replace procedure procedurename (formal parameters)
accessible by (anotherprocedurename1, anotherprocedurename2,.....)
is/as
begin
.....
end [procedurename];
```

12C

```
sql> create or replace procedure myproc
accessible by(p2)
is
begin
dbms_output.put_line('my procedure');
end;
/
```

Calling Programs

```
sql> create or replace procedure p1
is
begin
myproc
end p1;
/
Error:

sql> create or replace procedure p2
is
begin
myproc
end p2;
/

sql> exec p2;
my procedure
```

Whenever a procedure having "**accessible by clause**" then we are not allow to execute that procedure but we are allow to call the procedure from specified calling program, those calling programs are not available when we are creating "**accessible procedure**" also then oracle server does not return any error because compile time oracle server does not check calling program.

In oracle we can also drop procedure by using: **drop procedure procedurename.**

Function is a named PL/SQL block which is used to solve particular task and also function must return a value. In oracle user defined functions having two types.

1) **Function Specification**
2) **Function Body**

In function specification we are specifying name of the function and type of the parameters, whereas in function body we are solving actual task.

Syntax:

```
create or replace function functionname (formal parameter)
return datatype
is/as
variable declarations, cursors, user defined exception;
begin
---------
---------
return expression;
[exception]
---------
---------
end [function name];
```

Executing a function:

Method 1: (Using Select Statement)

When a function does not have parameters (or) when a function having all "IN" parameters then those functions is allowed to execute using "SELECT" statement.

```
Syntax: select functionname (actual parameters) from dual;
```

Method 2: (Using Anonymous Block)

```
Syntax:
begin
variablename := functionname(actual parameter);
end;
```

Example:

```
create or replace function f1(a varchar2)
return varchar2
is
begin
return a;
end;
/
```

Execution

Method 1: Using Select Statement

```
Sql> select f1('hi') from dual;
Output: hi
```

Method 2: Using Anonymous Block

```
declare
z varchar2;
begin
z:= f1('welcome');
dbms_output.put_line(z);
end;
/
```

Output: Welcome

- **Write a PL/SQL stored function for passing number as a parameter then return a message either even or odd based on the passed number?**

Ans:

```
create or replace function f1(a number)
return varchar2
is
begin
 if mod(a, 2) = 0 then
  return 'even';
 else
  return 'odd';
 end if;
end;
/
```

Execution:

Method 1: Using Select Statement

```
Sql> select f1(5) from dual;
Odd
```

Method 2: Using Anonymous Block

```
declare
 z varchar2(10);
begin
 z := f1(9);
 dbms_output.put_line(z);
end;
/
```

Output: odd

Method 3: Using Bind Variable

```
Sql> variable x varchar2(10);
Sql> begin
:x := f1(10);
end;
sql> print x;

Output: even
```

Method 4: Using "Packaged Procedure"

```
Sql> exec dbms_output.put_line(f1(3));
Output: odd
```

Method 5:

```
Sql> begin
dbms_output.put_line(f1(7));
end;
/
Output: odd
```

Note: In Oracle, we can also use "User Defined Functions" in DML statements.

Example:

```
sql> create table test(msg varchar2(10));
Sql> insert into test values (f1 (8));
1 row created
Sql> select * from test;
        MSG
        ----------
        Even
```

DML statements are used in functions

In oracle we can also use DML statements in user defined functions but we are not allow to execute those function by using "select" statements where as we are allow to execute those functions by using anonymous block.

- **Write a PL/SQL stored function for passing empno as a parameter from emp table then delete that employee record in emp table and also return deleted number of records number from emp table?**

Ans:

```
create or replace function f1(p_empno number)
return number
is
 v_count number(10);
begin
 delete from emp where empno = p_empno;
 v_count := sql%rowcount;
 return v_count;
end;
/
```

Execution:

Method 1: Using Select Statement

```
Sql> select f1(1) from dual;
Error: cannot perform a DML operation inside a query.
```

Method 2: Using Anonymous block

```
declare
x number(10);
begin
x:= f1(1);
dbms_output.put_line(x);
end;
/
```

Output: 0

Evaluating expressions:

In relational databases if you want to evaluate expression then we must use user defined functions. We can also call these functions within store procedure.

➢ **Write a PL/SQL store function for passing salary is a parameter which is used to calculate bonus of the employee based on following condition?**

- If sal < 1000 then bonus →5% of sal
- If sal >= 1000 and sal < 2000 then bonus →10% of sal
- If sal >= 2000 and sal < 3000 then bonus →20% of sal
- If sal >= 3000 and sal < 5000 then bonus →30% of sal

Ans:

```
create or replace function bonus1(sal number)
return number
is
 b number(10);
begin
 if sal < 1000 then
  b := sal * 0.05;
 elsif sal >= 1000 and sal < 2000 then
  b := sal * 0.10;
 elsif sal >= 2000 and sal < 3000 then
  b := sal * 0.20;
 elsif sal >= 3000 and sal < 5000 then
  b := sal * 0.30;
 else
  b := sal * 0.40;
 end if;
 return b;
end;
/
```

➢ **Write a PL/SQL stored procedure for passing empno as a parameter from emp table and also display empno, ename, sal, bonus by using above user defined function and also display final output based on following format?**

- My employee no is:
- My employee name is:
- My employee salary is:
- My employee bonus is:

Ans:

```
create or replace procedure p1(p_empno number)
 is
  v_ename varchar2(10);
  v_sal   number(10);
  v_bonus number(10);
begin
  select ename, sal into v_ename, v_sal from emp where empno = p_empno;
  v_bonus := bonus1(v_sal);
  dbms_output.put_line('My employee no is:' || ' ' || p_empno);
  dbms_output.put_line('My employee name is:' || ' ' || v_ename);
  dbms_output.put_line('My employee salary is:' || ' ' || v_sal);
  dbms_output.put_line('My employee bonus is:' || ' ' || v_bonus);
end;
```

Execution: Sql> exec p1(7566);

```
My employee no is: 7566
My employee name is: JONES
My employee salary is: 4675
My employee bonus is: 1403
```

Select ... into clause used in functions:

➢ **Write a PL/SQL stored function which is used to returns max(sal) from emp table?**

Ans:

```
create or replace function f1
return number  is
v_sal number(10);
begin
select max(sal) into v_sal from emp;
return v_sal;
end;
```

Execution

Method 1 Using same table

Sql> select ename, sal, f1 from emp;

Ename	Sal	Max(sal)
Smith	2600	5100
Allen	2100	5100
Ward	1750	5100

Method 2 Using dual table

Sql> select f1 from dual;

5100

Note: In Oracle, we can also use oracle predefined functions into User Defined Functions and also we are allow to "call" these User defined functions by using same table (or) by using dual table.

- **Write a PL/SQL stored function for passing ename as in parameter then return job of the employee from emp table and also if requested employee name is not available in emp table then return a message?**

Ans:

```
create or replace function f1(p_ename in varchar2)
return varchar2
is
 v_job varchar2(10);
begin
 select job into v_job from emp where ename = p_ename;
 return v_job;
exception
 when no_data_found then
  return 'Your employee does not exist';
end;
/
```

Execution:

```
Sql> select f1('SMITH') from dual;
Sql> select f1('abc') from dual;
o/p: Your employee does not exist
```

- **Write a PL/SQL stored function for passing empno is a parameter from emp table, if the employee salary is more than average salary of that employee department then return 1 other wise return 0?**

Ans:

```
create or replace function f1(p_empno number)
return number
is
 v_sal   number(10);
 v_deptno number(10);
 v_avgsal number(10);
begin
 select sal, deptno into v_sal, v_deptno from emp where empno = p_empno;
 select avg(sal) into v_avgsal from emp where deptno = v_deptno;
 if v_sal > v_avgsal then
  return 1;
 else
  return 0;
 end if;
exception
 when no_data_found then
  return - 1;
end;
/
```

Execution:

```
Select f1(7369) from dual;
o/p: 0
Select f1(7566) from dual;
o/p: 1
Select f1(2222) from dual;
o/p: -1
```

➢ **Write a query to display the employees who are getting more sal than the avgsal of their dept by using above user defined function?**

Ans: sql> select ename, sal, f1(empno) from emp where f1(empno) = 1;

Ename	Sal	F1(empno)
Jones	4875	1
Blake	4850	1
Scott	4900	1
King	7000	1

When to use procedure, when to use functions:

Procedure	Functions
1. When application required returning multiple value.	1. When application required returning single value and calling that value in select statement.
2. When application required DML statements.	2. When we are evaluating expressions.
3. When application required returning multiple records.	3. When application required returning multiple records.

➢ **Write a PL/SQL stored function for passing empno, date as parameters then return number of years that employee is working based on passed date from emp table?**

Ans:

```
create or replace function f1(p_empno number, p_date date)
return number
is
 x number(10);
begin
 select months_between(p_date, hiredate) / 12 into x from emp where empno = p_empno;
 return round(x);
end;
/
```

Execution:

Method 1: Using Dual table

```
Sql> select f1(7902, sysdate) from dual;
        34
```

Method 2: Using Emp table

```
Sql> select empno, ename, hiredate, f1(empno,sysdate) || ' ' || 'years' "EXPR" from emp where empno=7566;
```

Output:

EMPNO	ENAME	HIREDATE	EXPR
7566	JONES	02-APR-81	34 years

Note: Prior to oracle 11g, we are not allowed to use **"named notations"**, **"mixed notations"** in function executions where as in oracle 11g we are allowed to use named, mixed notations when a subprogram is executed using select statement.

IN ORACLE 11G:

```
select empno, ename, hiredate,
f1(p_empno => empno, p_date => sysdate) || ' ' || 'years' "EXPR" from emp where empno = 7566;
```

OUT Mode: If you want to return more than 1 value from function then we are using OUT parameter, here also OUT parameter internally behaves like an uninitialized variable, here also explicitly we must specified OUT keyword.

Note: When a function having OUT parameter then those functions are not allow to execute in SELECT statement.

- **Write a PL/SQL stored functions for passing deptno as IN parameter and return dname, loc by using OUT parameters from emp table?**

Ans:

```
create or replace function f1(p_deptno in number, p_dname out varchar2, p_loc out varchar2)
return varchar2
is
begin
 select dname, loc into p_dname, p_loc from emp where deptno = p_deptno;
 return p_dname;
end;
```

Execution: (Using Bind Variable)

```
Sql> variable a varchar2(10);
Sql> variable b varchar2(10);
Sql> variable c varchar2(10);
Sql> begin
     :a := f1(10, :b, :c);
     end;
     /
Sql> print b  c;

       B
----------
ACCOUNTING
       C
----------
  NEW YORK
```

In oracle, we can also develops our own aggregate functions same like a predefined aggregate functions. When these users defined aggregate function returns multiple values then we are using "CURSOR" within user defined functions. These functions also used in "group by" clause same like a predefined aggregate functions.

- **Write a PL/SQL store function for passing empno,dname as in parameter by using emp,dept table and also by using explicit cursor retrieve deptno from dept table based on passed dname and also modify passed employee deptno by using dname,deptno and also return 1 if successfully modify the deptno?**

Ans:

```
create or replace function f1(p_empno number, p_dname varchar2)
 return number
is
 v_deptno number(10);
 cursor c1 is select deptno from dept where dname = p_dname;
begin
 open c1;
 fetch c1 into v_deptno;
 close c1;
 update emp set deptno = v_deptno where empno = p_empno;
 return 1;
end;
/
```

Execution: (By using anonymous block)

```
declare
x number(10);
begin
x := f1(7902, 'sales');
dbms_output.put_line(x);
end;
/
Sql> select * from dept;
Sql> select * from emp;
```

Oracle 10g introduced "WM_CONCAT()" this function accepts column name as a parameter and display those column value horizontally by using "," (comma) separated values.

Example: sql> select wm_concat(ename) from emp;
Output: SMITH,ALLEN,WARD,JONES,MARTIN,BLAKE,CLARK,SCOTT,KING,TURNER,ADAMS,JAMES,FORD

Note: We can also use "WM_CONCAT" function in place of predefined aggregate function within group by clause queries. In this case this function returns multiple values in each group.

Example: sql> select deptno, count(*) from emp group by deptno;

Deptno	Count(*)
10	3
20	5
30	6

Example: sql> select deptno, wm_concat(ename) from emp group by deptno;

Deptno	WM_CONCAT(ENAME)
10	CLARK,MILLER,KING
20	SMITH,FORD,ADAMS,SCOTT,JONES
30	ALLEN,JAMES,TURNER,BLAKE,MARTIN,WARD

Note: In Oracle, all store functions information stored under "user_procedures", "user_source" data dictionaries. If we want to view code of the functions then we are using "user_source" data dictionary.

Example:

```
sql> desc user_source;
sql> select text from user_source where ename= 'F1';
```

In oracle we can drop a function by using: **[drop function functionname;]**

Package is a database object which encapsulates procedures, functions, cursors, global variables, constants, types into single unit.

Generally, packages are used to improves performance of the application because whenever we are calling a packages subprogram first time then automatically total packages loaded into memory area, whenever we are calling sub sequence subprogram then oracle server call those subprograms directly from memory area not from disk.

This process automatically reduces disk I/O (input/output) operations. That's why packages also improve performance of the applications.

In oracle every package having 2 parts these are...

1) Package Specification
2) Package Body

In Oracle, by default package specification objects are public, where as package body objects are private.
In package specification we are declaring variables, constants, cursors, types, procedures, functions. Whereas in package body we are implementing procedures, functions.

Package Specification:

Syntax:

```
create or replace package packagename
is/as
→ global variables declarations, constant declarations;
→ cursors declaration;
→ types declarations;
→ procedure declarations;
→ function declaration;
end;
```

Package Body:

Syntax:

```
create or replace package body packagename
is/as
→ Procedure implementation;
→ Function implementation;
end;
```

Calling Packaged Subprograms:

1. **Calling Packaged Procedures:**

   Method 1:

   Syntax: Sql> exec packagename.procedurename(actual parameters);

   Method 2: (using anonymous block)

   Syntax:

```
Sql> begin
packagename. procedurename(actual parameters);
end;
```

2. **Calling Package Functions:**
   Method 1: (Using select statement)
   Sql> select packagename. Functionname (actual parameters) from dual;

   Method 2: (using anonymous block)
   Syntax:
```
Sql> begin
varname:=packagename. Functioname (actual parameter);
end;
/
```

Example:
```
sql> create or replace package pj1;
is
procedure p1;
procedure p2;
end;
/
```
This is "Package Specification".

```
Sql> create or replace package body pj1
is
procedure p1
begin
dbms_output.put_line('First Procedure');
end p1;
procedure p2
begin
dbms_output.put_line('Second Procedure');
end p2;
end;
/
```
This is "Package Body".

Execution:
```
Sql> exec pj1.p1;
First Procedure

Sql> exec pj1.p2;
Second Procedure
```

Note: Generally in oracle procedures except parameters and also procedure can be nested and also we can call procedure directly but in oracle packages does not accepts parameter and also user defined packages cannot be nested and also we cannot call packages directly.

Example:

```
sql> create or replace package pj2
     is
     g number(10):=1000;
     procedure p1;
     function f1(a number) return number;
     end;
Sql> create or replace package body pj2
     is
     procedure p1
     is
     x number(10);
     begin
     x:=g/2;
     dbms_output.put_line(x);
     end p1;
     function f1(a number) return number
     is
     begin
     return a*g;
     end f1;
     end;
```

Execution:

```
Sql> exec pj2.p1;
500
Sql> select pj2.f1(4) from dual;
4000
```

Serially_reusable pragma:

In Oracle, if you want to maintain state of the global variable or state of the globalised cursor then we must use "serially_reusable" pragma in packages.

Syntax: pragma serially_resuable;

Example: (State of the Global Variable)

```
sql> create or replace package pj3
is
g number(10):=5;
pragma serially_reusable;
end;
/
Sql> begin
pj3.g:=90;
end;
/
Sql> begin
dbms_output.put_line(pj3.g);
end;
/
```

Output: 5

Example: (State of the Cursor)

```
sql> create or replace package pj4
     is
     cursor c1 is select * from emp;
     pragma serially_reusable;
     end;
     /
Sql> begin
     open pj4.c1;
     end;
     /
/*Here cursor is not closed*/
Sql> begin
     open pj4.c1;
     end;
     /
PL/SQL procedure successfully completed
/*cursor is reusable in other blocks*/
```

Forward Declaration: ***

Declaring a procedure in package body is called "Forward Declaration". Generally, before we are calling private procedures into public procedures first we must implement private procedures in package body before calling otherwise use a forward declaration in package body.

Example:

```
sql> create or replace package pj6
is
procedure p1;
end;
/
Sql> create or replace package body pj6
is
procedure p2; /*Forward Declaration*/
procedure p1
is
begin
p2;
end p1;
procedure p2
is
begin
dbms_output.put_line('Private proc');
end p2;
end;
/
```

Execution:

```
Sql> exec pj6.p1;
Private proc
```

Overloading Procedures:

Overloading refers to same name can be used for different purpose. In Oracle, we can also implement overloading procedures by using packages. Overloading procedures having same name with different type and different number of parameters.

Example:

```
sql> create or replace package pj5
is
procedure p1(a number, b number);
procedure p1(x number, y number);
end;
/
Sql> create or replace package body pj5
is
procedure p1(a number, b number)
is
c number(10);
begin
c:= a+b;
dbms_output.put_line(c);
end p1;
procedure p1(x number, y number)
is
z number(10);
begin
z:=x-y;
dbms_output.put_line(z);
end p1;
end;
/
```

Output: sql> exec pj5.p1(9,4);
Error: too many declarations of "P1" match this call

Note: In Oracle, whenever overloading procedures having same number of parameters and also having same types then we are allowed to execute those overloading procedures using "Named notations" only.

Solution:

```
Sql> exec (a=>5, b=>4);
13
Sql> exec(x=> 8, y=> 3);
5
```

In PL/SQL we can also create our own user defined data types by using "type" keyword.
Generally, in PL/SQL we are creating user defined data types by using two step processes i.e. first we are creating user defined types from appropriate syntax and then only we are allow to create a variable from that type.

PL/SQL having following user defined types. These are:

1. **PL/SQL record**
2. **Index by table (or) PL/SQL table (or) Associative Array**
3. **Nested Table**
4. **Varray**
5. **Ref Cursor**

1. Index by Table (or) PL/SQL table (or) Associative Array

Index by table is an unbound table which is used to store number of data items in a single unit.
Basically Index by table has two parts. These are...

1. **Value field**
2. **Key field**

Value field stores actual data where as key field stores indexes. These indexes are either integers (or) characters and also these indexes are either positive or negative. Here key field behaves like a primary key that's why key field does not accepts duplicate values.

Generally index by tables is used to improve performance of the application because by default these tables are stored in RAM memory area. That's why those tables are also called as memory tables.
For improve performance of the index by table then oracle provides special data type "**binary_integer**" into the key field.

This is a user defined type so we are creating in two step process i.e. First we are creating type then only we are creating variable of that type.

Syntax:

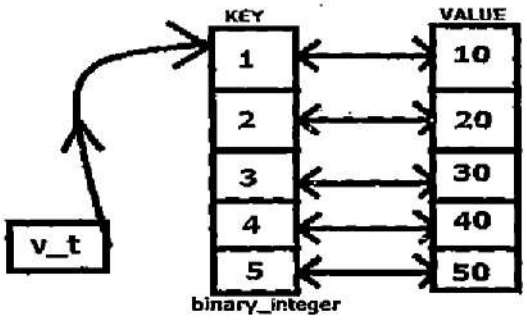
1. Type typename is table of datatype(size)
   index by binary_integer;

2. Variablename typename;

Example:

```
declare
 type t1 is table of number(10)
 index by binary_integer;
 v_t t1;
begin
 v_t(1) := 10;
 v_t(2) := 20;
 v_t(3) := 30;
 v_t(4) := 40;
 v_t(5) := 50;
 dbms_output.put_line(v_t(1));
 dbms_output.put_line(v_t.first);
 dbms_output.put_line(v_t.last);
 dbms_output.put_line(v_t.prior(3));
 dbms_output.put_line(v_t.next(3));
 dbms_output.put_line(v_t.count);
 v_t.delete(1, 3);
 dbms_output.put_line(v_t.count);
 v_t.delete;
 dbms_output.put_line(v_t.count);
end;
/
```

Output: 10 1 5 2 4 5 2 0



➢ **Write a PL/SQL program which is used to transfer all employee names from emp table and storing into index by table and also display content from index by table?**

Ans:

```
declare
 type t1 is table of varchar2(10)
index by binary_integer;
 v_t t1;
 cursor c1 is select ename from emp;
 n number(10) := 1;
begin
 open c1;
 loop
  fetch c1 into v_t(n);
  exit when c1%notfound;
  n := n + 1;
 end loop;
 close c1;
 for i in v_t.first .. v_t.last loop
  dbms_output.put_line(v_t(i));
 end loop;
end;
/
```

When a resource table having large amount of data and also when we are trying to transfer this data into collection by using "Cursors" then those types of applications degrades performance because cursor internally uses record by record process.

To overcome this problem if we want to improve performance of the application Oracle 8i introduced **"Bulk Collect"** clause.

This clause is used in executable section of the PL/SQL block, whenever target is a collection then only we are allow using **"Bulk collect"** clause. When we are using **"Bulk Collect"** clause Oracle server selects column data at a time and stores that data into "Collections".

Syntax: select * bulk collect into collectionvariablename from tablename where condition;

Solution:

```
declare
 type t1 is table of varchar2(10)
 index by binary_integer;
 v_t t1;
begin
 select ename bulk collect into v_t from emp;
 for i in v_t.first .. v_t.last loop
  dbms_output.put_line(v_t(i));
 end loop;
end;
/
```

- **Write a PL/SQL program which stores next 10 dates into index by table and also display content from index by table?**

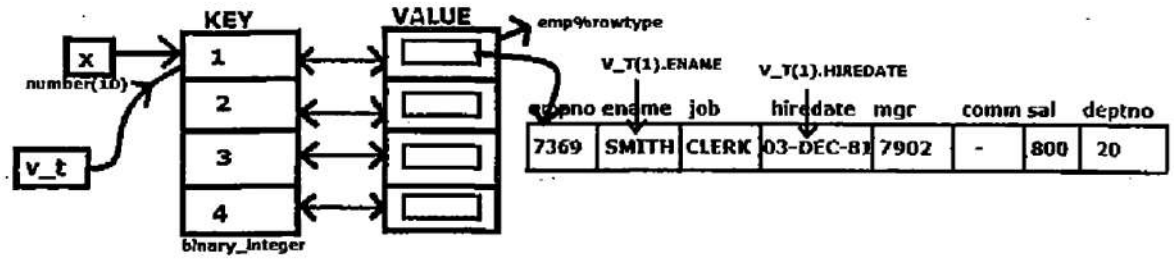
Ans:

```
declare
 type t1 is table of date
 index by binary_integer;
 v_t t1;
begin
 for i in 1 .. 10
  loop
  v_t(i) := sysdate + i;
 end loop;
 for i in v_t.first .. v_t.last
  loop
  dbms_output.put_line(v_t(i));
 end loop;
end;
/
```

- **Write a PL/SQL program which transfer all employees joining dates from emp table into index by table and also display content from index by table?**

Ans:

```
declare
 type t1 is table of date
 index by binary_integer;
 v_t t1;
begin
 select hiredate bulk collect into v_t from emp;
 for i in v_t.first .. v_t.last
  loop
  dbms_output.put_line(v_t(i));
 end loop;
end;
/
```



Example:

```
declare
 type t1 is table of emp%rowtype
 index by binary_integer;
 v_t   t1;
begin
 select * bulk collect into v_t from emp;
 for i in v_t.first .. v_t.last loop
   dbms_output.put_line(v_t(i).ename || ' ' || v_t(i).sal || ' ' || v_t(i).job);
 end loop;
end;
/
```

Return Result Set:

If we want to return large amount of data from Oracle database into client applications then first we must develop database server application and then only execute server application by using client application.

In oracle we are implementing database server application by using following two methods:

1. Method 1 (Using Index by table)
2. Method 2 (Using Ref Cursor)

Method 1 (Using index by table)

In this method we are returning large amount of data from oracle database by specifying index by table as return type within packaged functions.

➢ **Write a PL/SQL database server application by using index by table which is used to return employee details from emp table?**

Ans:

```
Sql> create or replace package pj1
is
 type t1 is table of emp%rowtype
 index by integer_binary;
 function f1 return t1;
end;
/
```

```
Sql> create or replace package body pj1
is
 function f1 return t1
  is
  v_t t1;
 begin
  select * bulk collect into v_t from emp;
  return v_t;
 end f1;
end;
/
```

Execution (By using PL/SQL client):

```
declare
 x pj1.t1;
begin
 x := pj1.f1;
 for i in x.first .. x.last
  loop
  dbms_output.put_line(x(i).ename || ' ' || x(i).sal || ' ' || x(i).deptno);
 end loop;
end;
/
```

Exists Collection Method Used In Index By Tables: ***

Exists collection method is used in "Index by table", "Nested table", "Varray". Exists collection method always return "Boolean" value either true (or) false. If we want to test weather requested data is available (or) not available in collection then we must use exist collection method.

Syntax: collectionvariablename . exists (indexvariablename);

Example:

```
declare
 type t1 is table of varchar2(10)
 index by binary_integer;
 v_t t1;
begin
 select ename bulk collect into v_t from emp;
 v_t.delete(3);
 for i in v_t.first .. v_t.last
  loop
  dbms_output.put_line(v_t(i));
 end loop;
end;
/
```

Output:

```
SMITH
ALLEN
ORA-1403: no data found
```

Note 1: Whenever Index by table or nested table having gaps and also when we are try to display content of those collections then oracle server returns an error "ORA-1403: no data found".
To overcome this problem we must use "exists" collection method.

Note 2: Whenever index by table or nested table having gaps then those collections are also called as <u>sparse</u> collections.

Solution:

```
declare
 type t1 is table of varchar2(10)
 index by binary_integer;
 v_t t1;
begin
 select ename bulk collect into v_t from emp;
 v_t.delete(3);
 for i in v_t.first .. v_t.last
  loop
  if v_t.exists(i) then
   dbms_output.put_line(v_t(i));
  end if;
 end loop;
end;
/
```

Example:

```
declare
 type t1 is table of number(10)
 index by binary_integer;
 v_t t1;
 x  boolean;
begin
 v_t(1) := 10;
 v_t(2) := 20;
 v_t(3) := 30;
 v_t(4) := 40;
 v_t(5) := 50;
 x := v_t.exists(3);
 if x = true then
  dbms_output.put_line('u r requested index 3 exists with having an element' || ' ' ||v_t(3));
 else
  dbms_output.put_line('u r requested index does not exist..');
 end if;
 for i in v_t.first .. v_t.last
  loop
  dbms_output.put_line(v_t(i));
 end loop;
end;
/
```

Oracle 8.0 introduced nested table, varray, these collection also used to store number of data item in a single unit. These collection does not have key-value pairs before we are storing actual data into these collection then we must initialize by using "Constructor". Here Constructor name is also same as type name.

Nested Table:

Nested table is an unbound table which is used to store number of data items in a single unit. Generally in index by table we are not allow to add or remove indexes and also index by table are not store permanently in oracle database.

To overcome this problem oracle 8.0 introduced extension of the index by table called nested table which is used to store permanently in oracle database by using SQL language and also in PL/SQL by using extend collection method we are allow to add indexes and also by using trim collection method, we are allow to remove indexes from nested table.

In Nested table by default indexes are integers and also these indexes are always starting with "1". Nested table having **exists, extend, trim, first, last, prior, next, count, delete(index), delete(one index, another index),** delete collection methods.

Nested table is a user defined type so we are created in 2 steps process i.e. first we are creating type then only we are allow to create variable from that type.

Syntax:

1. type typename is table of datatype(size);
2. variablename typename:= typename(); /*typename() - -> constructor name*/

Before we are storing actual data into nested table then we must use constructor, here coristructor name is same as typename.

Nested table
Example: (Index by table)

```
declare
 type t1 is table of number(10)
 index by binary_integer;
 v_t   t1;
begin
 v_t(500) := 80;
 dbms_output.put_line(v_t(500));
end;
/
```

Output: 80

Whenever we are using Index by table we are not required to resolve the memory explicitly because based on specified key value and also based on value field memory is automatically allocated.

Nested table:

```
declare
 type t1 is table at number(10);
 v_t t1 := t1();
begin
 v_t(500) := 80;
 dbms_output.put_line(v_t(500));
end;
/
```

Error: subscript beyond count.

Solution:

```
declare
 type t1 is table of number(10);
 v_t t1 := t1();
begin
 v_t.extend(500);
 v_t(500) := 80;
 dbms_output.put_line(v_t(500));
end;
/
```

Output: 80

Example:

```
declare
 type t1 is table of number(10);
 v_t t1 := t1();
begin
 v_t.extend(5);
 v_t(1) := 10;
 v_t(2) := 20;
 v_t(3) := 30;
 dbms_output.put_line(v_t(1));
end;
/
```

Output: 10

Note: Without using "Extend" collection method also we can store data directly into nested tables. In this case we must specify actual data within "Constructor" itself.

Example:

```
declare
 type t1 is table of number(10);
 v_t t1 := t1(10, 20, 30, 40, 50);
begin
 dbms_output.put_line(v_t.first);
 dbms_output.put_line(v_t.last);
 for i in v_t.first .. v_t.last
  loop
  dbms_output.put_line(v_t(i));
 end loop;
end;
/
```

Output:

```
1
5
10
20
30
40
50
```

Example:

```
declare
 type t1 is table of number(10);
 v_t  t1;
 v_t2 t1 := t1();
begin
 if v_t1 is null then
   dbms_output.put_line('v_t1 is null');
 else
   dbms_output.put_line('v_t1 is not null');
 end if;
 if v_t2 is null then
   dbms_output.put_line('v_t2 is null');
 else
   dbms_output.put_line('v_t2 is not null');
 end if;
end;
/
```

Output:

```
v_t1 is null
v_t2 is not null
```

Explanation:

1.
```
declare
type t1 is table of number(10);
v_t1 t1;        /* v_t1 -> null*/
```

2.
```
declare
type t1 is table of number(10);
v_t2 t1:= t1();          /*v_t2 -> |-|-|-|-| */
```

3.
```
declare
 type t1 is table of number(10);
 v_t t1 := t1();
begin
 v_t.extend;
 v_t(1) := 10;
 dbms_output.put_line(v_t(1));
end;

/ /* v_t -> |10|-|-|-|-| */
```

Example:

```
declare
 type t1 is table of varchar2(10);
 v_t t1 := t1(10, 20, 30, 40);
begin
 dbms_output.put_line(v_t.first);
 dbms_output.put_line(v_t.last);
 dbms_output.put_line(v_t.prior(3));
 dbms_output.put_line(v_t.next(2));
 dbms_output.put_line(v_t.count);
 v_t. extend;
 v_t. extend(2);
 v_t. extend(3, 2);
 v_t. trim;
 dbms_output.put_line(v_t.count);
 v_t(5) := 50;
 v_t(6) := 60;
 v_t(7) := 70;
 for i in v_t.first .. v_t.last
  loop
  dbms_output.put_line(v_t(i));
 end loop;
 v_t.delete;
 dbms_output.put_line(v_t.count);
end;
/
```

Output:

1	4	2	3	4	9	10	20	30	50	60	70	20	20	0

Difference between Trim, Delete:

Trim →Delete Indexes last indexes onwards
Delete (Index) →Delete any element in any position within Nested table

Example:

```
declare
 type t1 is table of number(10);
 v_t t1 := t1(10, 20, 30, 40, 50);
begin
 dbms_output.put_line(v_t.count);
 v_t.trim;
 dbms_output.put_line(v_t.count);
 v_t.delete(2);
 for i in v_t.first .. v_t.last
  loop
  if v_t.exists(i) then
   dbms_output.put_line(v_t(i));
  end if;
 end loop;
end;
/
```

Output: 5 4 10 30 40

Oracle 8.0 introduced Varray. Varray is a bonded array which is used to store up to 2GB data. This is a user defined type which is used to store number of data items in a single unit. Before we are storing actual data into Varray's also then we must initialize by using Constructor. Here Constructor name is also same as type name.

This is a user defined type so we are creating in two step process i.e. first we are creating type name then only we are allow to create variable from that table.

Syntax:

1. type typename is varray(maxsize) of datatype(size);
2. variablename typename := typename(); /* constructor name */

Note: In varray also always indexes are integers and also these indexes are always starting with 1.

Example:

```
declare
 type t1 is varray(10) of number(10);
 v_t t1 := t1(10, 20, 30, 40);
begin
 dbms_output.put_line(v_t.limit);
 dbms_output.put_line(v_t.count);
 dbms_output.put_line(v_t.first);
 dbms_output.put_line(v_t.last);
 dbms_output.put_line(v_t.prior(3));
 dbms_output.put_line(v_t.next(3));
 v_t.extend;
 v_t(5) := 50;
 v_t.extend(3, 2);
 v_t.trim;
 dbms_output.put_line(v_t.count);
 for i in v_t.first .. v_t.last
  loop
  dbms_output.put_line(v_t(i));
 end loop;
 v_t.delete;
 dbms_output.put_line(v_t.count);
end;
/
```

Note: In VARRAY'S we cannot delete particular elements (or) range of indexes by using "Delete" collection method, because VARRAY is not a sparse i.e. VARRAY does not have any gaps, but we can delete all elements by using "Delete" collection method.

Oracle 8.0 introduced Varray. Varray is a bonded array which is used to store up to 2GB data. This is a user defined type which is used to store number of data items in a single unit. Before we are storing actual data into Varray's also then we must initialize by using Constructor. Here Constructor name is also same as type name.

This is a user defined type so we are creating in two step process i.e. first we are creating type name then only we are allow to create variable from that table.

Syntax:

1. type typename is varray(maxsize) of datatype(size);
2. variablename typename := typename(); /* constructor name */

Note: In varray also always indexes are integers and also these indexes are always starting with 1.

Example:

```
declare
 type t1 is varray(10) of number(10);
 v_t t1 := t1(10, 20, 30, 40);
begin
 dbms_output.put_line(v_t.limit);
 dbms_output.put_line(v_t.count);
 dbms_output.put_line(v_t.first);
 dbms_output.put_line(v_t.last);
 dbms_output.put_line(v_t.prior(3));
 dbms_output.put_line(v_t.next(3));
 v_t.extend;
 v_t(5) := 50;
 v_t.extend(3, 2);
 v_t.trim;
 dbms_output.put_line(v_t.count);
 for i in v_t.first .. v_t.last
  loop
  dbms_output.put_line(v_t(i));
 end loop;
 v_t.delete;
 dbms_output.put_line(v_t.count);
end;
/
```

Note: In VARRAY'S we cannot delete particular elements (or) range of indexes by using "Delete" collection method, because VARRAY is not a sparse i.e. VARRAY does not have any gaps, but we can delete all elements by using "Delete" collection method.

- **Write a PL/SQL program to transfer first 10 employee names from emp table and store it into "Varray" and also display content from Varray?**

Ans:

```
declare
 type t1 is varray(10) of varchar2(10);
 v_t t1 := t1();
begin
 select ename bulk collect into v_t from emp where rownum <= 10;
 for i in v_t.first .. v_t.last
  loop
  dbms_output.put_line(v_t(i));
 end loop;
end;
/
```

Note: We can also use exist collection method in VARRAY i.e. whether data is available or not available in a VARRAY then we are using exist collection method i.e. it is used to test whether requested data is available in VARRAY or not. Whenever requested data is available it returns true where as requested data is not available it returns false.

Example:

```
declare
 type t1 is varray(10) of number(10);
 v_t t1 := t1(10, 20, 30, 40, 50);
 x   boolean;
begin
 x := v_t.exists(2);
 if x = true then
  dbms_output.put_line('u r requested index 2 exists with having an element' || ' ' || v_t(2));
 else
  dbms_output.put_line('u r requested index 2 does not exist');
 end if;
 for i in v_t.first .. v_t.last
  loop
  dbms_output.put_line(v_t(i));
 end loop;
end;
/
```

Output :

```
u r requested index 2 exists with having an element 20
10
20
30
40
50
```

Index By Table	Nested Table	Varray
Index by table is an unbound table having key value process.	This table is an unbound table does not have key value process.	Varray are bounded table which stores up to 2GB data.
We cannot add or remove indexes.	We can add or remove indexes by using Extend, Trim collection methods.	We can add or remove indexes by using Extend, Trim collection methods.
Index by table is not allowing to store permanently in oracle database.	Nested table are permanently store in oracle database by using SQL language.	Varray are permanently store in oracle database by using SQL language.
Here indexes are either integers (or) characters and also these indexes are +ve (or) –ve numbers.	Here indexes are integers only and also by default those indexes are start with 1.	Here indexes are integers only and also by default those indexes are start with 1.
Index by table is a sparse collection.	Nested table is a sparse collection.	Varray is a dense collection i.e. (there is no gaps).
Automatically initialized when declaring index by table.	Initiaiized explicitly by using constructor.	Initialized explicitly by using constructor.
Index by table having exists, first, last, prior, next, count, delete(index), delete(one index, another index) delete collection mthod.	Nested table having exists, extend, trim, first, last, prior, next, count, delete(index), delete(one index, another index) delete collection mthod.	Varray having exists, extend, trim, limit, first, last, prior, next, delete collection mthod.

Whenever we are submitting PL/SQL block into oracle server then all SQL statements are executed within SQL Engine and also all procedural statements are executed within PL/SQL engine. This type of execution methods are also called as **"Context Switching Execution"** methods.

Whenever PL/SQL block having more number of SQL, Procedural statements then these types of "Context Switching Execution" methods degrades performance of the application.

To overcome this problem Oracle 8i introduced Bulk Bind process. Bulk bind process is a special type of PL/SQL code which is used to improve performance of the application because it reduces number of context switching between SQL engine & PL/SQL engine.

Generally in PL/SQL block whenever DML statements is available within a loop that PL/SQL code degrades performance, in place of that code only we are using special type of code by using **forall** statements. This code is also called as bulk bind code which is used to improve performance of the application.

Forall is one of the most important performance enhanced features in PL/SQL we should use it whenever we are executing DML statements inside a loop that's why **"forall"** typically execute multiple DML statements.

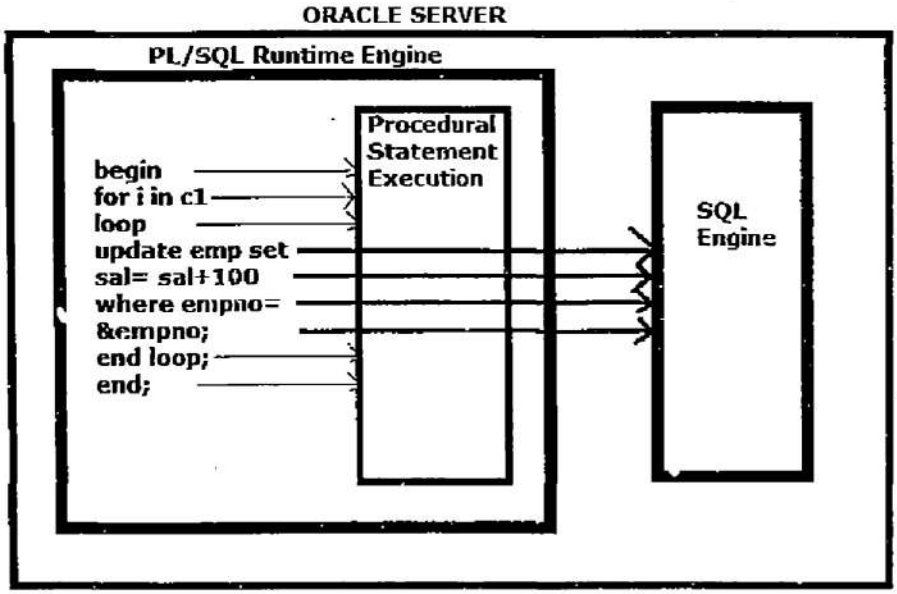
In bulk bind process we are using special type of coding through **"forall"** statements by using following syntax.

Syntax:

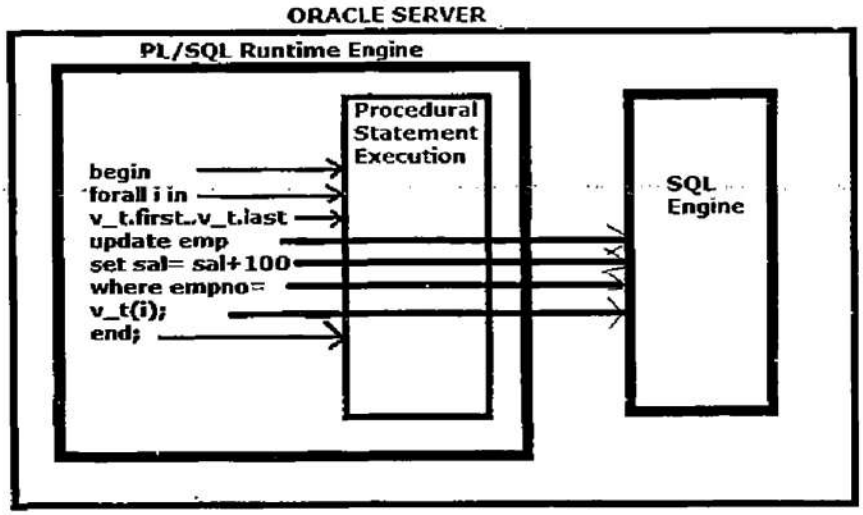
```
forall indexvarname in collectionvarname.first..collectionvarname.last
DML statement where columnname= collectionvarname(indexvarname);
```

In bulk bind process we are using bulk updates, bulk delete, bulk inserts by using for all statements.

<u>Without Using Bulk Bind (Performance Penalty for Many Context Switching)</u>



Using Bulk Bind (Improves performance because of much less overhead)



In Bulk Bind process, we are processing all data in a collection at a time by using SQL engine through "forall" statements. Before we are using this process we must "fetch" the data from resource into "Collections" by using "Bulk Collect" clause. That's why "Bulk Bind" is a two step process.

Step 1: Fetching data from resource into collection by using "Bulk Collect" clause.
Step 2: Process all data in a collection at a time by using SQL engine through "forall" statements.

Step 1: Fetching Data from resource into Collection by using Bulk Collect clause.

Before we are using "forall" statements we must fetching data from resource into collection by using "Bulk Collect" clause.

In Oracle, Bulk Collect clause is used in following 3 ways. These are...

1) **SELECT....Into Clause**
2) **Cursor..... Fetch ..... Statement**
3) **DML ..... Returning..... Into Clause**

1. **Bulk Collect clause used in SELECT..... into clause:**

Syntax: select * bulk collect into collectionvariablename from tablename where condition;

Example:

```
declare
 type t1 is table of emp%rowtype
 index by binary_integer;
 v_t t1;
begin
 select * bulk collect into v_t from emp;
 for i in v_t.first .. v_t.last
  loop
  dbms_output.put_line(v_t(i).ename);
 end loop;
end;
```

2. Bulk Collect clause used in cursor.....fetch statement:

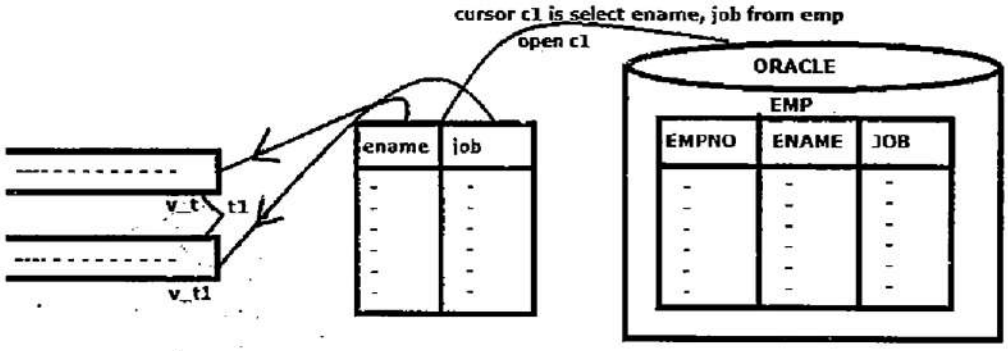
We can also use bulk collect clause in cursor fetch statement by using following syntax.

Syntax: fetch cursorname bulk collect into collectionvarname [limit anynumber];

Generally in PL/SQL collections are executed within PGA memory area. PGA memory area is available in server process. PGA memory area performance is very high compare to SGA memory area but PGA memory area size is very less compare to SGA memory area.

Whenever collection having more data than the PGA memory area size then PL/SQL blocks returns an error. To overcome this problem oracle provided **"limit"** clause within cursor fetch bulk collect into clause. Whenever we are using limit clause set of data at a time process based on the specified number within limit clause.

"Limit" clause is an optional clause which is used to reduce number of values within PGA memory area because in PL/SQL collections are executed within PGA memory area.



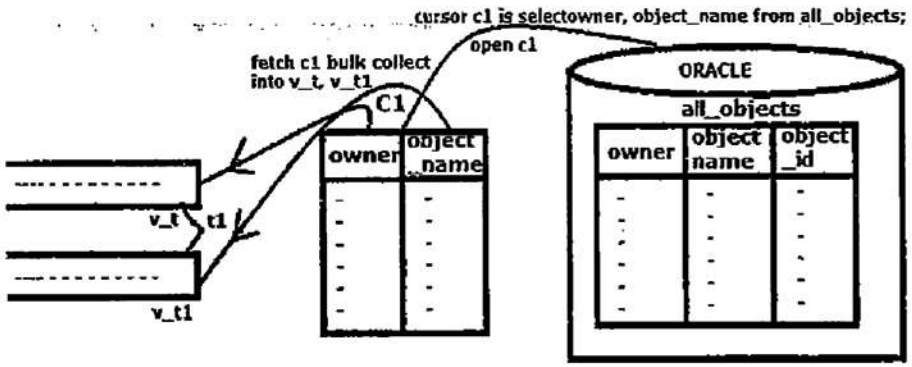
Example:

```
declare
 type t1 is table of varchar2(10)
 index by binary_integer;
 v_t  t1;
 v_t1 t1;
 cursor c1 is select ename, job from emp;
begin
 open c1;
 fetch c1 bulk collect into v_t, v_t1;
 close c1;
 for i in v_t.first .. v_t.last
  loop
  dbms_output.put_line(v_t(i) || ' ' || v_t1(i));
 end loop;
end;
/
```

Calculate Elapsed time in PL/SQL blocks:

In PL/SQL if we want to calculate elapsed time of a PL/SQL code then we are using "get_time" function from "dbms_utility" package. This function always returns number data type.

Syntax: Variablename := dbms_utility.get_time;



Example:

```
declare
 type t1 is table of all_objects.object_name%type
 index by binary_integer;
 v_t  t1;
 v_t1 t1;
 cursor c1 is select owner, object_name from all_objects;
 z1 number(10);
 z2 number(10);
begin
 z1 := dbms_utility.get_time;
 for i in c1
  loop
  null;
 end loop;
 z2 := dbms_utility.get_time;
 dbms_output.put_line('Elapsed time for normal fetch' ||' '|| (z2 - z1) ||' '||'hsecs');

 z1 := dbms_utility.get_time;
 open c1;
 fetch c1 bulk collect into v_t, v_t1;
 close c1;
 z2 := dbms_utility.get_time;
 dbms_output.put_line('Elapsed time for bulk fetch' ||' '|| (z2 - z1) ||' '||'hsecs');
end;
/
```

Output:

```
Elapsed time for normal fetch 101 hsecs
Elapsed time for bulk fetch 64 hsecs.
```

3. Bulk Collect clause used in DML..... returning.... into clauses:

SQL Language:

Example:

```
sql> variable a varchar2(10),
Sql> update emp set sal= sal+100 where ename= 'KING' returning job into :a;
1 row updated
Sql> print a;
```

Output:

```
A
---------
PRESIDENT
```

Returning into clauses are used in DML statements only. These clauses returns transactional data from DML statements and store it into variables. In SQL language these clauses returns only one value at a time or multiple values from a single record at a time.

To overcome this problem if we want to retrieve multiple records from DML transactions then Oracle 8i onwards we can also use **Bulk Collect** clause in DML returning into clauses within PL/SQL.

Example:

```
declare
 type t1 is table of number(10)
 index by binary_integer;
 v_t t1;
begin
 update emp set sal = sal + 100 where job = 'CLERK'
 returning sal bulk collect into v_t;
 dbms_output.put_line('Affected number of clerks' || ' ' || sql%rowcount);
 for i in v_t.first .. v_t.last
  loop
  dbms_output.put_line(v_t(i));
 end loop;
end;
/
```

Output:

```
Affected number of clerks 4
2700
2100
1950
2300
```

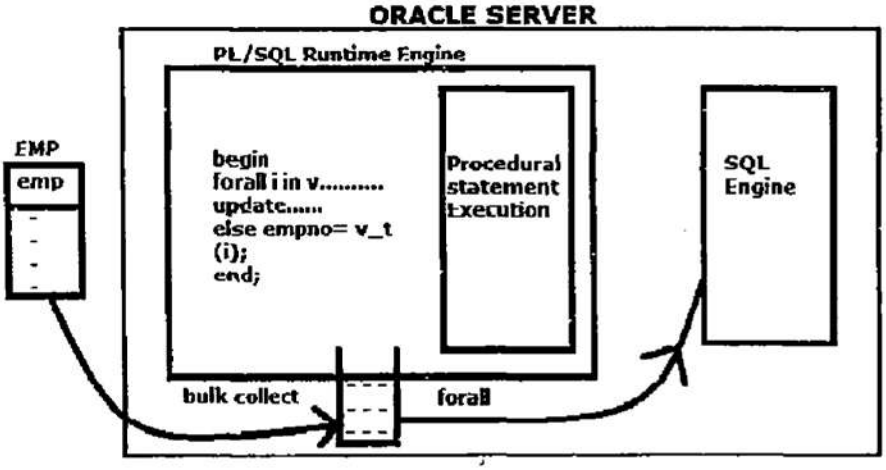
STEP 2: Process all data in a Collection at a time by using SQL Engine (or) process table data by using collection data at a time through "forall" statements (Actual Bulk Bind)

Once data is available in collection then they are processing all data in acollection at a time (or) processing table data by using this collection data by using "forall" statements is called bulk bind process. This process automatically improves performance because it reducesnumber of context switches between SQL language, PL/SQL engine.

Syntax:
forall IndexVarname in CollectionVariableName.first .. CollectionVariable.last
DML statement where columnname = CollectionVariableName(index variable name);

Example:

```
declare
 type t1 is varray(10) of number(10);
 v_t t1 := t1(10, 20, 30, 40, 50);
begin
 forall i in v_t.first .. v_t.last
   update emp set sal = sal + 100 where deptno = v_t(i);
end;
/
```



- **Write a PL/SQL bulk bind program which is used to retrieve all employee numbers from emp table and store it into index by table using bulk collect clause and also increment salary of all employees in emp table by using this collection data through forall statements (or) convert following PL/SQL code into bulk bind code?**

PL/SQL code

```
sql> begin
for i in c1
loop
update emp set sal=sal+100 where empno=i.empno;
end loop;
end;
/
```

Ans:

```
declare
 type t1 is table of number(10)
 index by binary_integer;
 v_t t1;
begin
 select empno bulk collect into v_t from emp;
 forall i in v_t.first .. v_t.last
   update emp set sal = sal + 100 where empno = v_t(i);
end;
/
```

Example:

```
declare
 type t1 is table of number(10)
 index by binary_integer;
 v_t t1;
begin
 select ename bulk collect into v_t from emp;
 v_t delete(3);
 forall i in v_t.first .. v_t.last
   update emp set sal = sal + 100 where empno = v_t(i);
end;
/
```

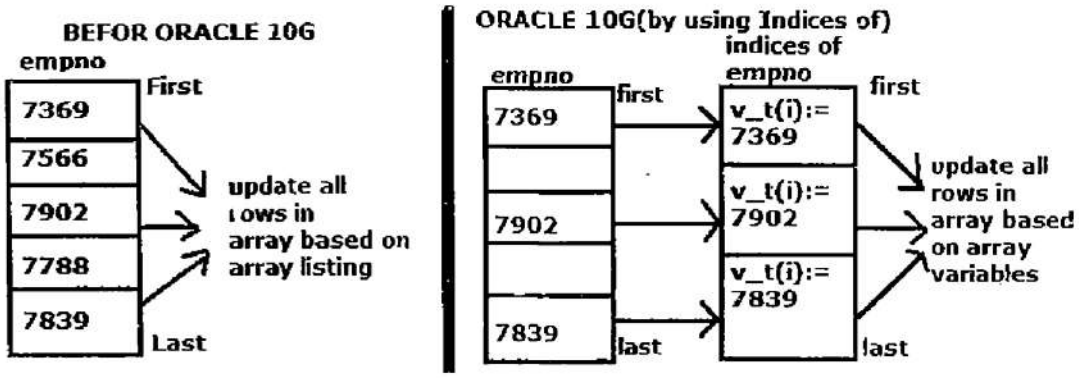
Error: element at index(3) does not exist.

Whenever index by table (or) nested table having "Gaps" then we are not allowed to use "Bulk Bind" process (forall statements). To overcome this problem then we are using "Varray" in Bulk Bind process. Because "Varray" does not have any gaps, but Varrays stores upto 2GB data.

To overcome all these problems Oracle 10G introduced "Indices of" clause within "Bulk Bind" process. This clause is used in "forall" statements. Whenever we are using "indices of" clause then we can also use sparse collections (index by table, nested table) in bulk bind process.

Syntax:

```
forall indexvarname in indices of collectionvarname
DML statement where columnname = collectionvarname(index variable name);
```



Solution: (indices of →10G)

```
declare
 type t1 is table of number(10) index by binary_integer;
 v_t t1;
begin
 select empno bulk collect into v_t from emp;
 v_t. delete(3);
 forall i in indices of v_t
   update emp set sal = sal + 100 where empno = v_t(i);
end;
/
```

Bulk deletes:

```
declare
 type t1 is varray(10) of number(10);
 v_t t1 := (10, 20, 30, 40, 50);
begin
 forall i in v_t.first .. v_t.last
   delete from emp where deptno = v_t(i);
end;
/
```

Bulk Inserts:

- **Write PL/SQL program which transfer all employee names from emp table and store it into another table by using bulk bind process?**

Ans:

```
sql> create table target(name varchar2(10));

sql>declare
 type t1 is table of varchar2(10)
 index by binary_integer;
 v_t t1;
begin
 select ename bulk collect into v_t from emp;
 v_t(3) := 'England';
 v_t(4) := 'Newzeland';
 forall i in v_t.first .. v_t.last
   insert into target values (v_t(i));
end;
/
```

Output:

```
Select * from target;
Select * from emp;
```

Forall is one of the most important performance inhance feature in PL/SQL. Forall statements executes multiple DML statements whenever an exception occurs in one of those DML statements then the default behavior is :

1) **That statement is rolled back and "forall" execution stops.**
2) **All (previous) successful statements cannot be rolled back.**

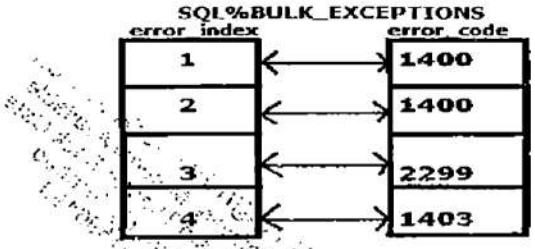
Generally when exception occur a bulk bind process is called bulk exception. If we want forall statements processing to continue even if an error occur in one of the statements then we must add "save exceptions" clause. This clause is used in forall statements.

Syntax:

```
forall indexvarname in collectionvarname.first .. collectionvarname.last save exception
DML statement where columnname = collectionvarname(indexvarname);
```

Whenever we are adding save exception clause then oracle server internally automatically creates an index by table , this index by table name is **"SQL%bulk_exceptions"**. This index by table is also called as PSEUDO collection because this index by table having only one collection method **"count"**. This index by table having 2 fields these are...

1) **Error_Index**
2) **Error_Code**



Example:

```
Sql> create table target(sno number(10) not null);

Sql> declare
  type t1 is table of number(10);
  v_t t1 := t1(10, 20, 30, 40, 50);
begin
  v_t(3) := null;
  v_t(4) := null;
  forall i in v_t.first .. v_t.last
    insert into target values (v_t(i));
end;
/
```

Error: ORA-01400:cannot insert null into sno.

Whenever exception occur in bulk bind process oracle server stops forall execution,these exception also called as bulk exceptin. To over come this problem for handling exception in bulk bind process then oracle server provided following 2 steps process. These steps are...

Step1: Add save exceptions clause.
Step2: Use count collection method in sql%bulk_exceptions PSEUDO collection.

Step1: Add save exceptions clause:
For handling bulk exceptions then first we must add save exception clause within forall statements. Save exceptions clause else to oracle to save exceptions information in index by table and then continiue forall statements executions.

Step2: Use count collection method in sql%bulk_exceptions PSEUDO collection:

When we are adding saving exception clause then oracle server internally automatically create SQL%bulk_exceptions PSEUDO collections. This index by table only automatically store exceptions information that's why through the count collection method, we must count exception information by using following syntax within default handler under exception section of the PL/SQL block.

Syntax: variablename := SQL%bulk_exceptions.count;

Solution:

```
declare
 type t1 is table of number(10);
 v_t t1 := t1(10, 20, 30, 40, 50);
 z  number(10);
begin
 v_t(3) := null;
 v_t(4) := null;
 forall i in v_t.first .. v_t.last save exceptions
  insert into target values (v_t(i));
exception
 when others then
  z := sql%bulk_exceptions.count;
  dbms_output.put_line(z);
end;
/
```

Output: 2

```
Select * from target;
SNO
---
10
20
30
```

Whenever we are executing above program more numbers of times then each and every time data is transferring to target table. When we are executing that program more number of time if you want to store target table data one time then we must erase previous data either by using **"delete"** (or) **"truncate"** command, but here we are using **"truncate"** command because **"truncate"** gives more performance than the **"delete"** command, but when we are try to use **"truncate"** command in PL/SQL block then oracle server return an error because we are not allow to use DDL command in **PL/SQL** block.

To overcome this problem then we must use Dynamic SQL constructs **"execute immediate"** clause within DDL statements by using following syntax.

Syntax: execute immediate 'SQL statements';

Example:

```
declare
 type t1 is table of number(10);
 v_t t1 := t1(10, 20, 30, 40, 50);
 z  number(10);
begin
 v_t(3) := null;
 v_t(4) := null;
 execute immediate 'truncate table target';
 forall i in v_t.first .. v_t.last save exceptions
  insert into target values (v_t(i));
exception
 when others then
  z := sql%bulk_exceptions.count;
  dbms_output.put_line(z);
end;
/
```

SQL%bulk_exceptions index by table having 2 fields:

1) **ERROR_INDEX**
2) **ERROR_CODE**

ERROR_INDEX : error_index fields stores index number of the collection where the exceptions are occur within bulk bind process.

Syntax: sql%bulk_exceptions(index variable name).error_index

ERROR_CODE: This field stores oracle error number where the exception is occur in bulk bind process.

Syntax: sql%bulk_exceptions(index variable name).error_code

Example:

```
declare
 type t1 is table of number(10);
 v_t  t1 := t1(10, 20, 30, 40, 50);
 z  number(10);
begin
 v_t(3) := null;
 v_t(4) := null;
 execute immediate 'truncate table target';
 forall i in v_t.first .. v_t.last save exceptions
  insert into target values (v_t(i));
exception
 when others then
  z := sql%bulk_exceptions.count;
  dbms_output.put_line(z);
  for j in 1 .. z
   loop
   dbms_output.put_line(Sql%bulk_exceptions(i).error_index || ' ' ||
   Sql%bulk_exceptions(j).error_code);
  end loop;
end;
```

SQL%bulk_rowcount:

This attribute also returns "number" data type. If you want to count affected number of rows in each process after Bulk bind(forall statement) then we must use "sql%bulk_rowcount" attribute.

Syntax: Sql%bulk_rowcount (index variable name);

Example:

```
declare
 type t1 is varray(10) of number(10);
 v_t t1 := t1(10, 20, 30, 40, 50);
begin
 forall i in v_t.first .. v_t.last
  update emp set sal = sal + 100 where deptno = v_t(i);
 for i in v_t.first .. v_t.last
  loop
  dbms_output.put_line('affected number of rows in deptno' || ' ' ||
              v_t(i) || ' is: ' || sql%bulk_rowcount(i));
 end loop;
end;
/
```

Output:

```
affected number of rows in deptno 10 is: 3
affected number of rows in deptno 20 is: 5
affected number of rows in deptno 30 is: 6
affected number of rows in deptno 40 is: 0
affected number of rows in deptno 50 is: 0
```

Oracle 7.2 introduced REF Cursors. REF Cursors is an user defined types which is used to process multiple records and also this is an record by record process.

Generally in static cursors we can execute only one SELECT statement for a single active set area, where as in Ref cursor oracle server execute multiple number of SELECT statement dynamically for a single active set area at runtime that's why this cursor is also called as Dynamic Cursor.

Generally, we are not allowed to pass static cursor as parameter to sub programs where as we are allow to pass REF Cursor as parameter to the sub programs because basically REF Cursor is an user defined type.

Generally we are allow to pass all user defined types as parameter to the sub programs.
Generally, static cursors does not return multiple records into client applications, whereas Ref cursors returns multiple records from database server into client applications.

Generally Ref cursors is an user defined type so we are creating in 2 step process i.e. first we are creating type then only we are creating variable from that type. That's why Ref cursors is also called as Cursor Variable.

All database systems having two types of Ref cursors.

1) **Strong Ref cursors**
2) **Weak Ref cursors**

Strong Ref cursors is a Ref cursors which having return type. This return type must be record type data type. Whereas Weak Ref cursors is a Ref cursor which does not have a return type.

Syntax:

1) type typename is ref cursor return recordtypedatatype;
   variablename typename; /* variablename ---> strong ref cursor variable */

2) type typename is ref cursor;
   variablename typename; /*variablename ----> weak ref cursor variable */

In Ref Cursors we can specify SELECT statement by using "open.....for" clause. This clause is used in executable section of the PL/SQL block.

Syntax: open refcursorname <u>for</u> select * from tablename where condition;

Weak ref cursor: Ref cursor which does not have any return type is called weak ref cursor.

Syntax:

```
type typename is ref cursor;
variablename typename;
```

Example:

```
declare
 type t1 is ref cursor;
 v_t t1;
 i  emp%rowtype;
begin
 open v_t for select * from emp where sal > 2000;
 loop
  fetch v_t into i;
  exit when v_t%notfound;
  dbms_output.put_line(i.ename || ' ' || i.sal);
 end loop;
 close v_t;
end;
```

- **Write a PL/SQL program by using ref cursor whenever user entered deptno "10" then display 10th department details from emp table whenever user entered deptno 20 then display 20th dept details from dept table?**

Ans:

```
declare
 type t1 is ref cursor;
 v_t    t1;
 i      emp%rowtype;
 j      dept%rowtype;
 v_deptno number(10) := &deptno;
begin
 if v_deptno = 10 then
  open v_t for select * from emp where deptno = v_deptno;
  loop
   fetch v_t into i;
   exit when v_t%notfound;
   dbms_output.put_line(i.ename || ' ' || i.sal || ' ' || i.deptno);
  end loop;
 elsif v_deptno = 20 then
  open v_t for select * from dept where v_deptno = 20;
  loop
   fetch v_t into j;
   exit when v_t%notfound;
   dbms_output.put_line(j.deptno || ' ' || j.dname || ' ' || j.loc);
  end loop;
 end if;
 close v_t;
end;
/
```

Output:

```
Enter value for deptno= 10
CLARK  3000  10
KING   7600  10
MILLER 5300  10
Sql> /
Enter value for deptno= 20
20   RESEARCH    DALLAS
```

Oracle 9i introduced "sys_refcursor" predefined type in place of "Weak Refcursor".

Syntax: Refcursorvariablename sys_refcursor;

Example:

```
declare
 v_t sys_refcursor;
 i emp%rowtype;
begin
 open v_t for select * from emp;
 loop
  fetch v_t into i;
  exit when v_t%notfound;
  dbms_output.put_line(i.ename || ' ' || i.sal);
 end loop;
 close v_t;
end;
/
```

Passing sys_refcursor as parameter to the Stored Procedure:

Passing sys_refcursor as IN parameter to the Stored Procedure:

- **Write a PL/SQL stored procedure for passing sys_refcursor as IN parameter which display all employee names and their salaries from emp table?**

Ans:

```
create or replace procedure p1(v_t in sys_refcursor)
is
 i emp%rowtype;
begin
 loop
  fetch v_t into i;
  exit when v_t%notfound;
  dbms_output.put_line(i.ename || ' ' || i.sal);
 end loop;
end;
/
```

Note: In oracle whenever we are passing refcursor as IN parameter to the stored procedure then we are not allowed to use "open.....for" statement. Because by default IN parameter use to pass the value into procedure where as "open.....for" statements return values from the procedures.

Execution:

```
declare
 v_t sys_refcursor;
 open v_t for select * from emp;
 p1 (v_t);
 close v_t;
end;
/
```

- **Write a PL/SQL stored procedure for passing refcursor as OUT parameter which returns employee details from emp table?**

Ans:

```
create or replace procedure p1(v_t out sys_refcursor);
is
begin
open v_t  for select * from emp;
end;
/
```

Execution: (using anonymous block)

```
declare
 v_t sys_refcursor;
 i  emp%rowtype;
begin
 p1(v_t);
 loop
  fetch v_t into i;
  exit when v_t%notfound;
  dbms_output.put_line(i.ename || ' ' || i.sal);
 end loop;
 close v_t;
end;
```

In oracle if you want to return multiple records from oracle database into client applications then we are using following 2 methods. These are ...

Method 1: Using refcursor as out parameter in store procedure.
Method 2: Using refcursor as return type from stored functions.

Method 2: Using refcursor as return type from stored functions.

- **Write a PL/SQL stored function by using sys_refcursor as return type which returns multiple records from emp table?**

Ans:

```
create or replace function f1
return sys_refcursor
is
 v_t sys_refcursor;
begin
 open v_t for select * from emp;
 return v_t;
end;
/
```

Execution: (By using select statements)

```
Sql> select f1 from dual;
```

Example:

```
Sql> create or replace package pj1
     is
      type t1 is ref cursor;
      procedure p1(v_t out t1);
     end;

Sql> create or replace package body pj1 is
     procedure p1(v_t out t1) is
     begin
      open v_t for
       select * from emp;
     end p1;
    end;
    /
```

Execution (By using bind variable):

```
Sql> variable a refcursor;
Sql> exec pj1.p1(:a);
Sql> print a;
```

Note: In oracle we are not allow to declare refcursor variable in package but we are allow to pass fefcursor as parameter to the packaged subprograms.

```
Sql> create or replace package pj2
     is
      type t1 is ref cursor;
      v_t t1;
     end;

Warning: Package created with compilation errors.
Error: Cursor Variables cannot be declared as part of a package
```

Local Sub Programs are name PL/SQL blocks which is used to solve particular task. These sub programs does not have "create or replace" keywords and also local sub programs does not store permanently into database. Oracle has 2 types of Local Sub Programs.

1) **Local Procedures**
2) **Local Functions**

In oracle Local Sub Programs are defined in either Stored Procedures or in Anonymous blocks.
Local Sub Programs must be declaring in "Bottom of the Declare Session" within stored procedure or in anonymous block and also "**CALL**" these local subprograms in immediate "Executable Section".

Syntax:

```
declare
→variable declarations, constant declarations;
→types declarations;
→cursor declarations;
→procedure procedurename(formal parameters)
is/as
begin
--------
--------
[exception]
----------
----------
end [procedurename];
→function functionname(formal parameters)
return datatype
is/as
begin
--------
--------
return expression;
end [function name];
begin
procedurename(actual parameters);
varname:= functionname(actual parameter);
end;
```

Example:

```
declare
 procedure p1 is
 begin
  dbms_output.put_line('Local Procedure');
 end p1;
begin
 p1;
end;
/
```

Output: Local Procedure

Example:

```
create or replace procedure p2
is
 procedure p1
  is
 begin
  dbms_output.put_line('Local Procedure');
 end p1;
begin
 p1;
end;
/
```

Execution:

```
Sql> exec p2;
Local Procedure
```

Example: (Reusing Local Procedures by "calling" in anonymous blocks)

```
declare
 type t1 is ref cursor return emp%rowtype;
 v_t t1;
 procedure p1(p_t in t1)
is
  i  emp%rowtype;
 begin
  loop
   fetch p_t into i;
   exit when p_t%notfound;
   dbms_output.put_line(i.ename || ' ' || i.sal);
  end loop;
 end p1;
begin
 open v_t for select * from emp where rownum <= 10;
 p1(v_t);
 close v_t;
 open v_t for select * from emp where ename like 'M%';
 p1(v_t);
 close v_t;
 open v_t for select * from emp where job = 'CLERK';
 p1(v_t);
 close v_t;
end;
/
```

In all relational databases store procedure are used to improves performance of the application because store procedure internally having one time compilation by default one time compilation program units improve performance of the application.

In all relational databases local procedures are used for code resuabality i.e. regularly used task are implemented within local procedure and then call those local procedure repeatatly whenever necessary.

Example:

```
declare
 type t1 is table of emp%rowtype
 index by binary_integer;
 procedure p1(p_t in t1) is
  i emp%rowtype;
 begin
  for i in p_t.first .. p_t.last loop
   dbms_output.put_line(p_t(i).ename);
  end loop;
 end p1;
 function f1 return t1 is
  v_t t1;
 begin
  select * bulk collect into v_t from emp;
  return v_t;
 end f1;
begin
 p1(f1);
end;
/
```

Strong refcursor is a refcursor which having return type, this return type must be a record type datatype.

Syntax:

```
Type typename is ref cursor return recordtype datatype;
Variablename typename;
```

Example:

```
declare
 type t1 is ref cursor return emp%rowtype;
 v_t t1;
begin
 open v_t for select * from emp where deptno = 10;
 dbms_output.put_line('Welcome');
 close v_t;
end;
/
```

Output: Welecome

Example:

```
declare
 type t1 is ref cursor return emp%rowtype;
 v_t t1;
begin
 open v_t for select * from dept where deptno = 10;
 dbms_output.put_line('Welcome');
 close v_t;
end;
```

Error: expression is of wrong type

This is an user defined type which is used to represent number of different data types into single unit it is also same as structures in 'C' language. This is an User defined type so we are creating a two step process i.e. first we are creating type then only we care creating a variable from that type.

Syntax:

1. type typename is record(attribute1 datatype(size), attribute2 datatype(size),.....);
2. variablename typename;

Example:

```
declare
 type t1 is record(
  a1 number(10),
  a2 varchar2(10),
  a3 number(10));
 v_t t1;
begin
 v_t.a1 := 101;
 v_t.a2 := 'shailendra';
 v_t.a3 := 3000;
 dbms_output.put_line(v_t.a1 || ' ' || v_t.a2 || ' ' || v_t.a3);
end;
/
```

Output: 101 shailendra 3000

PL/SQL record used in package:

```
Sql> create or replace package pj1
is
 type t1 is record(
  a1 number(10),
  a2 varchar2(10),
  a3 number(10));
 procedure p1;
end;
/

Sql> create or replace package body pj1
is
 procedure p1 is
  v_t t1;
 begin
  select empno, ename, sal into v_t from emp where ename = 'SMITH';
  dbms_output.put_line(v_t.a1 ||' '|| v_t.a2 ||' '|| v_t.a3);
 end p1;
end;
/

Sql> exec pj1.p1;
7369 SMITH 3800
```

Oracle 7.3 introduced "UTL_FILE" package. This package is used to write data into an Operating System file and also read data from an Operating System fie. If you want to write data into an OS file then we using "PUTF()" or "PUT_LINE()" procedure from "UTL_FILE" package where as if we want to read data from an OS file then we are using "get_line()" procedure from "UTL_FILE" package.

In Oracle, before we are using "UTL_FILE" package then we must create alias directory related to physical directory based on following syntax.

Syntax: create or replace directory directoryname as 'path';

Before we are creating alias directory then data base administrator must use create any directory system privilege given to the user. Otherwise oracle server return an error.

Syntax: grant create an directory to username1, username2,.....;

```
Sql> conn sys as sysdba;
Password: sys

Sql> grant create any directory to scott;
Sql> conn scott/tiger;

Sql> create or replace directory XYZ as 'C:\';
Directory Created
```

Before we are performing READ, WRITE operations in operating system file then database administrator must give READ, WRITE object privilege on alias directory by using following syntax.

Syntax: grant read, write on directory directoryname to username1, username2,.....;

```
sql> conn sys as sysdba;
password: sys
sql> grant read, write on directory XYZ to scott;
sql> conn scott/tiger;
```

Writing Data Into OS File

Step 1: Before we are opening the file then we must declare a file pointer variable by using "file_type" from "UTL_FILE" package in declare section of the PL/SQL block.

Syntax: filepointer variablename utl_file.file_type;

Step 2: Before we are writing data into an external file then we must open the file by using "**FOPEN()**" function from "UTL_FILE" package. This function is used in "Executable section" of the PL/SQL block. This function accepts "3" parameters and returns "file_type" datatype.

Syntax: Filepointervariablename:= utl_file.fopen('alias directoryname', 'filename', 'mode');

Mode-----> 3 types(**w**→Write, **r**→Read, **a** →Append)

Step 3: If we want to write data into file then we are using "**PUTF**" procedure from "**UTL_FILE**" package.

Syntax: utl_file.putf(filepointervariablename, 'content');

Step 4: After writing data into file then we must close the file by using "**FCLOSE**" procedure from "UTL_FILE" package.

Syntax: Utl_file.fclose(filepointer variablename);

Example:

```
declare
 fp utl_file.file_type;
begin
 fp := utl_file.fopen('XYZ', 'file1.txt', 'w');
 utl_file.putf(fp, 'abcdefg');
 utl_file.fclose(fp);
end;
/
```

- **Write a PL/SQL program to retrieve all employee names from emp table and store it into an Operating System file by using "UTL_FILE" package?**

Ans:

```
declare
  fp utl_file.file_type;
  cursor c1 is select * from emp;
begin
 fp := utl_file.fopen('XYZ', 'file1.txt', 'w');
 for i in c1 loop
   utl_file.putf(fp, i.ename);
 end loop;
 utl_file.fclose(fp);
end;
/
```

Note: Whenever we are writing data into an external file by using "UTL_FILE" package "PUTF" procedure then automatically table column data stored in horizontal manner within a file.

To overcome this problem if we want to store our own format then we must use "%s" access specifier within second parameter of the "PUTF" procedure and also if we want to store every data item in next line then we are using "**\n**" along with "%s" access specifier. This parameter must be specified within ' ' (single quotes).

Syntax: utl_file.putf(filepointervariablename, 'format',variablename);

Example:

```
declare
   fp utl_file.file_type;
   cursor c1 is select * from emp;
begin
  fp := utl_file.fopen('XYZ', 'file2.txt', 'w');
  for i in c1
loop
    utl_file.putf(fp, '%s\n',i.ename);
  end loop;
  utl_file.fclose(fp);
end;
```

Note: We can also write data into an OS file by using **put_line()** procedure from utl_file package. This procedure accepts 2 parameters.

Syntax: utl_file.put_line(filepointervariablename, variablename);

➢ **Write a PL/SQL program which is used to store employee details from emp table into an OS file by using put_line procedure from utl_file package?**

Ans:

```
declare
 fp utl_file.file_type;
 cursor c1 is
  select * from emp;
begin
 fp := utl_file.fopen('XYZ', 'file3.txt', 'w');
 for i in c1
loop
  utl_file.put_line(fp, i.ename || ' ' || i.sal);
 end loop;
 utl_file.fclose(fp);
end;
```

Example:

```
declare
 fp utl_file.file_type;
begin
 fp := utl_file.fopen('XYZ', 'file4.txt', 'w');
 utl_file.putf(fp, 'one');
 utl_file.putf(fp, 'two');
 utl_file.fclose(fp);
end;
```

Output: one two

Example:

```
declare
 fp utl_file.file_type;
begin
 fp := utl_file.fopen('XYZ', 'file5.txt', 'w');
 utl_file.putf(fp, 'one' || chr(10));
 utl_file.putf(fp, 'two');
 utl_file.fclose(fp);
end;
```

Output:

```
one
two
```

If we want to read data from an Operating System file then we must use "**get_line**" procedure from "**utl_file**" package this procedure accepts 2 parameters before we are using this procedure then we must use read mode (r) within **fopen()**.

Syntax: utl_file.get_line(filepointer variablename, buffer variablename);

Before we are using this procedure then we must use "Read mode (r)" in "fopen" function from "utl_file" package.

- **Write a PL/SQL program by using utl_file package read data from file1.txt and display that data?**

Ans:

```
declare
 fp utl_file.file_type;
 x varchar2(200);
begin
 fp := utl_file.fopen('XYZ', 'file1.txt', 'r');
 utl_file.get_line(fp, x);
 dbms_output.put_line('data from file' || ' ' || x);
 utl_file.fclose(fp);
end;
/
```

Output: abcdefg

- **Write a PL/SQL program which is used to read multiple data items from 'file2.txt' by using using utl_file package and also display content from file?**

Ans:

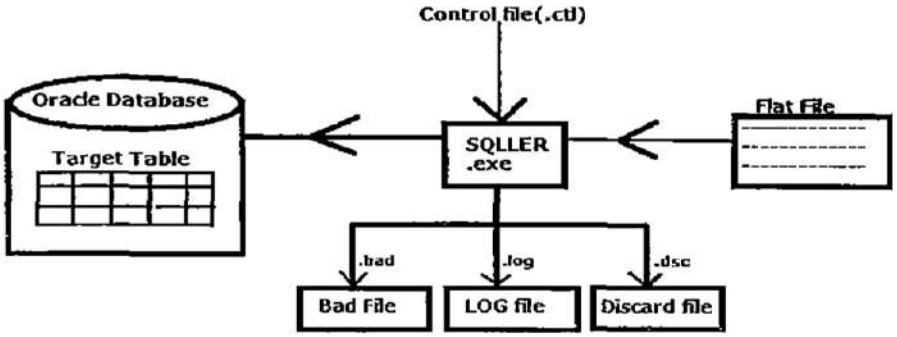
```
declare
 fp utl_file.file_type;
 x varchar2(200);
begin
 fp := utl_file.fopen('XYZ', 'file2.txt', 'r');
 loop
   utl_file.get_line(fp, x);
   dbms_output.put_line(x);
 end loop;
 utl_file.fclose(fp);
exception
 when no_data_found then
   null;
end;
```

Note: In oracle whenever we are reading multiple data items from an external file by using utl_file package then Oracle server returns and error "**ORA-1403: no data found**" whenever control reached end of file.
To overcome this problem we must use "**no_data_found**" exception name.

SQL Loader is an utility program which is used to transfer data from **"Flat File"** into Oracle database. SQL Loader tool is also called as **"Performance Loader"** or "Bulk Loader".

SQL Loader always executes **"Control files"** this file extension is **".ctl"** based on the **"flat file"** we are creating a **"control file"** into SQL Loader tool. Whenever we are submitting **"control file"** into the SQL Loader tool then only SQL Loader tool transfer data from flat file into Oracle database.

During this process SQL Loader tool automatically creates **"Log file"** as same name as **"Control file"**. This **"Log file"** stores all other files information and also these files stores Loaded, Rejected number of records "numbers". This log file also stores Oracle error numbers, error messages.



Whenever we are transferring data from flat file into Oracle database based on some reasons some records are not loaded into database those records are also called as **"Rejected records"** (or) **"Discard records"**. Those rejected (or) discarded records also stored in **"Bad Files", "Discard Files"**.

Bad files stores rejected records based on data type mismatch, business rules violation. Discard file also stores rejected records based on when condition within "Control file".

FLAT FILES

Flat file is a structured file which contains number of records. All database systems having two types of Flat files.

1) **Variable Length Record Flat file.**
2) **Fixed Length Record Flat file.**

1. **Variable Length Record Flat file:** A Flat file which having "Delimeters" in between the fields is called "Variable length record Flat file".

Example:

101, abc, 2000
102, kkk, 9000
103, pqr, 7000

2. **Fixed Length Record Flat File:** A Flat File which does not have "Delimiter" in between fields is called "Fixed Length Record Flat File".

Example:

101 abc 2000
102 kkk 9000
103 pqr 7000

Control file: Based on the type of flat file we are creating control file this file extension is (.ctl) SQl loader tool always execute control file.

Creating A Control File For Variable Length Record Flat File:

Always control file execution starts with "**Load data**" clause. After "**load data**" clause we must specify path of the "**flat file**" by using "**Infile**" clause.

Syntax: load data infile 'path of the flat file'

After specifying path of the "**flat file**" then we are loading data into Oracle database by using "**into table tablename**" clause.

Before "**into table tablename**" clause we are using "**insert/append/replace/truncate**" clauses. If your target table is an empty table then we are using "**insert**" clause. By default clause is "**insert**" clause.

Syntax: insert/append/truncate/replace into table tablename

Based on the type of flat files data after "into table tablename" clause we are specifying following clauses. These are...

1) **Fields terminated by 'Delimiter name'**
2) **Optionally enclosed by 'Delimiter name'**
3) **Trailing 'Nullcols'**

After specifying these clauses then we must specify oracle table columns within paranthesis.

Control File (.ctl):

Syntax:

```
load data
infile 'path of flat file'
badfile 'path of bad file'
discardfile 'path of discard file'
insert/append/replace/truncate
into table tablename
fields terminated by 'delimeter name'
optionally enclosed by 'delimeter name'
trailing nullcols
(col1,col2,col3,.....)
```

Invoking SQL Loader:

Start → Run → cmd → c:\sqlldr userid= scott/tiger (press "Enter button")
Control= path of the control file

Example:
Step 1: Creating text file file1.txt

```
101,abc,3000
102,xyz,4000
103,pqr,9000
104,mmm,5000
```

Save this "note pad" file as "**file1.txt**".

Step 2: Creating table in Oracle database.

```
Sql> create table target(empno number(10), ename varchar2(10), sal number(10));
```

Step 3: Creating Control file (.ctl) by opening new note pad and write following code.

```
load data
infile 'C:\ file1.txt'
insert into table target
fields terminated by ','
(empno, ename, sal)
```

Now save the file as "save as" → "shailendra.ctl" and "type" → "all types".

Step 4: Execution process

Goto Command prompt and to the directory where "**sql loader**" is stored.

```
C:\ sqlldr userid= scott/tiger
Control= C:\ shailendra.ctl
```

Now goto Oracle database and check to know the data is dumped into target file.

```
Sql> select * from target;
```

During this process "SQL LOADER" tool automatically creates a "**Log File**" as same name as "**Control file**". This "**Log file**" stores all other files information and also stores Loaded, Rejected number of records "Numbers". This "**Log file**" also stores Oracle error numbers, error message.

Note: We can also use flat file data within "**control file**" itself. In this case we must use "*" in place of path of the "**flat file**" within "**INFILE**" clause and also we must specify "**begindata**" clause in the above of the "**flat file**" data.

Example:

Step 1: Creating table in Oracle database.

```
Sql> create table target(empno number(10), ename varchar2(10), sal number(10));
```

Step 2: Creating Control file(.ctl) by opening new note pad and write following code.

```
Inserting data directly through Control File
load data
infile *
insert into table target
field terminated by ','
(empno, ename, sal)
begindata
101,abc,2000
102,xyz,5000
```

Step 3: Now execute the "Control files" as using command prompt.

```
C:\ sqlldr userid= scott/tiger
Control= C:\ shailendra.ctl
```

Now goto Oracle database and check to know the data is dumped into target file.

```
Sql> select * from target;
```

CONSTANT: If we want to store default values into Oracle database by using SQL Loader then we must use "CONSTANT" clause within control file.

Syntax: Columnname constant 'default value'

→ Default value means 'NUMBERS also in single quotes (' ')

Whenever flat file having less number of columns and also if target table require more number of columns then only we are using "CONSTANT" clause.

FILLER: If you want to skip columns from flat files then we must use "**filler**" clause. Whenever flat file having more number of columns and also oracle table required less number of columns then only we are using "**filler**" clause.

Syntax: columnname filler (or) anyname filler

Example:

Step 1: creating flat file file1.txt

```
101, abc
102, xyz
103,pqr
```

Step 2: Creating target table in database

```
Sql> create table target(empno number(10), ename varchar2(10), loc varchar2(10));
```

Step 3: Creating Control file (.ctl) by using Constant and Filler.

```
load data
infile 'C:\ file1.txt'
insert
into table target
fields terminated by ','
(empno, ename filler, loc constant 'hyd')
```

Here second column ename is skipped by using "FILLER" clause and 'loc' column is filled with "hyd" by using "CONSTANT" clause.

BAD file:

This file extension is .bad based on some reason some record are rejected those rejected records are automatically stored in "**bad file**". Bad files are automatically created as same name as "**flat file name**".

Note: We can also create bad file explicitly by specifying "**bad file**" clause within "control file" itself.

Syntax: badfile 'path of badfile'

Bad files stores rejected records based on following 2 reasons:

1) **Data type mismatch**
2) **Business rule violation**

1) **Data type mismatch**

Example:

Step 1: Create a flat file 'file1.txt'

```
101,abc
102,xyz
103,pqr
104,kkk
```

Step 2: Creating a Target file

```
Sql> create table target(empno number(10), ename varchar2(10));
```

Step 3: Creating a Control file(.ctl)

```
load data
infile 'C:\ file1.txt'
insert
into table target
fields terminated by ','
(empno, ename)
```

Step 4: Execution of 'text1.txt' file

```
C:\ sqlldr userid= scott/tiger
```

→Bad file file1.bad

```
'102',abc
'103',pqr
```

2) **Based on Business Rule Violations:**

Example:

Step 1: Creating a flat file "file1.txt"

```
101,abc,9000
102,xyz,2000
103,pqr,8000
104,kkk,3000
```

Step 2: Creating a table in Oracle database.

```
Sql> create table targer(empno number(10), ename varchar2(10), sal number(10) check(sal>5000));
```

Step 3: Creating a Control file "shailendra.ctl"

```
load data
infile 'C:\ file1.txt'
insert into table target
fields terminated by ','
(empno, ename, sal)
```

Step 4: Execute through command prompt

Step 5: sql> select * from target;

Bad file:

```
102, xyz, 2000
104, kkk, 3000
```

Note: Whenever flat file having null values in last fields of a column then SQLLDR rejected those null value records and also those null value records are automatically stores in "Bad file". If we want to store those null value records in Oracle database then we must use **"Trailing nullcols"** clause within control file.

Example:

→**Creating text file file1.txt**

```
101,abc,9000
102,xyz
103,pqr
104,kkk,3000
```

→**Creating target file in Oracle database**

```
Sql> create table target(empno number(10), ename varchar2(10), sal number(10));
```

→**Creating control file "shailendra.ctl"**

```
load data
infile 'C:\ file1.txt'
insert into table target
fields terminated by ','
trailing nullcols
(empno, ename, sal)
```

→**Executing through command prompt**

→Sql> select * from target;

RECNUM

This clause is used in control file only. This clause automatically assigns numbers to loaded, rejected,skipped number of records.

Syntax: Columnname recnum

Example:

→Creating flat file file1.txt

```
101,    abc
'102',  xyz
'103',  pqr
104,    kkk
```

→Creating table in Oracle database

```
Sql> create table target(empno number(10), ename varchar2(10), rno number(10));
```

→Creating a Control file(.ctl) 'shailendra.ctl'

```
load data
infile 'C:\ file1.txt'
insert
 into table target
fields terminated by ','
(empno, ename, rno recnum)
```

→Sql> select * from target;

EMPNO	ENAME	RNO
101	abc	1
104	kkk	4

Bad file:

```
'102',xyz
'103',pqr
```

We can also use Oracle predefined, user defined functions in "Control files".
In this case we must specify function functionality within double quotes (" ") and also we must use colon operator (:) infront of the column name within function functionality.

Syntax: columnname "functionname(: columnname)"

➢ **Develop a control file based on the following conditions within last column of the following flat file.**
If gender = 'm' then convert m --- > male
If gender = 'f' then convert f --- > female

Ans:

→Creating a flat file file1.txt

```
101, abc, m
102, xyz, f
103, pqr, m
104, kkk, f
```

→Creating table in Oracle database.

```
Sql> create table target(empno number(10), ename varchar2(10), gender varchar2(10));
```

→Creating a control file 'shailendra.ctl'

```
load data
infile 'C:\ file1.txt'
insert
into table target
fields terminated by ','
(empno, ename, gender "decode(:gender, 'm', 'male', 'f', 'female')")
```

→Sql> select * from target,

EMPNO	ENAME	GENDER
101	abc	male
102	xyz	female
103	pqr	male
104	kkk	female

Method 1: using Data type
Method 2: using to_date()

Method 1: Using Data Type

Syntax: colname date"flatfile dateformat"

Example:

→Creating a flat file file1.txt

```
101,abc,120705
102,xyz,240307
103,pqr,190804
```

→Creating a table in Oracle database

```
Sql> create table target(empno number(10), ename varchar2(10), col3 date);
```

→Creating a Control file 'shailendra.ctl'

```
load data
infile 'C:\ file1.txt'
insert
into table target
fields terminated by ','
(empno, ename, col3 date "DDMMYY")
```

→Sql> select * from target

```
EMPNO ENAME  COL3
--------- --------- ---------
101       abc   12-JUN-05
102       xyz   24-MAR-07
103       pqr   19-AUG-04
```

Method 2: Using to_date()

→Creating a control file 'murali.ctl'

```
load data
infile 'C:\ file1.txt'
insert into table target
fields terminated by ','
(empno, ename, col3 "to_date(:col3, 'DDMMYY')")
```

Discard file extension is (.dsc). Discard file also stores rejected records based on "**when**" condition within "Control file" (.ctl). This "**when**" condition must be specified in after "**into table tablename**" clause within "control file".
Generally "**Bad files**" are created automatically but "**discard file**" are not created automatically in this case we must specify discard file within "**control file**" by using "**discardfile**" clause.

Syntax: discardfile 'path of the discard file'
Syntax: when condition

Example:
→Creating a flat file file1.txt

```
101,abc,10
102,xyz,20
103,pqr,10
104,kkk,30
```

→Creating a table in Oracle database

```
Sql> create table target(empno number(10), ename varchar2(10), deptno number(10));
```

→Creating a control file 'shailendra.ctl'

```
load data
infile 'C:\file1.txt'
discardfile 'C:\file1.dsc'
insert
into table target
when deptno = '10'
fields terminated by ','
(empno, ename, deptno)
```

→Executing through command prompt
→Sql> select * from target;

EMPNO	ENAME	DEPTNO
101	abc	10
103	pqr	10

Discard file:

```
102, xyz, 20
104, kkk, 30
```

Note 1: Always "**WHEN**" condition values must be specified within single quotes (' ').

Note 2: In "**WHEN**" clause we are not allowed to use other than "=", "<>" Relational operator.

Note 3: In "**WHEN**" clause we are not allowed to use logical operator "OR" but we are not allowed to use logical operator "**AND**".

Through the control file we can also skip number of records from flat file. In this case we must use option clause in the above of the load data clause within option clause we must specified skip number of records by using skip clause.

Syntax: options(skip = number)

Example:

→Creating a flat file file1.txt

```
101,abc
102,xyz
103,pqr
104,kkk
```

→Creating a table in Oracle database

```
Sql> create table target(empno number(10), ename varchar2(10));
```

→Creating a control file 'shailendra.ctl'

```
Options (skip=2)
load data
infile 'C:\file1.txt'
insert
into table target
fields terminated by ','
(empno, ename)
```

→Executing through command prompt

→Sql> select * from target;

```
EMPNO    ENAME
---------------------
 103       pqr
 104       kkk
```

When flat file having different deleimetrs then we must use "TERMINATED BY" clause along with column name within control file.

Syntax: columnname terminated by 'delimiter'

Example:

→Creating a flat file file1.txt

```
101,abc $2000 ^10
102,xyz $3000 ^20
103,pqr $4000 ^30
```

→Creating a table in Oracle database

```
Sql> create table target(empno number(10), ename varchar2(10), sal number(10), deptno number(10));
```

→Creating a control file 'shailendra.ctl'

```
load data
infile 'C:\file1.txt'
insert
into table target
fields terminated by ','
(empno, ename terminated by '$',sal terminated by '^',deptno)
```

→Executing through command prompt

→Sql> select * from target;

Empno	Ename	Sal	Deptno
101	abc	2000	10
102	Xyz	3000	20
103	pqr	4000	30

Syntax: colname "sequencename.nextval"

Example:

→Creating a flat file file1.txt

```
101
102
103
104
105
```

→Creating a sequence in Oracle database

```
Sql> create sequence s1;
```

→Creating a table in oracle database

```
Sql> create table target(sno number(10));
```

→Creating a Control file 'shailendra.cti'

```
load data
infile 'C:\ file1.txt'
insert
into table target
fields terminated by ','
(sno "s1.nextval")
```

→Sql> select * from target;

SNO
1
2
3
4
5

A flat file which does not have delimiters is called Fixed length record flat file. When resource has fixed length record flat file then we must use "Position" clause along with column name within control file. In this "position" clause we must specify starting, ending position of the every fields by using (:) colon operator.

After position clause we must specify SQL loader data type. SQL loader having following 3 types of data type.

1) **Integer external**
2) **Decimal external**
3) **Char**

Syntax: position (startingpoint : endingpoint) SQL Loader datatype

Note: Whenever we are using function (or) expression within control file then we are not allow to use SQL loader data type. In place of SQL loader data, we must use function functionality by using following syntax.

Syntax: Colname position (startingpoint : endingpoint) "functionname(: columnname)"

Example:

→Creating a flat file file1.txt

```
101abc2000
102xyz5000
103pqr8000
```

→Creating table in Oracle database

```
Sql> create table target(empno number(10), ename varchar2(10), sal number(10));
```

→Creating a control file 'shailendra.ctl'

```
load data
infile 'C:\ file1.txt'
insert
into table target
(empno position(01:03) integer external,
 ename position(04:06) char,
 sal position(07:10) integer external)
```

→Sql> select * from target;

EMPNO	ENAME	SAL
101	abc	2000
102	xyz	5000
103	pqr	8000

Question By Murali Sir

→Sql> create sequence s1;

→Sql> create table target(sno number(10), value number(10), time date, col1 number(10), col2 varchar2(10), col3 date);

→Creating a Control file

```
load data
infile *
Insert
into table target
(
 sno "s1.nextval",
 value constant '50',
 time "to_char(sysdate, 'HH24:MI:SS')",
 col1 position(01:06) ":col1/100",
 col2 position(07:13) "upper(:col2)",
 col3 position(14:19) "to_date(:col3, 'DDMMYY')"
)
begindata
100000aaaaaaa060805
200000bbbbbbb170903
```

→Sql> select * from target;

SNO	VALUE	TIME	COL1	COL2	COL3
1	50	18:11:12	100000	aaaaaaa	06-AUG-05
2	50	18:11:59	200000	bbbbbbb	17-SEP-03

Note: Whenever we are "SYSDATE", "user functions" then we are not allowed to use ":" (colon) operator along with these operator.

By using SQL Loader we can also transfer multiple flat files data into single Oracle table by using number of **"INFILE"** clauses within control file itself and also we can also transfer single flat file data into multiple Oracle tables by using multiple **"into table tablename"** clauses within control file.

But when resource has combination of flat file, oracle database data (or) when resource as different databases data then SQL Loader cannot transfer this data into Oracle table.

To overcome this problem now a day organization uses data ware housing tools which transferred data into target database.

Trigger is also same as "Stored Procedure" and also it automatically invoked wherever DML operations performed on "Table (or) View".

All relational databases have 2 types of triggers.

1) **Statement Level Trigger**
2) **Row Level Trigger**

In Statement Level Triggers, Trigger body is executed only **"ONCE"** per DML Statement. Whereas in Row Level Trigger, Trigger body is executed for each row for DML Statement.

In oracle every Triggers also has two parts.

1) **Trigger Specification**
2) **Trigger Body**

In trigger specification we are specifying name of the trigger and also type of operation, where as in trigger body we are solving actual task.

Syntax: /*[.....] means optional*/

```
Trigger Timings  create or replace trigger triggername   Trigger Events
                 before/after insert/update/delete on tablename
                 [for each row] ----> only for Row Level Triggers
                 [when condition]
                 [declare]
                 -> variable declarations, cursors, userdefined exception;
                 begin
                 -------
                 -------
                 end;
```

Difference between Statement Level Triggers and Row Level Triggers

Statement Level Triggers:

Example:

```
create or replace trigger tr1
 after update emp
begin
 dbms_output.put_line('Statement level trigger');
end;
/
```

Testing:

```
Sql> update emp set sal=sal+100 where deptno=10;
        Statement level
        3 rows updated.
Sql> update emp set sal=sal+100 where deptno=90;
        Statement level
        0 rows updated.
```

→In Statement Level Trigger number of rows effected (or) not affected also it will fires and print message only once.

Row Level Trigger

Example:

```
create or replace trigger tr1
 after update emp
 for each row
begin
 dbms_output.put_line('Row level');
end;
/
```

Testing:

```
Sql> update emp set sal= sal+100 where deptno=10;
        Row level
        Row level
        Row level
        3 rows updated.

sql> update emp set sal=sal+100 where deptno=90;
        0 rows updated
```

Row Level Triggers

In Row Level Trigger, trigger body is executed for each and row for DML statement, that's why we must use **"for each row"** clause in trigger specification. In oracle whenever we are using Row level trigger then DML transaction values are internally automatically stores in 2 **rollback segment qualifiers.** These are...

1) :new
2) :old

These rollback segment qualifiers are also called as Record type variables. These qualifiers are used in either trigger specification (or) in trigger body.

Syntax- :old.columnname
Syntax- :new.columnname

Note: Whenever we are using these qualifiers in trigger specification then we are not allowed to use ":" in front of the qualifier name.

	INSERT	UPDATE	DELETE
:NEW	YES	YES	NO
:OLD	NO	YES	YES

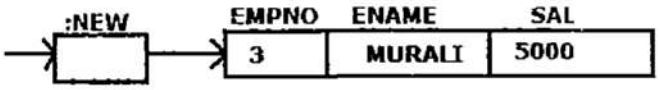
Note: Basically **:old, :new qualifier** pointer variable these pointer variable points to where actual data store in a buffer.

Insertion

EMP

EMPNO	ENAME	SAL
1 2	ABC XYZ	2000 3000

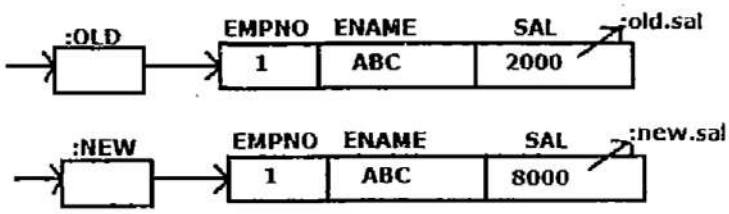
Sql> insert into emp values (3, 'murali', 5000);



Ename → :new.ename
Sal → :new.sal

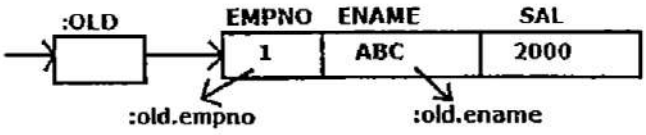
Updation:

Sql> update emp set sal=8000 where empno = 1;



Deletion:

Sql> delete from emp where empno=1;



➢ **Write a PL/SQL row level trigger on employee table whenever user inserting data into emp table then user inserted salary should be more than 5000?**

Ans:

```
create or replace trigger tr1
 before insert on emp
 for each row
begin
 if :new.sal < 5000 then
   raise_application_error(-20143, 'Salary should be less than 5000');
 end if;
end;
/
```

Testing:

```
Sql> insert into emp (empno, ename, sal) values(1, 'abc', 3000);
Error: ORA-20143: Salary should not be less than 5000.

Sql> insert into emp(empno,ename,sal) values(1,'abc',6000);
        1 row inserted.
Sql> select * from emp;
```

- **Write PL/SQL Row Level Trigger on the following table not allows inserting duplicate data into that table (It acts like a primary key)? *****

```
sql> create table test(sno number(10));
Sql> insert into test values(&sno);
10 10 10 20 20 30 30 40
Sql> select * from test;
```

SNO
10
10
20
20
30
30
40

Ans:

```
create or replace trigger tr2
 before insert on test
 for each row
declare
 cursor c1 is select * from test;
begin
 for i in c1
loop
   if i.sno = :new.sno then
     raise_application_error(-20143, 'We cannot insert duplicate values');
   end if;
 end loop;
end;
/
```

Testing:

```
Sql> insert into test values(10);
ERROR: ORA-20143: We cannot insert duplicate values

Sql> insert into test values(50);
1 row inserted.
```

- **Write a PL/SQL Row Level Trigger on the above table not allowed to insert duplicate data into that table without using cursors? *****

Ans:

```
create or replace trigger tr3
 before insert on test
 for each row
  v_count number(10);
begin
 select count(*) into v_count from test where sno = :new.sno;
 if v_count >= 1 then
  raise_application_error(-20143, 'Cannot insert duplicate values');
 elsif v_count = 0 then
  dbms_output.put_line('Your record is Inserted');
 end if;
end;
/
```

Testing:

```
Sql> insert into test values(10);
ERROR: ORA-20143: Cannot insert duplicate values.

Sql> insert into test values(60);
Your record is inserted
1 row inserted
```

Update:

Whenever we are performing update operation through the row level trigger then new value for the qualifier is store in :new qualifier and also old value for the transaction is store in :old qualifier.

- **Write a PL/SQL Row Level Trigger on emp table whenever user modifying salaried then automatically display old salary, new salary, salary difference for the transaction?**

Ans:

```
create or replace trigger tr5
 after update on emp
 for each row
declare
 x number(10);
begin
 x := :new.sal - :old.sal;
 dbms_output.put_line('Old Salary is :' || ' ' || :old.sal);
 dbms_output.put_line('New Salary is :' || ' ' || :new.sal);
 dbms_output.put_line('Difference Salary is :' || ' ' || x);
end;
/
```

Testing:

```
Sql> update emp set sal=sal+100 where ename= 'SMITH';
```

Output:

```
Old Salary is: 800
New Salary is: 900
Difference Salary is: 100
```

- **Write a PL/SQL Row Level Trigger on dept table whenever user modifying deptno in dept table then automatically those modifications are effected in emp table? *****

Ans:

```
create or replace trigger tr6
 after update on dept
for each row
begin
 update dept set deptno = :new.deptno where depnto = :old.deptno;
end;
/
```

Testing:

```
Sql> update dept set deptno=1 where deptno=10;
Sql> select * from dept;
Sql> select * from emp;
```

Delete:

In oracle whenever we are performing deletion operation in row level trigger then those transaction are automatically stores in :old qualifier.

- **Write a PL/SQL Row Level Trigger on emp table not allow to delete 10th department from emp table?**

Ans:

```
create or replace trigger tr7
 before delete on emp
 for each row
begin
 if :old.deptno = 10 then
  raise_application_error(-20123, 'We cannot delete deptno 10');
 end if;
end;
/
```

Testing:

```
Sql> delete from emp where deptno =10;
Error: -20123: we cannot delete deptno 10.
```

- **Write a PL/SQL Row Level Trigger on emp table whenever user deleting data those deleted records are automatically stored in another tables?**

```
sql>create table backup as select * from emp where 1=2;
```

Ans:

```
create or replace trigger tr8
 after delete on emp
 for each row
begin
 insert into backup
 values
  (:old.empno,
   :old.ename,
   :old.job,
   :old.mgr,
   :old.hiredate,
   :old.sal,
   :old.comm,
   :old.deptno);
end;
```

Testing:

```
Sql> delete from emp where deptno=10;
6 rows deleted
Sql> select * from test;
```

Note: In Oracle, we can also change particular column values by using "OF" clauses. These clauses are used in trigger specification only. Insert, Delete events does not contain "OF" clauses.

Syntax: update of columnname;

In oracle row level trigger are used in following applications. These are...

1) **Implementing business rule**
2) **Auditing a columns**
3) **Auto increment**

1) **Implementing business rule:-**

➢ **Write a PL/SQL Row Level Trigger on emp table whenever user inserting data if job is 'CLERK' then automatically set comm of the CLERK to 'NULL' in emp table?**

Ans:

```
create or replace trigger tr9
 before insert on emp
 for each row
 when (new.JOB = 'CLERK')
begin
 if :new.comm is not null then
   :new.comm := null;
 end if;
end;
```

Testing:

```
Sql> insert into emp(empno, ename, job, comm) values(1, 'Shailendra', 'CLERK', 3000);
1 row inserted.
Sql> select * from emp;

EMPNO      ENAME       JOB        COMM
---------- ----------- ---------- -------
   1       SHAILENDRA  CLERK      –
```

Note: In oracle when we are using "when" clause specification then we are not allow to use colon in front of the qualifiers within when clause.

Syntax: new.columnname **Syntax:** old.columnname

2) **Auditing Of Columns:**

In all relational databases, whenever we are modifying a column then those transaction values are automatically stored in another table is called Auditing a Column. Auditing a column also implemented by using row level triggers.

➢ **Write a PL/SQL Row Level Trigger on emp table whenever user modifying values in salary column those transaction details are automatically stored in another table?**

Ans:

```
Sql> create table test(empno number(10),ename varchar2(10),
oldsalary number(10), newsalary number(10), col5 date, username varchar2(10));
```

```
Sql> create or replace trigger tr10
  after update on emp
  for each row
begin
  insert into test values
    (:old.empno, :old.ename, :old.sal, :new.sal, sysdate, user);
end;
/
```

Testing:

```
Sql> update emp set sal = sal+100 where deptno=10;
3 rows updated
Sql> select * from test;
```

Note: In Oracle, we are not allowed to use old, new qualifiers in front of the "SYSDATE", user functions.

3) Auto Increment:

In all relational databases, if we want to generate primary key values automatically then we are using auto increment concept. In Oracle, we are implementing auto increment concept by using **"Row level Triggers, sequences"**. That is, first we are creating a "Sequence" in "SQL" and then only used that sequence in "PL/SQL" row level trigger.

Example:

```
Sql> create table test(sno number(10) primary key, name varchar2(10));
Sql> create sequence s1;

Sql>create or replace trigger th1
  before insert on test
  for each row
begin
  select s1.nextval into :new.sno from dual;
end;
/
```

Testing:

```
Sql> insert into test(name) values('&name');
shailendra
dinesh
naresh
siva
Sql> select * from test;
```

SNO	NAME
1	Shailendra
2	Dinesh
3	Naresh
4	siva.

Here SNO values are generated automatically by using **"AUTO INCREMENT"**.

Note: Oracle 11g, introduced variable assignment concept when we are using "Sequences" in PL/SQL block. In this case we are not allowed to use "DUAL" table or "Select....into" clause.

Syntax:

```
Sql> begin
Variablename:= sequencename.nextval;
end;
```

For previous example [only for 11g, 12c]

```
create or replace trigger tr1
 before insert on test
 for each row
begin
 :new.sno := s1.nextval;
end;
/
```

Example:

```
create or replace trigger tr1
 after insert on test
 for each row
begin
select s1.nextval into :new.sno from dual;
end;
/
```

Generating Alpha Numeric data as Primary key in "AUTO INCREMENT" concept:

Example:

```
sql> create table test(sno varchar2(10) primary key, name varchar2(10));
Sql> create sequence s1;

Sql> create or replace trigger tr1
 before insert on test
 for each row
begin
 select 'ABC' || lpad(s1.nextval, 5, '0') into :new.sno from dual;
end;
/
```

Testing:

```
Sql> insert into test(name) values('&name');
Sql> select * from test;
```

SNO	NAME
ABC00001	Shailendra
ABC00002	Murali
ABC00003	Abc
ABC00004	Xyz
ABC00005	Kkk

Oracle triggers having two timing points. These are...

1) **Before Timing**
2) **After Timing**

1) Before Timing

Whenever we are using **"Before"** timing internally trigger body is executed first then only DML statements are executed. Generally whenever we are specifying **BEFORE** timing DML transaction then those transactions are directly not affected in database, in this case those DML transactions are first effected in trigger body then only those transaction values are stored in database. That's why in **"row level"** trigger whenever we are assigning or modifying **":new"** qualifiers body then we must use before timing. Otherwise Oracle server returns an error cannot change ":new" values for this trigger type.

Syntax:

```
begin
 :new.columnname := value;
end;
```

Note: Generally in all relational database triggers if you want to restrict invalid database entry then also we are allowed to use before timing.

➢ **Write a PL/SQL row level trigger on emp table whenever user inserting data into ename column then automatically inserted value converted into upper case and also the data is stored into ename column?**

Ans:

```
create or replace trigger tr1
 before insert on emp
 for each row
begin
 :new.ename := upper(:new.ename);
end;
/
```

Testing:

```
Sql> insert into emp(empno, ename) values(1, 'shailendra');
```

EMPNO	ENAME
1	Shailendra

Note: In oracle whenever we are using row level trigger particular cell value also represented by using ":new" qualifier.

2) After Timing

Whenever we are using after timing DML transaction value are executed first then only trigger body is executed i.e. whenever we are submitting DML transaction then those transaction values are directly affected within Oracle database then only those values are affected in trigger body.

Generally in all relational database one table transaction value are affected in another table then we are using after timing.

Example:

```
Sql> create table students(stno number(10),sub1 number(10),sub2 number(10),sub3 number(10));
Sql> create table target(stno number(10),total number(10), avg number(10));

Sql> create or replace trigger tr1
 after insert on students
 for each row
begin
 insert into target
 values
  (:new.stno,
   :new.sub1,
   :new.sub2,
   :new.sub3,
   (:new.sub1 + :new.sub2 + :new.sub3) / 3);
end;
/
```

Testing:

```
sql> insert into students values (1, 30, 40, 90);
sql> insert into students values (2, 10, 20, 30);
```

STNO	TOTAL	AVG
1	160	53
2	60	20

In statement level triggers, trigger body is executed only once per DML statements. Statement level triggers does not have "for each row" clause and also in oracle statement level trigger does not have ":old", ":new" qualifiers.

Generally, when we are implementing time component based application by using trigger then we must use statement level triggers. Generally in all relational databases by default statement level trigger performance is very high compare to row level trigger.

Example:

- **Write a PL/SQL statement level trigger on emp table not allow to perform DML operations on Saturday and Sunday?**

Ans:

```
create or replace trigger tr1
 before insert or update or delete on emp
begin
 if to_char(sysdate, 'DY') in ('SAT', 'SUN') then
  raise_application_error(-20143, 'We cannot perform DML operations on Saturday and Sunday');
 end if;
end;
/
```

Testing:

```
/*First change system date to Saturday or Sunday and then perform testing*/
Sql> delete from emp where deptno=10;
Error: ORA-20143: We cannot perform DML operations on Saturday and Sunday
```

Note: In Oracle we are not allowed to use "when" clause in "Statement Level Trigger".

Note: We can also convert statement level trigger into row level triggers and row level trigger into statement level triggers without having qualifiers (old,new).

Converting above Statement Level Trigger into Row Level Trigger.

```
create or replace trigger tr1
 before insert or update or delete on emp
 for each row
 when (to_char (sysdate, 'DY') in ('SAT', 'SUN'))
begin
 raise_application_error(-20143,'We cannot perform DML operations on Saturday and Sunday');
end;
/
```

Note: In Oracle we can also convert statement level trigger into row level trigger and also convert row level trigger into statement level trigger without using ":old", ":new" qualifier and also by default in all databases, statement level triggers performance is very high compared to row level triggers. Because in statement level triggers trigger body is executed only once per DML statements.

➢ **Write a PL/SQL Statement Level Trigger on emp table not allow to perform DML Operations in last date of the month?**

Ans:

```
create or replace trigger tr1
 before insert or delete or update on emp
begin
 if sysdate = last_day(sysdate) then
  raise_application_error(-20143, 'Cannot perform DML on last day of the month');
 end if;
end;
/
```

Testing:

Sql> delete from emp where deptno=10;

ORA-20143: Cannot perform DML on last day of the month

Trigger Execution Order

1) **Before Statement Level**
2) **Before Row Level**
3) **After Row Level**
4) **After Statement Level**

Example:

```
sql> create table test(sno number(10));

Sql> create or replace trigger tz1
 after insert on test
 for each row,
begin
 dbms_output.put_line('After Row Level');
end;
/

Sql> create or replace trigger tz2
 after insert on test
begin
 dbms_output.put_line('After Statement Level');
end;
/

Sql> create or replace trigger tz3
 before on test
 for each row
begin
 dbms_output.put_line('before Row Level');
end;
/

Sql> create or replace trigger tz4
 before insert on test
begin
 dbms_output.put_line('Before Statement Level');
end;
/
```

Testing:

```
Sql> insert into test values(10);

before statement level
before row level
after row level
after statement level
```

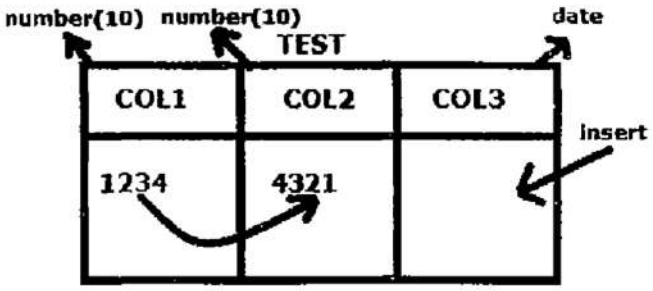
FOLLOWS Clause: (Guarantee Execution Order)

Oracle 11g, introduced "FOLLOWS" clause in triggers. Generally, whenever we are using same level of triggers on same table then we cannot control execution order of the triggers.

To overcome this problem Oracle 11g introduced "FOLLOWS" clause in trigger specification. Whenever we are using "follows" clause we can also control execution order of the trigger explicitly when we are using same level of trigger on same table.

Syntax:

```
create or replace trigger triggername
before/after insert/update/delete on tablename
[for each row]
[when (condition)]
follows another triggername1, another triggername2,.....
[declare]
-------
-------
begin
--------
---------
[execption]
-------
-------
end;
```



Example:

```
sql> create table test(col1 number(10), col2 number(10), col3 date);
Sql> create sequence s1
     start with 1234;

Sql> create or replace trigger tr1
      before insert on test
      for each row
     begin
      select s1.nextval into :new.col1 from dual;
      /* :new.col1= s1.nextval; (for 11g) */
      dbms_output.put_line('Trigger1 fired');
     end;
     /

Sql> create or replace trigger tr2
      before insert on test
      for each row
     begin
      select reverse(to_char(:new.col1)) into :new.col2 from dual;
      dbms_output.put_line('Trigger2 fired');
     end;
     /
```

Testing:

```
Sql> insert into test(col3) values(sysdate);
Trigger2 fired
Trigger1 fired

Sql> select * from test;
```

Output:

COL1	COL2	COL3
1234		03-Aug-19

Solution: (FOLLOWS clause using in Oracle 11g)

```
Sql> create or replace trigger tr1
       before insert on test
       for each row
      begin
       select s1.nextval into :new.col1 from dual;
       /* :new.col1= s1.nextval; (for 11g) */
       dbms_output.put_line('Trigger1 fired');
      end;
      /

Sql> create or replace trigger tr2
       before insert on test
       for each row
       follows tr1
      begin
       select reverse(to_char(:new.col1)) into :new.col2 from dual;
       dbms_output.put_line('Trigger2 fired');
      end;
      /
```

Testing:

```
Sql> insert into test(col3) values(sysdate);
Trigger1 fired
Trigger2 fired

Sql> select * from test;
```

Output:

COL1	COL2	COL3
1234	4321	03-Aug-19

Note: In Oracle, if we want to provide guarantee execution order when we are using same level of triggers on same table then we must use "FOLLOWS" clause.

Oracle 11g, introduced "Compound Trigger". Compound Triggers allows different blocks within a trigger to be executed at different timing points. Compound Trigger also having global declaration section same like a packages.

Syntax:

```
create or replace trigger triggername
 for insert or update or delete on tablename
 compound trigger
 →Global Variable declarations;
 →Types declarations;
 before statement is
 begin
  ------
  ------
 end [ before statement ];
 before each row is
 begin
  ------
  ------
 end [ before each row ];
 after each row is
 begin
  ------
  ------
 end [ after each row ];
 after statement is
 begin
  ------
  ------
 end [ after statement ];
end;
```

➢ **Write a PL/SQL compound trigger which is used to display default execution order of trigger?**

Ans:

```
sql> create table test(sno number(10));

Sql> create or replace trigger tr1
       for insert on test
       compound trigger
       before statement is
       begin
        dbms_output.put_line('Before Statement Level');
       end [ before statement ];
       before each row is
       begin
        dbms_output.put_line('Before Row Level');
       end [ before row level ];
       after each row is
       begin
        dbms_output.put_line('After Row Level');
       end [ after each row ];
```

```
 after statement is
 begin
   dbms_output.put_line('After Statement Level');
 end [ after statement level ];
end;
```

Testing:

```
Sql> insert into test values(10);

before statement level
before row level
after row level
after statement level
```

Instead of trigger

Oracle 8i introduced instead of trigger. Instead of trigger are created on view by default instead of trigger are row level trigger.

Generally we cannot perform DML operations through complex view to base table. To overcome this problem Oracle 8i introduced instead of trigger in PL/SQL when we are creating instead of trigger on complex view then only we are allow performing DML operation through complex view to base table, that's why instead of trigger convert non-updatable views into updatable views.

Syntax:

```
create or replace trigger triggername
instead of insert/update/delete on viewname
for each row
 [declare]
 ----------
 ----------
 begin
 ----------
 ----------
 end;
```

Example:

```
Sql> create table test1 (name varchar2 (10));
Sql> create table test2 (name varchar2 (10));

Sql> create or replace view v1 as select name, sub from test1,test2;

Sql> insert into v1 values('shailendra', 'oracle');
Error: Cannot modify a column which maps to a non key-preserved table.
```

Solution (instead of trigger)

```
create or replace trigger tr2
 instead of insert on v1
 for each row
begin
 insert into test1 (name) values (:new.name);
 insert into test2 (sub) values (:new.sub);
end;
```

Testing:

```
sql> insert into v1 values ('shailendra', 'oracle');
1 row created
Sql> select * from v1;
```

NAME	SUB
Shailendra	Oracle

- **Write a PL/SQL trigger on emp table whenever user deleting rows from the table then automatically display remaining number of records number from emp table at the bottom of the deleted statements after deleting records?**

Ans:

```
create or replace trigger tr1
 after delete on emp
declare
 v_count number(10);
begin
 select count(*) into v_count from emp;
 dbms_output.put_line(v_count);
end;
```

Testing:

```
Sql> delete from emp where empno=3;
1 row deleted
```

Example:

```
create or replace trigger tr2
 after delete on emp
 for each row
declare
 v_count number(10);
begin
 select count(*) into v_count from emp;
 dbms_output.put_line(v_count);
end;
/

Trigger created
```

Testing:

```
Sql> delete from emp where empno=7902;
Error: ORA-04091:table SCOTT.EMP is mutating.
```

When a Row Level Trigger based on a table then trigger body cannot read data from same table and also we cannot perform DML operations on same table. If you are trying this then Oracle server return an error "ORA-04091: table is mutating". This error is called as "Mutating Error".

Mutating error is an "Run time" error occurred in Row Level Triggers only. Mutating Error does not occurred in "Statement Level Trigger". Generally whenever we are using Statement Level Triggers then DML transactions value are internally automatically committedthats why trigger body read this commited data without any problem.

Whereas whenever we are using Row Level Trigger then DML transaction values are not committed automatically on database. Using trigger body when we try to read not committed data by using same table then database server returns mutating errors. That's why Row Level Triggers only returns mutating errors.

To overcome this problem for avoiding "mutating errors" we make it those transaction independent by using "autonomous" transactions in triggers. In this case we must use "automomous_transaction" pragma in declare section of the trigger and also we must use commit in trigger body. Autonomous transactions automatically avoids rotating errors also these transactions returns previous results.

Solution (By using autonomous transaction)

```
create or replace trigger tr1
 after delte on emp
 for each row
declare
 v_count number(10);
 pragma autonomous_transaction;
begin
 select count(*) into v_count from emp;
 commit;
 dbms_output.put_line(v_count);
end;
/
```

Testing:

```
Sql> delete from emp where empno= 7902; 14
13
1 row deleted

Sql> commit;
Sql> delete from emp where empno= 7389;
12
1 row deleted
```

In oracle we can also use multiple tables within trigger body and also we are allowed to perform number of operations in this table in trigger body.

In Oracle, triggers if we want to define multiple conditions on different tables within trigger body then we are using trigger body events. These are Inserting, Updating and Deleting clauses.

These triggering events are also called as Trigger Predicate clauses. These triggering events are used in either Statement Level triggers (or) Row Level triggers. These triggering events are used in within trigger body only.

Syntax:

```
if inserting then
stmts;
elsif updating then
stmts;
elsif deleting then
stmts;
end if;
```

- **Write a PL/SQL statement level trigger on emp table not allow to perform any DML operation in any days by using triggering events?**

Ans:

```
create or replace trigger tr1
 before insert or delete or update on emp
begin
 if inserting then
  raise_application_error(-20123, 'We cannot perform Inserting');
 elsif updating then
  raise_application_error(-20124, 'We cannot perform Updating');
 elsif deleting then
  raise_application_error(-20125, 'We cannot perform Deleting');
 end if;
end;
/
```

Testing:

```
Sql> delete from emp where deptno=10;
ORA-20111: We cannot perform Deleting
```

Example:

```
Sql> create table test(msg varchar2(20));
Sql> create or replace trigger tr1
       before insert or update or delete on emp
      declare
       z varchar2(20);
      begin
       if inserting then
        z := 'rows Inserted';
       elsif updating then
        z := 'rows Updated';
       elsif deleting then
        z := 'rows Deleted';
       end if;
       insert into test values (z);
      end;
```

Testing:

```
Sql> insert into emp(empno) values(1);
Sql> insert into emp(empno) values(2);
Sql> update emp set sal= sal+100 where deptno=10;
Sql> delete from emp where empno in(1,2);
Sql> select * from test;

        MSG
    --------------------
    Rows inserted
    Rows inserted
    Rows updated
    Rows deleted
```

- **Write a PL/SQL Row Level Trigger on emp table whenever user inserting data into emp table then those transactions are automatically stored in another table and also whenever user modifying data on emp table and then those transactional details are stored in another table and also whenever user deleting data then those deleted records are stored in another table?**

Ans:

```
Sql> create table t1(empno number(10), name varchar2(10), salary number(10));
Sql> create table t2(empno number(10), name varchar2(10), salary number(10));
Sql> create table t3(empno number(10), name varchar2(10), salary number(10));
Sql> create or replace trigger tr2
       after insert or update or delete on emp
       for each row
      begin
       if inserting then
        insert into t1 values (:new.empno, :new.ename, :new.sal);
       elsif updating then
        insert into t2 values (:old.empno, :old.ename, :new.sal);
       elsif deleting then
        insert into t3 values (:old.empno, :old.ename, :old.sal);
       end if;
      end;
```

Testing:

```
Sql> delete from emp where deptno=10;
Sql> select * from t3;
```

In Oracle. we can also defined logical conditions in trigger specification by using "WHEN" clause.

This condition always return Boolean value either true or false, whenever condition is true then only trigger body is executed where as condition is false trigger body never executed.

Syntax: when (condition)

Note 1: Here when condition must be an SQL expression.

Note 2: Whenever we are using "WHEN" clause then we are not allowed to use ":" colon infront of the qualifier within "WHEN" clause logical condition.

Note 3: In Oracle when condition is used in for row level trigger only.

Note 4: Generally when trigger having when condition then those trigger improve performance of the application.

Example:

```
create or replace trigger tr1
 after insert on emp
 when (new.empno = 1)
begin
 dbms_output.put_line('When empno = 1 then only my trigger body is executed');
end;
./
```

Testing:

```
Sql> insert into emp(empno) values(1);
When empno=1 then only my trigger body is executed
1 row inserted

Sql> insert into emp(empno) values(2);
1 row inserted
```

Calling Procedure In Trigger

In Oracle, we can also "CALL" stored procedure into trigger by using "CALL" statement.

Syntax:

```
create or replace trigger triggername
before/after insert/update/delete on tablename
call procedurename
```

Example:

```
Sql> create table test(totalsal number(20));

Sql> create or replace procedure p1 is
       v_sal number(20);
     begin
       delete from test;
       select sum(sal) into v_sal from emp;
       insert into test values (v_sal);
     end;
     /

Sql> create or replace trigger tg1
       after insert or update or delete on emp
       call p1
       /
```

Testing:

```
Sql> update emp set sal= sal+100;
Sql> select * from test;

       TOTSAL
     ----------
       32125
```

Note: In oracle triggers we can also use commit command but in this case Oracle server return an runtime error ORA-4092: cannot commit in a trigger. If you want to use commit command in trigger then we must use **"autonomous transaction"** in trigger.

Example:

```
Sql> create table target(sno number(10));

Sql> create or replace trigger tr1
       after insert on emp
       for each row
     begin
       insert into target values (:new.empno);
       commit;
     end;
     /
     Trigger created
```

Testing:

```
Sql> insert into emp(empno) values(5);
Error: ORA-4092: cannot commit in a trigger.
```

Solution (By using autonomous transaction)

```
create or replace trigger tr1
 after insert on emp
 for each row
declare
 pragma autonomous_transaction;
begin
 insert into target values (:new.empno);
 commit;
end;
/
```

Testing:

```
Sql> insert into emp(empno) values(5);
1 row created
Sql> select * from emp;
Sql> select * from target;
```

Enable/Disable a trigger

In oracle by using alter command we can also enable/disable single trigger in a table.

Enable/Disable trigger

Syntax: alter trigger triggername enable/disable;

Enable/Disable all trigger in a table:

Oracle 8i onwards we can also enable/disable all trigger in a table by using following syntax.

Syntax: alter table tablename enable/disable all triggers;

Example: Sql> alter table emp disable all triggers;

In all relational databases we can also craete trigger based on DDL events. These type of triggers are also called as System Trigger (or) DDL Triggers. These Triggers are created by database Administrators.

In Oracle, we are defining system triggers in two levels. These are database level, schema level.
For creating DDL triggers Oracle provided predefined trigger events attribute functions, these functions are Ora_dict_obj_owner, Ora_dict_obj_type, Ora_dict_obj_name

Syntax:

```
create or replace trigger triggername
before/after create/alter/drop/truncate/rename on database (or) username.schema
[declare]
------
------
begin
------
------
end;
```

Example:

- **Write a PL/SQL DDL trigger of SCOTT schema not allow to drop emp table?**

```
create or replace trigger tr1
 before drop on SCOTT.schema
begin
 if ora_dict_obj_name = 'EMP' and ora_dict_obj_type = 'TABLE' then
  raise_application_error(-20123, 'We cannot drop EMP table');
 end if;
end;
```

Testing:

```
Sql> drop table emp;
Error: ORA-20123: We cannot drop emp table
```

- **Write a PL/SQL DDL trigger on database level not allow to create any database object?**

Ans:

```
sql> conn sys as sysdba
     Password:sys

Sql> create or replace trigger tr1
 before create on database
begin
 raise_application_error(-20123, 'Not allow to create any database object');
end;
/
```

Testing:

```
sqi> drop table emp;
Error: ORA-20123: Not allow to create any database object
```

Oracle having 12 types of triggers based on statement level, before,after,insert,update,delete statements.

Oracle also supports system triggers same like all other relational databases and also Oracle having instead of trigger or views.

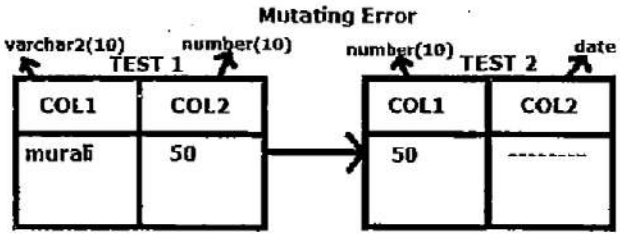
In oracle all trigger information stored under "**user_triggers**" data dictionary.

Example: desc user_triggers;

In oracle we can also drop a trigger by using **drop trigger triggername;**

Mutating error is a runtime error occurs in row level triggers for avoiding mutating error.Oracle provided packaged global variable method by using following 3 steps. These are...

Step 1: Create packaged global variable
Step 2: Create after row level trigger
Step 3: Create after statement level trigger



Example:

```
Sql> create table test1(col1 varchar2(20), col2 number(10));
Sql> create table test2(col1 number(10), col2 date);

Sql> create or replace trigger tr1
 after insert on test1
 for each row
declare
 id number(10);
begin
 select col2 into id from test1 where col2 = :new.col2;
 insert into test2 values (id, sysdate);
end;
/
Trigger created
```

Testing:

```
Sql> insert into test2 values('murali', 50);
Error: ORA-04091: Table is mutating
```

Solution:

Step 1:Create global variable (create a Packaged global variable for user inserted data)

```
sql> create or replace package pz1 is
 id number(10);
end;
/
```

Step 2: After row level trigger
(Using row level trigger collect value from user and then storethat value into global variable)

```
Sql> create or replace trigger tr1
 after insert on test1
 for each row
begin
 pz1.id := :new.col2;
end;
/
```

Step 3: After statement level trigger
(Using global variable value for solving task by using after statement level trigger)

```
Sql> create or replace trigger tr2
  after insert on test1
begin
  insert into test2.values (pz1.id, sysdate);
end;
/
```

Testing:

```
Sql> insert into test1 values('murali', 50);
1 row inserted
Sql> select* from test1;
Sql> select * from test2;
```

Note: In Oracle 11g, onwards we can also avoid mutating error by using compound trigger. Because compound trigger having global declarations section same like packages.

- **Write a PL/SQL Compound trigger in oracle 11G which is used to avoid mutating errors by using above table?**

Ans:

```
create or replace trigger tr1
  for insert on test1
  compound trigger
  id number(10);
  after each row is
  begin
    id := :new.col2;
  end after each row;
  after statement is
  begin
    insert into test2 values (id, sysdate);
  end after statement;
end;
/
```

- **Write a PL/SQL row level trigger on emp table whenever user inserting department then not allow to insert more than 4 employees in each department from emp table?**

Ans:

```
create or replace trigger tr1
  after insert or update or delete on emp
  for each row
declare
  v_count number(10);
begin
  select count(*) into v_count from emp where deptno = :new.deptno;
  if v_count > 4 then
    raise_application_error(-20123, 'We cannot insert more than 4 employees');
  end if;
end;
/
```

Testing:

```
sql> insert into emp (empno, deptno) values (1, 10);
Error: ORA-04091:Table SCOTT.EMP is mutating.
```

Solutions:

Step 1:Create global variable

```
create or replace package pz2 is
 g number(10);
end;
/
```

Step 2: Create after row level trigger

```
create or replace trigger tr1
 after insert or update or delete on emp
 for each row
begin
 pz2.g := :new.deptno;
end;
/
```

Step 3: Craete after statement level trigger

```
create or replace trigger tr2
 after insert or update or delete on emp
declare
v_count number(10)
begin
 select count(*) into v_count from emp where deptno=pz2.g;
 if v_count>4 then
   raise_application_error(-20123,'Not to store more than 4 employees');
   end if;
   end;
/
```

Testing:

```
Sqi> insert into emp (empno, deptno) values (1, 10);
Sql> insert into emp (empno, deptno) values (2, 10);
Error: ORA-20123: Not to store more than 4 employees.
```

Oracle 7.1 introduced Dynamic SQL. It is the combination of SQL, PL/SQL. In Dynamic SQL, SQL statements are executed at runtime.

Generally, in PL/SQL we are not allowed to use DDL, DCL statements. If we want to use those statements in PL/SQL then we must use Dynamic SQL constructor.

Generally, in Dynamic SQL, SQL statements must be specified within single quotes and also those statements are executed by using "execute immediate" clause. This "execute immediate" clause must be used in Executable Section of the PL/SQL block.

Syntax:

```
begin
 execute immediate 'SQL Statements';
end;
```

Example:

```
begin
 execute immediate 'create table students(sno number(10))';
end;
/
```

➢ **Write a PL/SQL program which is used to create an user defined "ROLE"? *****

Ans:

```
sql> conn sys as sysdba;
Enter Password: sys

Sql> begin
  execute immediate 'create role r1';
end;
/
```

➢ **Write a PL/SQL Stored Procedure for passing table as a parameter then drop all the indexes from the passed table? *****

Ans:

```
sql> create table emp1 as select * from emp;
sql> select * from emp1;
sql> desc user_indexes;
sql> select index_name, index_type from user_indexes where table_name = 'EMP1';

sql> create or replace procedure p1(p_table varchar2) is
  cusor c1 is select index_name from user_indexes where table_name = p_table;
begin
  for i in c1
loop
    execute immediate 'drop index || i.index_name;
  end loop;
end;
/
```

Execution:

```
Sql> exec p1('EMP1');
```

If we want to retrieve data from database by using Dynamic SQL statement then we are using "into" clause.

➢ **Write a Dynamic SQL program which is used to retrieve maximum salary from emp table and also display that salary?**

Ans:

```
declare
 v_sal number(10);
begin
 execute immediate 'select max(sal) from emp'
  into v_sal;
 dbms_output.put_line(v_sal);
end;
/
```

Output: 5400

Note: In dynamic SQL we can also use bulk collect clause which returns multiple values into collection.

➢ **Write a Dynamic SQL program which is used to retrieve all employees name from emp table and store it into index by table and also display content from index by table through bulk collect clause?**

Ans:

```
declare
 type t1 is table of varchar2(10)
 index by binary_integer;
 v_t t1;
begin
 execute immediate 'select ename from emp'
 bulk collect into v_t;
 for i in v_t.first .. v_t.last
 loop
  dbms_output.put_line(v_t(i));
 end loop;
end;
/
```

If we want to pass values into Dynamic SQL statement then we are using "**place holders**" in place of actual values. In Oracle place holders are represented by using colon (:) operator and also in these place holders we are passing values through "**using**" clause.

➢ **Write a Dynamic SQL program which is used to insert a record into dept table?**

Ans:

```
declare
 v_deptno number(10) := &deptno;
 v_dname  varchar2(10) := '&dname';
 v_loc   varchar2(10) := '&loc';
begin
 execute immediate 'insert into dept values(:1, :2 , :3 )'
   using v_deptno, v_dname, v_loc;
end;
/

Enter value for deptno :1
Enter value for dname :a
Enter value for loc :b
Sql > select * from dept;
```

Note: We can also use "**USING**", "**INTO**" clause in a single Dynamic SQL statement at a time, in this case always into clause precedes then "**USING**" clause.

➢ **Write a Dynamic SQL program for user entered deptno then display deptname, location from dept table by using into, using clauses?**

Ans:

```
declare
 v_deptno number(10) := &deptno;
 v_dname  varchar2(10);
 v_loc   varchar2(10);
begin
 execute immediate 'select dname,loc from dept where deptno =:1'
   into v_dname, v_loc
   using v_deptno;
 dbms_output.put_line (v_dname || ' ' || v_loc);
end;
/
```

Output:

```
Enter value for deptno: 10
ACCOUNTING NEWYORK
Sql> /
Enter value for deptno: 20
RESEARCH DALLAS
```

Oracle 8.0 introduced LOBS. These are predefined data types which are used to store large amount of data, images into database.

In Oracle, if we want to store more than 2000 bytes of alpha numeric data then we are using "varchar2" data type. Varchar2 data types stores upto 4000 bytes of alpha numeric data.

If we want to store more than 4000 bytes of alpha numeric data then we are using "LONG" data type, because Long data type stores upto 2GB data but there can be only one LONG column for a table and also we cannot create primary key on LONG data type column.
To overcome these problems Oracle 8.0 introduced "CLOB" datatype.

Syntax: <column name> long

Example:
Sql> create table test (col1 long, col2 long);
ERROR: a table may contain only one column of type LONG.
Sql> create table test(col long primary key);
ERROR: key column cannot be of LONG data type.

In Oracle if we want to store binary data or hexa decimal data then we are using RAW data type. It stores upto 2000 bytes of binary data.

Syntax: <column name> raw(size)

If we want to store more than 2000 bytes of binary data then we are using LONG RAW data type. It stores upto 2GB data but there can be only 1 long raw column for a table. To overcome this problem Oracle 8.0 introduced "BLOB" data type.

Syntax: <column name> long raw

Example:

```
sql> create table test(col1 raw(200));
sql> insert into test values('1010101');
1 row created

sql> insert into test values('ABCDEF');
1 row created

sql> insert into test values('ABCDEFG');
Error: invalid hex number

sql> create table test1(col1 long raw);
sql> desc test;
```

All database systems having 2 types of large objects. These are...

1) **Internal Large Objects.**
2) **External Large Objects.**

1. **Internal Large Objects:** Internal Large Objects are stored within database. Oracle having 2 types of Internal Large Objects.

   1) Character Large Object(CLOB)
      Syntax: <column name> clob

   2) Binary Large Object(BLOB)
      Syntax: <column name> blob

2. **External Large Objects:** External Large Objects are stored in outside of the database. i.e. these objects are stored in Operating System files. This is a "BFILE" data type.
   Syntax: <column name> bfile

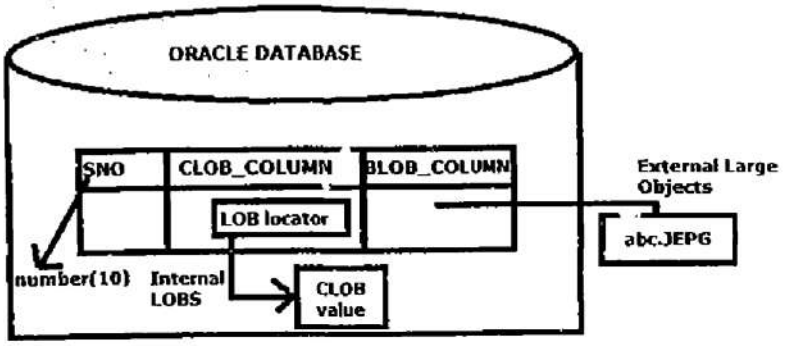
Difference between LONG, LOBS data type

LONG	LOBS
It can store up to 2GB data.	It can store up to 4GB data.
A table contains only one Long column.	A table contains more than one Lob columns.
Subquery cannot select a long data type.	Subquery can select LOB column.
Long data type does not work with regular expressions.	CLOB data type work with regular expressions.

LOB Locator

Generally whenever we are storing data into LOB columns then that data in LOB column not stored directly in table column same like other data type column in the row.

Instead of this one it stores a pointer that pointer points to where the actual data stored within the database. This pointer is also called as LOB locator. In case of external LOBS pointer points to the data within operating system file.



These functions are part of the SQL DML statements. These functions are used to initialize LOB locator into empty locator in Insert, Update statement.

Note: Before we are writing data into LOBS by using **"dbms_lob"** package (or) by using OCI (Oracle Call Interface) then we initialize LOB locator into empty locator by using **"empty_clob"** and **"empty_clob"** functions.

Whenever we are using these functions Oracle server automatically store other than **"null"** value into LOB locator and then only the LOB locator points to where the actual data is store within Oracle database.

Difference between Empty, Null

Example:

```
sql> create table test(sno number(10), col2 clob);
Sql> insert into test values(1, 'abc');
Sql> insert into test values(2, empty_clob);
Sql> insert into test values(3, null);

Sql> select * from test;
```

Sno	Col2
1	Abc
2	
3	

```
Sql> select * from test where col2 is not null;
```

Sno	Col2
1	Abc
2	

```
Sql> select col1, col2, length(col2) from test;
```

Sno	Col2	Length(Col2)
1	Abc	3
2		0
3		

Note: Whenever we are initializing LOB locator by using **empty_clob** (or) **empty_blob** functions then Oracle server automatically stores zero length string within LOB locator.

Step 1: Before we are storing data or image by using **dbms_lob** package then we must create alias directories related to physical directory by using following syntax.

Syntax: Create or replace directory <directory name> as 'path';

Note: Directory name ----> Capital letters.

Before we are creating alias directory then database administrator must use create any directory system privilege to user by using following syntax.

Syntax: grant create any directory to <user name>;

Example:

```
sql> conn sys as sysdba;
Enter password: sys
Sql> grant create any directory to scott;
Sql> conn scott/tiger;
Sql> create or replace directory XYZ as 'C:\';
Directory created
```

In Oracle if you want to view logical directory and also path of the directory then we are using "all_directories" data directories.

```
Sql>desc all_directories;
Sql> select directory_name,directory_path from all_directories;
```

DIRECTORY_NAME	DIRECTORY_PATH
XYZ	C:\

Step 2: <u>**Create a table in Oracle database by using LOB columns:**</u>

Example: sql> create table test(sno number(10), col2 clob);

Step 3: Develop a PL/SQL program to store large amount of data into CLOB column (or) image into BLOB column by using "dbms_lob" package.

Step A: In declare section of the PL/SQL block we must define LOB variables by using CLOB, BLOB, BFILE data types.

Syntax:

```
<Variable name1> clob;
<Variable name2> blob;
<Variable name3> bfile;
```

Step B: After declaring the variables in executable section of the PL/SQL block we must initialize "LOB locator" into empty by using "empty_clob()" (or) "empty_blob()" functions and also by using "returning into" clause we must store LOB locator value into appropriate LOB locator variable by using following syntax.

Syntax: Insert into tablename values (empty_clob()) returning lobcolname into lobvariablename;

Step C: Before we are writing data into LOB column then we must concatenate alias directory along with actual file name by using BFILE name function. This function accepts two parameters. These are alias directory name, filename.

Syntax: <bfile variable name> := bfilename('alias directory name', 'file name');

Step D: If we want to load data into LOB column by using "dbms_lob" package then we must use "loadfromfile" procedure from "dbms_lob" package. This procedure accepts 3 parameters.

Syntax: dbms_lob.loadfromfile (<lob variable name>, <bfile variable name>, <length of bfile>);

Note: If we want to find "length of the BFILE" then we are using "GETLENGTH()" from "dbms_lob" package.

Syntax: dbms_lob.getlength (bfilevarname);

Step E: Before we are loading data into LOB column just we must open the BFILE pointer by using "fileopen" procedure from "dbms_lob" package.

Syntax: dbms_lob.fileopen (bfile variable name);

Step F: After loading data into LOB column then we must close the file by using "fileclose" procedure from "dbms_lob" package.

Syntax: dbms_lob.fileclose (bfile variable name);

Example

```
sql> conn scott/tiger;
sql> create table test (sno number(10), col2 clob);

sql> declare
  v_clob  clob;
  v_bfile bfile;
begin
  insert into test values (1, empty_clob()) returning col2 into v_clob;
  v_bfile := bfilename ('XYZ', 'file1.txt');
  dbms_lob.fileopen (v_bfile);
  dbms_lob.loadfromfile (v_clob, v_bfiel, dbms_lob.getlength(v_bfile));
  dbms_lob.fileclose (v_bfile);
end;
/
```

Testing:

```
sql> select * from test;
```

SNO	COL2
1	welcome

Note: In Oracle, we can also store large amount of data or images into Oracle database by using "BFILE" datatype. When we are using "BFILE" name function within insert statement but this BFILE data or image cannot be displayed in oracle database.

Example:

```
sql> create table test(col1 bfile);
sql> insert into test values (bfilename ('XYZ', 'file1.txt'));
1 row created

sql> select * from test;
Error: SP2-0678: column or attribute type cannot be displayed by SQL*Plus.
```

Loading Large Amount of Data (or) Image into LOB Column by Using SQL Loader Tool *

If you want to store large amount of data or images into LOB columns by using SQL loader tool then we must use LOB file function within control file. This function accepts file name. This file name specifies location of the file and also LOB file functions store this file contain into LOB column.

Syntax: colname lobfile (file name);

Example

file1.txt

```
101, abc, 3000
102, xyz, 4000
103, pqr, 9000
```

fil2.txt

```
file1.txt
save as file2.txt

sql> create table test(file_data clob);
```

Control file

```
load data
infile 'C:\file2.txt'
insert
into table test
fields terminated by ','
(file_name filler char(100), file_data lobfile (file_name) terminated by EOF)

sql> select * from test;
```

FILE_DATA
101,abc,3000
102,xyz,4000
103,pqr,9000

- **Write a PL/SQL cursor program by using explicit cursor modifying salaries of the employees in emp table based on following conditions?**
  a. If job='CLERK' then increment sal --> 100
  b. If job='SALESMAN' then decrement sal --> 200

Ans:

```
declare
 cursor c1 is select * from emp;
 i emp%rowtype;
begin
 open c1;
 loop
  fetch c1 into i;
  exit when c1%notfound;
  if i.job = 'CLERK' then
   update emp set sal = sal + 100 where empno = i.empno;
  elsif i.job = 'SALESMAN' then
   update emp set sal = sal - 200 where empno = i.empno;
  end if;
 end loop;
 close c1;
end;
/
```

Output: select * from emp;

In all databases, whenever we are using DML operations then database servers automatically establish locks but if you want to perform locks prior to DML statements then we must use locking mechanism explicitly by using cursors. If you want to perform locks through cursor then we must use "FOR UPDATE" clause within cursor "SELECT" statement.

Syntax: cursor cursorname is select * from tablename where condition for update;

Whenever we are opening the cursor then only oracle server internally uses exclusive locks based on the "FOR UPDATE" clause.

Where Current of:

Where Current Of clause is used in update, delete lastly fetched row from the cursor. "**Where Current Of**" clause is uniquely identifying a record because "**Where Current Of**" clause internally using "**ROW ID**". "**Where Current Of**" clause used in **update, delete** statements only.

Note: In all database whenever we are using WHERE CURRENT OF clause then we must use "FOR UPDATE" clause within cursor SELECT statement otherwise Oracle server returns an error.

Syntax: update <table name> set <column name> = <new value> WHERE CURRENT OF <cursor name> ;
Syntax: Delete from <table name> WHERE CURRENT OF <cursor name>;

Note: WHERE CURRENT OF clause is used to update (or) delete lastly fetched row from the cursor. Whenever resource table does not have primary key then we are updating and deleting records from those tables by using cursors then we must use "WHERE CURRENT OF" clause. Whenever after processing data we are releasing locks by using "COMMIT" in PL/SQL blocks.

Generally whenever resource table column having duplicate data and also when we are using that duplicate data column in where condition of the **update, delete** statements in cursor then oracle server returns wrong results.

To overcome this problem in place of our own where condition then we must use "WHERE CURRENT OF" clause because "WHERE CURRENT OF" clause internally uses rowed.

Note: After processing we are releasing locks by using commit or rollback commands within cursor program.

➢ **Write PL/SQL cursor program to modify salaries of the employees in the following table?**

-> Increment sal in each record by 1000

Ans:

```
sql> create table test (ename varchar2(20), sal number(10));
sql> insert into test values('&ename', &sal);
sql> select * from test;
```

ENAME	SAL
A	1000
B	2000
A	3000
B	4000
C	5000

Without using "WHERE CURRENT OF" clause.

```
declare
 cursor c1 is select * from test;
begin
 for i in c1
 loop
  update test set sal = sal + 1000 where ename = i.ename;
 end loop;
end;
/
```

Output:

```
sql> select * from test;
```

ENAME	SAL
A	3000
B	4000
A	5000
B	6000
C	6000

```
sql> rollback;
```

Solutions
With Using "WHERE CURRENT OF" clause.

```
declare
 cursor c1 is select * from test for update;
begin
 for i in c1
 loop
  update test set sal = sal + 100 where current of c1;
 end loop;
 commit;
end;
/
```

Testing: sql> select * from test;

ENAME	SAL
A	2000
B	3000
A	4000
B	5000
C	6000

This is correct result.

- **Write a PL/SQL explicit cursor program which is used to increment salary of the 5th record from emp table by using explicit cursor locking mechanism?**

Ans:

```
declare
 cursor c1 is select * from emp for update;
begin
 for i in c1 loop
  if c1%rowcount = 5 then
   update emp set sal = sal + 100 where current of c1;
  end if;
 end loop;
 commit;
end;
/
```

- **Write a PL/SQL program which is used to delete 10th record from emp table by using explicit cursor locking mechanism?**

Ans:

```
declare
 cursor c1 is select * from emp for update;
begin
 for i in c1 loop
  if c1%rowcount = 10 then
   delete from emp where current of c1;
  end if;
 end loop;
 commit;
end;
/
```

```
sql> create table test (sno varchar2(20) primary key, name varchar2(10));
sql> create sequence s1;

sql> create or replace trigger tr
 before insert on test
 for each row
begin
 select 'ABC' || lpad(s1.nextval, 10, '0') into :new.sno from dual;
end;
/

sql> insert into test(name) values('&name');
Enter value for name: abc
sql>/
Enter value for name: xyz
sql>/
Enter value for name: pqr

sql> select * from test;
```

SNO	NAME
ABC0000000001	Abc
ABC0000000002	Xyz
ABC0000000003	Pqr

SQL Object Type

Oracle 8.0 introduced object technology in SQL language.Object is a user defined type which contains predefined data type, it is also same as structure in C language.

Syntax: Create or replace type typename as object (attribute1 datatype(size), attribute2 datatype(size),.....);

Once we are creating Object type then we are using that object type in PL/SQL by using following 2 steps.

Step 1: Instantiating an Object: In declare section of the PL/SQL block we are instantiating an object by using following syntax.

Syntax: Variablename objecttypename;

Step 2: Assign values into Object: In executable section of the PL/SQL block we are assigning value into object by using following syntax.

Syntax: Variablename := objectname (value1, value2,.....);

Example:

```
sql> create or replace type student as object stno number(10), sname varchar2(10));
/
```

Instantiating an Object:

```
declare
 st1 student;
begin
 st1 := student(101, 'shailendra');
 dbms_output.put_line('student id is :' || ' ' || st1.sno);
 dbms_output.put_line('student name is :' || ' ' || st1.name);
end;
/
```

Output:

```
Student id is: 101
Student name is: shailendra
```

PL/SQL Object Type

PL/SQL object type is also same as class in object oriented languages. PL/SQL objects type having 2 parts.

1) Data
2) Members subprogram (These member subprogram access data within object type)

If we want to define member subprograms then we must use type body within object type.
Object type having two parts.

1) Object Specification
2) Object Body

In Object specification we must declare member data, member subprograms. In object body we are implementing those member subprograms. Once we are implementing member's subprograms then only we are calling those subprograms through object instance.

Object Specification:
Syntax:

```
create or replace type typename as object
(
 attributename datatype(size),
 member procedure declaration,
 member function declaration
);
```

Object Body:
Syntax:

```
create or replace type body typename
as
member procedure implementation;
member function declaration;
end;
/
```

Example

```
sql>
create or replace type circle as object
(
 r number,
 member procedure p1,
 member function f1 return number
);
/

sql> create or replace type body circle as
 member procedure p1 is
 begin
  dbms_output.put_line('proc');
 end p1;
 member function f1 return number is
 begin
  return 2 * 3.14 * r;
 end f1;
end;
/
```

Execution: (Instantiating an Object)

```
declare
 c circle;
begin
 c := circle(4);
 dbms_output.put_line(obj1.f1);
end;
/
```

Output: 25.12

1) **Oracle 12C introduced Invisible Columns:**

Oracle 12C, introduced invisible columns either at the time of table creation or after table creation using alter command.

Syntax: Columnname datatype(size) invisible
Syntax: alter table tablename modify(columnname invisisble);

If you want to view invisible column then we are using following syntax at SQL prompt.

Syntax: sql> set colinvisible on;

Example: 12C

```
sql> create table test(sno number(10), name varchar2(10));
table created
sql> insert into test values(50);
Error: Insufficient values inserted.
sql> alter table test modify (name invisible);
sql> insert into test values(50);
1 row inserted
sql> select * from test;
```

SNO
50

```
sql> set colinvisible on;
sql> select * from test;
```

SNO	NAME
1	

2) **Oracle 12C introduced new-auto increment concept:**

Generating primary key value automatically is also called as auto increment concept. Prior to Oracle 12C if we want to generate primary key values automatically then we are using Sequences, Row level trigger. Whereas in Oracle 12C we can also implement auto increment concept without using row level trigger by using following 2 methods. These are...

1) **With using Sequences**
2) **Without using Sequences. (Identity column)**

With Using Sequences:

if we want to generate primary key value automatically then we are using "**default**" clause along with sequence PSEUDO columns within primary key.

Syntax: columnname datatype(size) default sequencename.nextval primarykey

Example: 12C

```
sql> create sequence s1
     start with 5;
sql> create table test(sno number(10)) default s1.nextval primary key, name varchar2(10);
sql> insert into test(name) values('abc');
sql> insert into test(name) values('xyz');
sql> select * from test;
```

SNO	NAME
5	abc
6	xyz

Without using Sequences: (Identity column)

In this method Oracle server only internally automatically create a sequence when we are using following clause but by default this sequence values are always starts with 1 and also automatically incremented by 1. these columns are also called as identity column. These clauses are...

1) **Generated always as identity**
2) **Generated by default as Identity**
3) **Generated by default on null as identity**

Example: (Generated always as identity)

Whenever we are using always as clause then we are not allow to insert our own value into identity column.

```
sql> create table test(sno number(10) generated always as identity, name varchar2(10));
sql> insert into test(name) values('x');
sql> insert into test(name) values('y');
sql> select * from test;
```

SNO	NAME
1	X
2	Y

```
sql> insert into test(sno,name) values (8, 'z');
Error:
```

3) **Oracle 12C introduced new Top-N analysis:**

Prior to Oracle 12C if you want to implement top-n- analysis then we are using **inline view, rownum** where as Oracle 12C introduced new top-n-analysis by using fetch/next clauses.

Syntax: select * from tablename order by columnname [asc/desc] offset n rows/fetch first/next n rows only;

- **Write a query to display first 5 highest salary employees from emp table by using fetch first clauses?**

Ans: sql> select * from emp order by sal desc fetch first 5 rows only;

- **Write a query to display except first 3 rows then display next onwards 5 rows only from emp table?**

Ans: sql> select * from emp order by sal desc offset 3 rows fetch first 5 rows only;

- **Write a query to display first 50% highest salary employees from emp table by using fetch first clause?**

Ans: sql> select * from emp order by sal desc fetch first 50 percent rows only;

Restrictions

1) When a select query having for update clause then we are not allow to use fetch first clause.
2) When a select statement having sequence PSEUDO column then we are not allow to use fetch first clauses.
3) In incremental refresh materialized view we are not allow to use fetch first clauses.

4. **Multiple indexes on table column:**

Prior to Oracle 12C we cannot create multiple indexes on same column , whereas Oracle 12C introduced multiple indexes on same column by using invisible clause.

Syntax: create index indexname on tablename(columnname) invisible;

Example:

```
sql> create index in1 on emp(ename);
indx created
sql> create bitmap index in2 on emp(ename);
Error: such column list already indexed.
```

Solution: 12C

```
sql> create bitmap index in2 on emp(ename)invisible;
index created
```

5. **12C introduced Truncate table cascade:**

Prior to Oracle 12C, when we are truncating master table if child table records are exists within child table then oracle server returns an error.

Error: unique/ primary keys in table references by Enabled Foreign keys.

If child table contains within "ON DELETE" cascade (or) without "ON DELETE" cascade also.
To overcome this problem Oracle 12C introduced "truncate table cascade" clause which truncates master table records automatically if child table records exists also. But in this case child table must contain "ON DELETE cascade" clause.

Syntax: truncate table tablename cascade;

Example:

```
sql> create table mas(sno number(10) primary key);
Sql> insert into mas values(&sno);
Sql> select * from mas;
SNO
-----
1
2
Sql> create table child(sno number(10) references mas(sno) on delete cascade);
Sql> insert into child values(&sno);
Sql> select * from child;
SNO
-----
1
1
2
2
1
```

```
Sql> truncate table mas cascade;
Sql> select * from mas;
Sql> select * from child;
```

6. **12C introduced default on null clause in a table column**

7. **12C introduced accessible by clause in store procedure**

Oracle 12C, introduced "Accessesible By" clauses within "Stored sub programs" which provide more security for the accessing sub programs. Accessible by clause used in sub program specification only.

Syntax:

```
create or replace procedure procedurename(formal parameters)
accessesible by(another procedure name)
is/as
begin
-------
[exeception]
-------
end[procedurename];
```

Example:

```
sql> create or replace procedure p1
is/as
begin
p2;
dbms_output.put_line('first proc');
end;
/
sql> create or replace procedure p2
accessesible by(p1)
is
begin
dbms_output.put_line('second proc');
end;
/
```

8. 12C introduced WITH clause in PL/SQL local function:

This function directly used in select statement.

Oracle 12C, introduced "WITH" clause functions and those functions are immediately calling in **SELECT** statement. These used defined functions are "Local functions".

Prior to Oracle 12C, Oracle 9i introduced "WITH" clauses in Top-N analysis. Those "WITH" clauses temporary tables are used in SELECT statement.

Syntax:

```
with function functionname(formal parameters)
return datatype
is / as
-----------
-----------
begin
----------
----------
return expr;
end;
select functionname(actual parameters) from dual;
/
```

Note: With clause also improves performance of the local function.

Example: 12C

```
with function f1(a number)
return number
is
begin
return a*a;
end;
select f1(4) from dual;
/
```

Output:16

1) Oracle 11G introduced read-only tables:

Oracle 11G introduced read only tables by using "alter" command. In these tables we cannot perform DML operations.

```
Syntax: alter table tablename read only;
Syntax: alter table tablename read write;
```

2) Oracle 11G introduced Simple_integer data type:

Oracle 11G, introduced "Simple_integer" data type in PL/SQL which does not accepts null values because Simple_integer data type internally having "not null" clause. That's why we must assign the value at the time of variable declaration in declaring section of the PL/SQL block.

```
Syntax: variablename simple_integer:= value;
```

Example:

```
declare
 a simple_integer := 50;
begin
 dbms_output.put_line(a);
end;
/
```

Output: 50

Note: Simple_integer data type performance is very high compare to "pls_integer" data type.

Example:

```
declare
 a pls_integer;
begin
 a := 90;
 dbms_output.put_line(a);
end;
/
```

Output: 90

Difference between pls_integer, binary_integer,simple_integer data type

a) **Pls_Integer:-** Pls_integer data type stores data in hardware arthmatic format so it is faster than number data type and also pls_integer data type required less storage.

Note: Use pls_integer data type when more calculations are in used.

b) **Binary_integer:-** Binary_integer data is used for declaring signed integer variable. Binary_integer variables stores data in binary format that's why it takes less space.

Calculation on binary integer can run slightly faster because the values are already in binary format.

Generally binary_integer data type is used in declaring index field for index by table.

c) **Simple_integer:-** It is introduced in 11G simple_integer data type give more performance ..... this data type internally having not null.

3) Oracle 11G introduced Pivot function:

Oracle 11G, introduced pivot() function which is used to display aggregate values in tabular format and also rows are converted into columns. Prior to Oracle 11G if we want to achieve pivoting reports then we are using "decode ()" within "group by" clause. But Pivot() function performance is very high compare to decode().

Syntax:

```
select * from (select col1, col2,..... from tablename)
pivot (aggregate functionname (colname)
for columnname in(value1, value2.....));
```

Example:

```
select * from (select job, deptno, sal from emp)
pivot(sum(sal)
for deptno in (10 as deptno10, 20 as deptno20, 30 as deptno30));
```

Output:

JOB	DEPTNO10	DEPTNO20	DEPTNO30
CLERK	3000	6200	2650
SALESMAN			6300
PRESIDENT	5400		
MANAGER	2850	3375	3250
ANALYST		6650	

4. Oracle 11G introduced Continue statement in PL/SQL loop:

Oracle 11G, introduced "**Continue**" statement within PL/SQL loops. It is also same as "C" language Continue statement.

Example:

```
begin
 for i in 1 .. 10
  loop
  if i = 5 then
   continue;
  end if;
  dbms_output.put_line(i);
 end loop;
end;
/
```

Output:

1	2	3	4	6	7	8	9	10

Whenever we are using "Continue" statement automatically control sends to the beginning of the loop.

5. Oracle 11G introduced "reg exp_count()".

6. Oracle 11G introduced "Compound Triggers".

7. Oracle 11G introduced "FOLLOWS" clause in triggers.

8. **Oracle 11G introduced "Variable Assignment" concept when we are using sequences in PL/SQL block.**

   Oracle 11G, introduced "Variable" assignment concept when we are using sequences in PL/SQL block. In this case we are not allowed to use Dual table.

Syntax:

```
begin
 varname := sequencename.nextval;
end;
/
```

9. **Oracle 11G introduced Enable (or) Disable clause in trigger specification:**

   Oracle 11G introduced Enable (or) Disable clauses in trigger specification itself.

Syntax:

```
create or replace trigger triggername
befor/after insert/update/delete on tablename
[for each row]
[when condition]
[enable/disable]
[declare]
begin
-------
-------
end;
/
```

10. Oracle 11G, introduced "**Named**", "**Mixed**" notations when a subprogram is executed by using "SELECT" statement.