

Basic Introduction of Oracle

1. **Types of DataBases :**

1. FDBMS
2. NDBMS
3. RDBMS

2. **Data Types**

3. **Types of SQL Statements :**

1. **DDL(Data Definition Language) :**

1. create
2. alter
3. drop

2. **DML(Data Manipulation Language) :**

1. insert
2. delete
3. update
4. select
5. desc

3. **DCL(Data Control Language) :**

1. grant

4. **Operators :**

1. Arithmetic
2. Relational
3. Logical
4. Like
5. In
6. Between

5. **Functions :**

1. **Grouping Functions :**

1. min
2. max

3. avg
4. Standard deviation
5. variance
6. stddev

2. Character Functions :

1. ascii()
2. chr()
3. lower()
4. upper()
5. concat()
6. substr()
7. initcap()
8. length()

3. Member Functions :

1. abs()
2. power()
3. mod()
4. sqrt()

4. Mathematical Functions :

1. log()
2. sin()
3. cos()
4. tan()

5. Date Functions :

1. add_months()
2. last_day()
3. next_day()
4. months_between()

6. Set Commands

7. Show Commands

8. Constraints in Oracle :

1. unique
2. not null
3. primary key
4. null

9. Constants in Oracle :

1. sysdate
2. user
3. uid
4. rowid
5. distinct

10. Clauses in sql :

1. where
2. order by
3. group by
4. having
5. starting with

11. Table Set Commands :

1. union
2. union all
3. intersect
4. minus

12. VIEWS :

13. Joins :

1. self join
2. equi join
3. non-equi join

**14. Executing Multiple Queries from a file to sql
* plus editor**

15. Bind Variables

PL/SQL

1. Conditional Statements :

1. if
2. if-else

2. Control Statements :

1. while
2. do-while
3. for loop

3. Data Types in pl/sql :

1. number
2. varchar
3. varchar2
4. date

4. Structure of the pl/sql Program

5. Procedures & Functions

6. Triggers

ORACLE

DataBase:

1. Storing information about particular things. Using this database we can create the database

and we can store the information and we can access the information and we can modify the database

2. Using this database we can create any type of data & we can store the data in the form of information. We have different types of databases, we can create.

Ex: If we want to create a database regarding employees, we

can create the emp database.

emp database describes complete details of emp using that information, we can complete our tasks in time Using this database , we can modified the data & we can add &delete the data.

We can perform when we want to the modification before that we must contain details of existing database information

Data : Data is nothing but Collection of raw facts , using this data we can't get complete structure of the data.

Ex: Ramu , Ashok , 70 and etc.,

Information :

Information is nothing but giving the meaning full data about particular thing . Using this information we can decide whether this information set to our database or not .

Types of DataBases :

1. FDBMS
2. NDBMS
3. RDBMS

File Management DBMS :

1. In earlier days we have no systems at this time we did use File Management System.
2. Using this File Management System entire data managed by file.

3. Files contains Collection of Pages with database details using this File database, we can get entire information through Files.
4. Come to the modifications very difficulty.
5. For accessing the old record is difficulty.
6. For deleting the old record is difficulty.
7. Using this File Management System , Files will be waste for store in a data.
8. If we get the small modification we have to re-enter the data again.
9. For clear this drawbacks we design a database through Computer.
10. Using with computer database entire data storing in Hard-disk instead of files.
11. For storing the database we need a tool, that tool is Oracle.
12. Oracle is one type of database Using this Oracle we can create our own database.
13. What we created the database , we can perform all operations on this data base.
14. Using this database we can clear all the problems in File Management System.
15. Using this Oracle database we can store data permanently and modifications easily and we can access the data any time, any place.

NetWork DBMS :

1. Using this network DBMS we can connect with the database through NetWork.
2. In network we have no. of systems , every system connect with the main server

3. Main server contains the database , rest of the systems does not contain database , It can access when ever it needs the main database data keep in server database.
4. Server may be connect with only one system or thousands of systems.
5. Our system requests to the server , server will response and send back results to the system, system will display that output.
6. The entire data processing server only.

Relational DataBase Management System :

1. Relational DataBase Management System representation of tables only.
2. In RDBMS we can store the data in the form of tables.
3. Using this RDBMS we can create database easily.
4. We can insert the data easily.
5. We can modify the data easily.
6. Using the RDBMS we can perform all operations on the table .
7. One table contains columns.
8. One table may contains 1 column or no. of columns .
9. One table must contain 1 column , without column we can't create table .
10. Which database we are going to be create regarding that we can create the table .
11. The table name must matched to our database name.
12. The database may be emp, student .Cell shop etc.,

13. In columns we will write the attributes of the databases.

- **Table definition :** Collection of records, record is nothing but collection of columns of attributes.
- Using this RDBMS entire data we keep it in the form of tables.
- Table contains the no. of records , every record displays particular database details. One row means one record.
- RDBMS performing one language that is called SQL.

SQL (Structured Query Language) :

1. Using this sql, we can create the tables , we can modify the tables, we can drop the tables.
2. In sql we have no. of queries for access the database we can apply the queries on the tables & we get the results.
3. **E.F.CODD** -----sql founder

Query :

- Query is nothing but condition or question to the database.
- This Query required by the database & execute the query in database. After executing the query send result back to the client.
- Ex: select empno from emp;
- Using this sql we can apply no. of queries on the table.

SQL * PLUS :

- sql*plus is a editor using this editor we can perform the all operations on table.
- This editor helps for doing the operations are entire RDBMS. In entire RDBMS entire data should be in the form of tables
- Using this editor we can write the sql queries & execute the queries , this editor displays the result of the query.
- Using this sql * plus editor not only for SQL ,It can also used for writing the PL/SQL programs.

PL/SQL :

1. PL/SQL is nothing but Procedure Language Structured Query Language , Using this pl/sql language we can write the programs.
 2. Using this programs we can create the database & perform the database actions .
 3. In pl/sql programs executes with in the help of sql * plus editor.
 4. Using the pl/sql we can write programs and this programs results storing in table.
 5. ***PL/SQL is a programming language, it contains data types***
 1. number
 2. varchar
 3. varchar2
 4. date
- In sql also contains data types , using the data types we can create the tables .
 - After giving the data types , to the columns.

- After giving the data types , to the particular column then onwards that column doesn't accept another type of data.
- If we want to store another type of data it shows that error.

Ex:

-

```
empno number //declare
empno='Ashok'; //wrong
```

- data types only applicable for column names
- Table name never contains the data type.

Data Types :

1. number
2. varchar
3. varchar2
4. date

1. **number :**

1. Using this number data type , we can store numbers in a particular column in table.
2. If we create one column of number data type, then that column allows only numbers, In that column it doesn't allow characters or any decimal points.
3. **Ex:** empno number ;
We can give number data type we want to store decimal values then we can use this number data

type , when we want to store decimal points then we have to give the range. In that range we give the decimal values.

4. **Ex:** empno number(5, 2) ;

2. **varchar :**

1. Using this data type we can store characters and strings in a particular column
2. If we want to store string values then go for this data types.
3. **Ex:** ename varchar ;
4. In varchar data type one more facility is we can declare the number of characters we want to store , we can give the size of the characters in braces.
Ex: ename varchar(20);

3. **varchar2**

1. Using this data type also we can store characters and strings.
2. Using this data type it will save the memory.
3. **Ex:** ename varchar2(20) ;
ename varchar2 ;

Difference between varchar & varchar2 :

1. **varchar:** Using this varchar data type when we give the size of the column value that number of bytes occupied by the for that column.
If we declare size is 20 , but if we use only 10 characters then rest of the 10 characters memory will be waste.
This memory not useful for another variable in memory.

2. **varchar2** : In this varchar2 datatype when we declare one variable with the size of a particular column then if we use that size completely then no problem. If we don't use the complete size of the declare time. Then rest of the memory useful to another variable or another program.

Date :

1. Using this date data type we can give the dates do a particular columns.
2. We can store the dates in columns as a value dates we can declare in the format of dd-mm-yy. Ex:

```
joindate date ;
joindate '10-Aug-95' ;
```
3. Always we have to declare characters & dates in the form of single quote.

Clob data type (Character Large Object data) :

- Using this data type we can store characters in the form of bytes , we can store group of characters , range is 2 GB.

Blob data type(Binary Large Object data) :

- When we want to store particular images and files in the form of bytes, range is 4 GB .
- We can store files and images in a table columns in the form of bytes .
- Actually we don't store the files and images into a particular column.
- We store files and images in a particular place in a memory from that place we can create one pointer

using this pointer we can store in a column values that pointer is called "Locator".

For Clob & Blob data types we can find the data with the help of locators.

Ex:

image blob ; //4GB

file clob ; //2GB

TYPES OF SQL STATEMENTS :

1. Data Definition Language----->
create, alter, drop
2. Data Manipulation Language----->
> insert, delete, update, select, desc
3. Data Control Language----->
grant, revoke

DDL commands:

1. **Create :**
syn: create table tablename(column datatype);
Ex: create table student(sno number, sname varchar2(20), dob date);
2. **Alter: -----add, modify, rename, drop**
Syn:- alter table tablename add columnname datatype;
Ex:- alter table emp add phno number;
Syn:- alter table tablename modify columnname datatype;
Ex:- alter table emp modify sname varchar2(30);

Syn:- alter table tablename rename column ocn to ncn ;

Ex:- alter table emp rename column sno to studno;

Syn:-alter table tablename drop column columnname;

Ex:- alter table student drop column sno;

3. Drop:

Syn: drop table tablename;

Ex: drop table emp;

DML commands:

1. **Insert :** **Syn:-**insert into tablename values(columnvalue);

Ex:-insert into student values(1,'apec');

Syn2:- insert into student values(&no,'&sname');

2. Delete :

Syn: delete tablename where column=value;

Ex:- delete emp where empno=10;

3. Update:

Syn:-update tablename set columnname=value where condition;

Ex:-update student set sname='apec' where sno=1;

4. Select:

Syn:- select columnname from tablename;

Ex:- select * from student;

5. Desc:

Syn:-desc tablename;

Ex:- desc student;

DCL commands:

1. Grant:

Syn:-grant permissions on tablename to username;

Ex:- grant alter on student to scott;

Ex:- grant update on emp to bank;

Ex:- grant alter,update on student to dsnr;

Ex:- grant any commands on student to scott;

DDL COMMANDS :

create :

1. create is a keyword.
2. It is using for creating one table in a database, after executing create command table will be created. The table contains zero records and structure with zero values.
3. Table contains number of columns , in create query we can give complete details of table.
4. In create command we give the column names with data types.
5. Syntax : create table tablename(columnname datatype);
Ex1: create table employee(empno number , dob date, sal number);
Ex2: create table bank(bname varchar2(30) , bal number);
Ex3: create table cell(cname varchar2(30) , cost

number(9,2));

Ex4: create table school(sname varchar2(20) , splace
varchar2(30));

Ex5: create table college(cid number , cname
varchar2(30));

Ex5: create table course(cid number , cname
varchar2(30));

delete :

1. delete is a key word , using this keyword we can delete particular row in a database table .
2. Delete command using for delete rows only , but not table.
3. Before executing the delete command that table name must be exist.If we give the unknown table name we will get an error
4. **syntax :** delete tablename where columnname=value;
Ex : delete employee where empno=10 ;

empno	empname	sal	commission
10	a	1000	10
20	b	2000	20
30	c	3000	30
40	d	4000	50

5. After executing the above delete command empno=10 that record will be deleted, rest of the rows are still executing in the table.

empno	empname	sal	commission
-------	---------	-----	------------

20	b	2000	20
30	c	3000	30
40	d	4000	50

6.

Ex 2 : delete employee where empno=20 ;

empno	empname	sal	commission
10	a	1000	10
20	b	2000	20
20	b	2000	40
30	c	3000	30
40	d	4000	50

7. After executing the above delete command empno=20 , this condition is satisfied 2 records , so oracle will delete 2 records from the emp table.

empno	empname	sal	commission
10	a	1000	10
30	c	3000	30
40	d	4000	50

8. After executing delete query one message will be display , the message is " 2 rows deleted".

Ex: delete employee where empno=50 ;

empno	empname	sal	commission
--------------	----------------	------------	-------------------

10	a	1000	10
20	b	2000	20
30	c	3000	30
40	d	4000	50

9. sql>no rows deleted

Ex: delete employee22 where empno=10 ;

sql> table or view doesn't exist.

Note : employee22 table not exists in database.

drop :

1. Using this dropcommand we can drop the table from database. If we can delete entire table from the database then we should go for drop command.
2. After executing the drop command , then table is permanently deleted from database entire table is deleted at a time.
3. **syntax : drop table tablename ;**
Ex1 : drop table employee ;
Ex2 : drop table student ;
Ex3 : drop table school ;
Ex4 : drop table bank ;
Ex:

empno	empname	sal	commission
10	a	1000	10
20	b	2000	20
20	b	2000	40
30	c	3000	30

40	d	4000	50
----	---	------	----

4. sql > drop table employee ;
sql > table is dropped ;

DML commands :

insert :

1. Insert is a keyword , this command is using for inserting values in a table
2. After creating the table we insert the values record by record .and using this command we insert any number of rows in a table.
3. Once we execute insert command one row is created.
4. Insert command must be use after table is created, otherwise it will show error on the sql prompt.
5. syntax : insert into tablename values(column1value, column2value , , columnNvalue);

Ex1: insert into employee
values(10,'ashok',1000,10) ;

sql > select * from employee ;

empno	empname	salary	commission
10	ashok	1000	10

6. sql > one row is created ;
7. **Ex2:** insert into employee values(&empno, &empname, &salary, &commission) ;
For insert multiple records at a time we no need to write this query every time.

empno	empname	salary	commission
10	a	1000	10
20	b	2000	20
20	b	2000	40
30	c	3000	30
40	d	4000	50

alter :

- Using this alter command we can modified the table and we can perform number of operations on the table.
 1. Add the new columns to the table
 2. Modify the existing data types
 3. Rename the existing column name
 4. Drop the column in a table

Add new columns to the table :

1. Using this alter command we can modify the table ,
First we create the table with 3 columns then if we want to add one more column into our table then using this alter command. then our table contains 4 columns
2. **syntax : alter table tablename add columnname datatype ;**
 Ex1 : alter table student add saddress varchar2(20) ;
 Ex2 : alter table employee add empjob varchar2(20) ;
 Ex3 : alter table bank add bname varchar2(20) ;
 Ex4 : alter table cell add cno number ;

Ex5 : alter table college add ejoin date ;

Example :

empno	empname	salary
10	a	1000
20	b	2000
20	b	2000
30	c	3000
40	d	4000

3. **Ex2 :** alter table employee add empjob
varchar2(20) ;

empno	empname	salary	empjob
10	a	1000	Security guard
20	b	2000	service man
20	b	2000	employ
30	c	3000	Team Leader
40	d	4000	HR

Modify the existing data types :

1. The 2nd function of alter command is modify , it is used for change the datatypes of the columns.
2. if we create one table with 3 columns with 3 datatypes of some pecified sizes
3. After createing the table if we want to increase or decrease of the column value size , After executing this query will changed the columns of the size .
4. **syntax :** alter table tablename modify columnname datatype;

Ex:create table student(sno number,sname
varchar2(20),dob date);

Ex : alter table student modify sname varchar2(40);
sql > table altered

Using this modify can changed datatype size is 20 to
40 size of sname in student table.

Rename the existing column name :

1. The one more function of alter is rename , rename is a keyword.Using this keyword we can rename the column names.
2. If we create one table with 3 columns with values ,
After existing the table if we don't satisfy with the
existing column names then we can rename the
column names .
3. Rename is nothing but deleting the old name and
giving the new name, after existing the rename
command new column name appear instead of old
column name .
4. **syntax :** alter table tablename rename column
oldcolumnname to newcolumnname ;

Ex : alter table student rename column empno to
employeeeno ;

sql > select * from employee ;

employeeeno	empname	salary	empjob
10	a	1000	Security guard
20	b	2000	service man
20	b	2000	employ
30	c	3000	Team Leader

40	d	4000	HR
----	---	------	----

Drop the column in a table :

1. The drop command is using for deleting a particular column in a table . If we create a one table with 3 columns in that if we want to delete any column then go for this command
2. After executing the drop command the column will be permanently deleted from the table .
3. After dropping the column corresponding values in that column will be deleted.
4. syntax : alter table tablename drop column
columnname ;
Ex : alter table student drop column sno ;
'sno' column will be deleting from student table with values.

Update :

1. This command is using for doing the modifications on the column values.
2. If we create one table with 4 columns then we can insert the values., after inserting the values if we want to do any modifications in the column values then go for this command.
3. Using this command perform 2 operations.
 1. **Replacing the new value with old value.**
 2. **Giving the values to the new columns .**
4. When we add the new columns to the table when that new columns does not contains values.

5. If we want to send the values for the new column then go for update.
6. **Syntax : update tablename set columnname=value where condition ;**
Ex1 : update employee set empname='ashok' where empno=1 ;
Ex2 : update student set sname='jobs' where sno=2;

Select :

1. This command is using for display the table with values., this command is using for seeing the information of table.
2. This command is used for display the no. of records reliable in a table.
3. First we create the table then we insert the values.
4. After inserting the values we get the message like one row is inserted.
5. If we want to check that row and we want see that values then go for select command
6. This is non-action statement.
7. After executing the select query it doesn't effect on database.
8. After executing this query no modifications in the table .
9. Using this select command we can see entire table at once.
10. Using this select command we can select not entire table , we can select particular no. of columns also.

11. When we use the select command we can access one column or more column or entire table we can access.

12. **syntax :_select * from tablename ;**

* means displaying the all column names with values in a table.

Ex:_select sno, sname from student ;

Desc :

1. Using this command we can see the structure of the table , Structure is nothing but how many columns available in table and it display the data types also
2. desc will display the result in the form of table. tha table contains number of columns with data types.

3. **syntax : desc tablename ;**

Ex : desc employee ;

Ex : desc student ;

DCL(Data Control Language):

- Using with DCL we can perform to operations those are grant and revoke.

Grant :

1. It is uesd for giving the permissions to the users in Oracle , we have different types users.
2. Every user has create table their own databases .
3. Every user can give the permissions on their created table .
4. We can give the permissions through access one user to another user .

5. Without permissions another user doesn't access the table.
6. This command is used only giving the permission to the tables or users .
7. **syntax : grant permissions on tablename to username ;**
Ex : grant alter on student to scott ;
Ex : grant update on emp to apec ;
Ex : grant alter, update on bank to dsnr ;

Operators :

1. Arithmetic Operators : (+, -, *, /, %)

Using this operators we can performs on the tables .

1. **Ex :** select esal from emp where sal=sal+1 ;

2. Relational Operators :(< , > , >= , <= , = , !=)

Using this operators we can perform operations on the table.

1. **Ex :** select esal from emp where sal=100 and eno=1 ;

Ex1 : select esal from emp where eno=2 or eno=1 ;

Ex2 : select esal from emp where ename='ashok' or eno=1 ;

Ex3 : select esal from emp where sal <=100 ;

Ex4 : select esal from emp where sal =100 ;

Ex5 : select esal from emp where eno!=10 ;

Ex6 : select esal from emp where sal <> 100 ; // not equals to

Ex7 : select esal from emp where sal ~=100 ;

//tild symbol

3. Logical Operators : (**and , or**)

Using this log operators we can perform the Operations on the table .

Ex1 : select esal from emp where eno=10 and sal=1000 ;

Ex2 : select esal from emp where eno=10 or sal=1000 ;

4. Like Operator :

1. Like is a operator , this operator is used for finding the string into the table.
2. If we want find the name in Oracle table we can use this operator.
3. Like operators contain using with to operation we can perform search the strings.
4. Those operators are **% , _**
5. **%** :
 - This operator is used for when we don't know about the complete name when use this operator.
 - **% ---** indicates number of characters after writing the characters
 - We can given one or more characters also
 - ***syntax*** : select * from emp where ename like a% ;

empno	ename	salary	empjob
10	amp	1000	Security guard
20	ashok	2000	service man

6. **_** :

- underscore(_) search for single characters , when we give the underscore when we find only one character
- We our name contains 5 characters then we have give the 5 underscores.
- Which string we want to search that searching letter write first beside with underscores.
- ***syntax*** : select * from emp where ename like a---- ;

empno	ename	salary	empjob
10	aksha	1000	Security guard
20	ashok	2000	service man

5. In Operator :

- Using with **in** operator we can find the records in the given range of values .
- What ever we given the range in the range values written as output.

empno	empname	salary	empjob
10	a	1000	Security guard
20	b	2000	employ
30	c	3000	Team Leader
40	d	4000	HR

- **Ex:** select * from employee where salary in(2000, 4000) ;

empno	empname	salary	empjob
20	b	2000	employ

30	c	3000	Team Leader
40	d	4000	HR

6. Between Operator :

- Using with between operator we can find the column values of the given range , if we want to find out the salaries between one range to another range then go for this operator.

empno	empname	salary	empjob
10	a	1000	Security guard
20	b	2000	employ
30	c	3000	Team Leader
40	d	4000	HR

- Ex:** select * from employee where salary between 2000 and 5000 ;

empno	empname	salary	empjob
30	c	3000	Team Leader
40	d	4000	HR

Order by Cluse :

- Using with order by class we can perform operations on the table.
- Using with command we can access the rows with 2 times
 - ascending order
 - descending order

3. When we print the ascending order it will print the rows in that format, values can be display ascending order.
4. We can perform ascending order operations with the help of one column.
5. What ever giving the column name regarding that column name it will display the values with ascending order.
6. Ex : select * from emp where empno orderby asc ;

empno	empname	salary	empjob
10	a	1000	Security guard
20	b	2000	service man
20	b	2000	employ
30	c	3000	Team Leader
40	d	4000	HR

7. Ex : select * from emp where empno orderby desc ;

empno	empname	salary	empjob
40	d	4000	HR
30	c	3000	Team Leader
20	b	2000	employ
10	a	1000	Security guard

FUNCTIONS

:

Grouping Functions :

Grouping Functions are

1. min()
2. max()
3. avg()

empno	empname	salary	empjob
10	a	1000	Security guard
20	b	2000	employ
30	c	3000	Team Leader
40	d	4000	HR

1. **min()** :

min() function is used for finding the minimum value in the given table, this min() function will return which is the lowest value in the table on the given conditions.

syntax : select min(salary) from employee ;

empno	empname	salary	empjob
10	a	1000	Security guard

2. **max()** :

max() function is using for bring the highest salary in the given condition or column name.

syntax : select max(salary) from employee ;

empno	empname	salary	empjob
40	d	4000	HR

3. **avg()** :

avg() function returns the average of the given values of the specified column name.

syntax : select avg(salary) from employee ;

4. **stddev() :**

standard deviation function is used for finding the standard deviation of the given column.

This function is applied for number column names , not for character column names.

Syntax : select stddev(sal) from employee ;

5. **variance() :**

This function is using finding the variance of the column name.

This function is applied for number column names , not for character column names.

syntax : select variance(salary) from employee ;

CHARACTER FUNCTIONS :

1. **Ascii() :**

This is one of the character function , using this function we can give the character it will return the ascii value .

Syn:- **select ascii(character) from anytable;**

Ex1:- select ascii('a') from emp;

Ex2:- select distinct ascii('a') from student; // it is using stoping the repeated values from the result.

2. **Chr() :**

This function is using for return the character of the given value

This function will take the argument is number , that number match to which character , that character should be return.

Syn :- select chr(number) from table;

Ex :- select chr(67) from emp;

3. **Lower() :**

This function is using for converting given string from upper case letter to small case letter.

Syn :- select lower(character) from table;

Ex:- select lower('APEC') from emp ;

4. **Upper() :**

This is using for converting the given string from lower case letters to upper case letters .

Syn:- select upper(character) from table;

Ex:- select upper('apec') from emp;

5. **Concat() :**

This function is using for combining the 2 strings .

Using this function we give the two arguments first string each concatnate with 2nd string , the output is combination of that string.

Syn :- select concat(str1,str2) from table ;

select concat(name, surname) from table ;

Ex :- select concat('apec','DSNR') from emp;

6. SubStr() :

Using this function we can find out the part of the string from the entire string .

We give the index value regarding the index value we can access the part of the string.

Syn :-select substr(string or columnname , position) from tablename ;

Ex :- select substr('apec', 2) from emp ; //indexes starts from 1,2,3.....onwards

empno	empname	salary	empjob
10	ashok	1000	Security guard
20	arunkumar	2000	employ
30	akshay	3000	Team Leader
40	saicharan	4000	HR

Ex :- select substr(empname, 2) from emp ;

empno	empname	salary	empjob
10	ashok	1000	Security guard
20	arunkumar	2000	employ
30	akshay	3000	Team Leader
40	saicharan	4000	HR

7. InitCap() :

This function is using for displaying the given string first character is capital , rest of the characters in lower case .

Syn :- select initcap(string) from tablename;

Ex :- select initcap('avanthi') from student ;

8. Length() :

This funtion is using for finding the length of the given string , We return the length of the string in the form of the number.

Syn :- select length(string) from tablename ;

Ex :- select length('dilsuknagar') from student ;

SHOW COMMANDS :

1. show ti ;
2. show heading ;
3. show feedback ;
4. show pagesize ;
5. show linesize ;
6. show serveroutput ;
7. show sqlprompt ;
8. show und ;

SET Commands :

1. set ti on/off ;

1. This command is using for displaying the time all the sql prompt.
2. ti is nothing but Time.

3. When we want to display time on the console then go for set " ti " all.
4. When we all mode it will display the time on the sql prompt.
5. The time is display on the current system time .
6. When we mode of set ti off at that time , time does not appear on the sql prompt.
7. **syntax** : set ti on ; //for set the time
syntax : set ti off ; //for stop the time.
syntax : show ti ;

2. set heading on/off ;

1. Using thiscommand if we want to display the column names or if we want to not display the column names then go for heading command .
2. Heading is nothing but display the column name or not .
3. When we set heading on it will display the column names , When we set the heading off it will not display the column names , It will not display the column names but it will display only records with values.
4. When we see the values at the time we don't know the column names.
5. In a editor for all tables heading command is defaultly on .
6. **syntax** : set heading on ;
select * from employee ;

empno	empname	salary	empjob
10	ashok	1000	Security guard
20	arunkumar	2000	employ

30	akshay	3000	Team Leader
40	saicharan	4000	HR

7.

set heading off ;

10	ashok	1000	Security guard
20	arunkumar	2000	employ
30	akshay	3000	Team Leader
40	saicharan	4000	HR

8. **syntax** : show heading ;

3. **set feedback on/off :**

1. This command is using for display the information about output.
2. In general after executing the command , the output will be display like 4 rows selected, 1 row updated like this messages will be displayed.
3. If we want to access this information like 4 rows selected then go for set feedback on ;
4. So when we **set feed back on** , it will display the information about selected rows.
5. When we set the feed back off , then it will not display the information like 3 rows selected , 5 rows updated like that does not appear.
6. show feedback ; //it is show the status of the feedback command .

4. **set pagesize n :**

1. This command is using for setting the pagesize in the oracle .

2. Oracle editor contains no. of pages , it will display the information with the help of pages.
3. Pages are using for display the result of the table.
4. **syntax** : set pagesize n ;
Ex : set pagesize 40;
syntax : show pagesize ;

5. **set linesize n** :

1. This command is using for set the linesize in the oracle editor , defaultly the linesize is 80 .
2. If we want to increase the linesize , we have to set the command like set linesize n .
3. n is nothing but no. of linesize .
4. How many characters we want to display in a single line that no. we have to give .
5. **syntax** : set linesize n ;
Ex : set linesize 100 ;

6. **set serveroutput on/off** :

1. This command is using for display the output of perticular programs .
2. In pl/sql we write the programs , after writing the program we will execute it .
3. After execution we need to see the output , but before see the output we need to set the command is set server output on .
4. When we are in on mode it will display the output after executing the program .
5. When we are in off mode it will not display the output after executing the program.
6. In editor defaultly set server output is always off mode only .

- 7. **syntax** : set serveroutput on ;
 syntax : set serveroutput off ; //default
- 8. on : we can get the result of pl/sql programs.
 off : we can't get the result of pl/sql programs.

7. **set sqlprompt name :**

- 1. This command is using for set the prompt in the sql editor .
- 2. What ever give the message or string that message will display the sql prompt , after giving the name sql prompt display the given name instead of sql prompt .
- 3. It will replace the sql prompt displays the our name.
- 4. After set onwards our name work as like sql prompt , we can give the name with operators also .
- 5. **syntax** : set sqlprompt 'apec>' ;
 syntax : set sqlprompt 'ashok\$' ;
 ashok\$ select * from tablename ;

8. **set und char :**

- 1. This command is using for set the underline of the column names.
- 2. When we display the table data it will display the column names with the help of underscore character.
- 3. If we don't satisfy with that character , we can change the characters , we can give our own character.
- 4. In this command we give only one character, that character should be writing single code.

5. **syntax** : set und char ;

Ex : set und '\$' ;

9. **set ttittle msg** :

1. This command is used for set the heading to the our table.
2. What ever we give the message , that message will be appear as ttittle .
3. ttittle is nothing but top tittle and setting the header to the table.
4. ttittle information always display above the table , after executing the query ttittle will display with the result.
5. **syntax** : set ttittle ashoksoft ;

Ex : **select * from employee ;**

empno	empname	salary	empjob
10	ashok	1000	Security guard
20	arunkumar	2000	employ
30	akshay	3000	Team Leader
40	saicharan	4000	HR

6. When we print the date and name of the week it will display .

10. **set btittle msg** :

1. this command is using for display the information as a footer.
2. btittle is nothing but bottom tittle , what ever we set the btittle msg , the msg will be display as a footer after executing the command.
3. **syntax** : btittle msg ;
Ex : btittle ashok123;

Member Functions :

1. **abs()** : This function is using for finding the absolute value of the given number.
Ex : select abs(-10) from dual ;
output : 10
2. **power()** : This function is using for finding the power of the given number.
Ex : select power(2,3) from dual ;
3. **mod()** : This function is using for finding the remainder of the given value.
Ex : select mod(5,2) from dual ;
4. **sqrt()** : This function is using for finding the sqrt of the given number.
Ex : select sqrt(4) from dual ;

Mathematical Functions :

1. **log()** : log function is using for finding the log value of the given numbers.
Ex : select log(2,1) from dual ;
2. **sin()** : sin function is using for finding the sin value of the given radians.
Ex : select sin(90) from dual ;

3. **tan()** : tan is using for finding the tan values of the given number.

Ex : select tan(0) from dual ;

Date Functions :

1. **add_months()** :

- This functions using for displaying the details using the function. If we want to add_months to the existing date , then go for this functions.
- **In this finction it has 2 arguments .**
 1. First argument is 'date'
 2. Second argument is 'number'

that number is how many months we want to add.

- **syntax** : select add_months(date , number)
from student ;

Ex : select add_months('1-jan-14',4) from student ;

output : 1-may-14

2. **last_day()** :

- This function is using for finding the last day of the current month.
 - If months ends with the 30 , it will return 30 .
 - **syntax** : select last_day('1-jan-14') from student ;
- output** : 31-jan-14

3. **next_day()** :

1. This function is using for finding the day of given date.

2. In this function we give the arguments are first argument is date , then 2nd argument is name of the day .

3. Using with 2 arguments it will return next day after the given dates.

4. **syntax : select next_day(date , day) from student ;**

Ex : select next_day('2-may-2013' , 'thursday')
from student

output : 9-may-13

4. **months_between() :**

0. This function is used for finding the different of two months.

1. In this function we give the two arguments first argument is 'date' , second argument is 'date' .

2. This function finds the difference of the given 2 months.

3. **syntax : select months_between(date1 , date2) from student .**

Ex : select months_between('1-jan-14','1-mar-14') from student .

output : 2

Truncate :

1. This command is using for deleting the all rows in a table.

2. Truncate is nothing but clearing the table with out rows.

3. delete command is using for deleting a single row.
4. truncate command is using for deleting all rows in a given table name.
5. When we execute the truncate then we give the table name.
6. What ever we give the table name , in the table all rows are deleted.
7. After executing the truncate command table contains 'zero' rows or no rows
8. Truncate command doesn't delete the table but it will clear the table with zero rows.
 // delete=1 row delete
 // truncate = all rows deleted in a table
9. **syntax** : truncate table tablename ;
 truncate table student ;
 output : all records or rows deleted
10. After executing this command table contains zero rows.

Constants in Oracle :

1. sysdate :

1. This command is used for display the current date in the Oracle.
2. After executing this command it will display the current day in date format.
3. **syntax** : select sysdate from anytable;
 Ex : select sysdate from student ;

2. user :

1. This command is used for display the username which are operating the oracle .

2. When we enter the oracle editor , we give the username , password values .
3. When we ask the user it will return the username of the current editor.
4. The output is display in the user name .
5. **syntax** : select user from anytable ;
6. **Ex** : select user from emp ; //output : scott

3. uid :

1. uid is nothing but userid , after executing the uid command it will return the id number.
2. When we enter into the editor every user has one id number , when we ask the id number , it will return that number.
3. **syntax** : select uid from anytable ;
- Ex** : select uid from emp ;

4. rowid :

1. This is using for display in the rowid for every row , in oracle if we create one table oracle will give defaultly rowid.
2. rowid is using for representing the row.
3. rowid is using for identify in the row uniquely.
4. every row contains one unique number.
5. Using this unique number we can represents the row.
6. **syntax** : select rowid from anytable;
- Ex** : select rowid from emp ;

5. distinct :

1. distinct is a keyword using this keyword , we can represent single values from the result .
2. When we display the table if table contains repeated values it will display defaultly.

3. If we want to control repeated values and if we want to display repeated values only one time then go for this command.
4. distinct keyword is using for stopping the repeated values and display the repeated values only one time , not more than one time.
5. **syntax : select distinct salary from employee ;**

empno	empname	salary	empjob
10	ashok	1000	Security guard
20	arunkumar	2000	employ
30	akshay	3000	Team Leader
40	saicharan	3000	HR

6. // 3000 doesn't display 2nd time why because distinct keyword controlled by second 3000 value.

CONSTRAINTS IN ORACLE :

1. Unique
2. Not Null
3. Primary Key
4. Null

- constraints are using for giving the conditions on the column names in generally , if we create the table we don't give any instructions.
- If we want to give any conditions while creating the table then go for this constructions.

1. Unique :

1. unique constraint is using for declaring the column .
2. unique does not apply to the table names , only for columns.
3. Which column contains the unique constraint that column value should be unique.
4. column doesn't contains repetative values or redundancy values.
5. **syntax :** create table employee(eno number ,
ename varchar2(20) unique);

Syn :- create table tablename (columnname
datatype constraint) ;

Ex :- create table emp2(eno number unique , ename
varchar2(20)) ;

2. Not Null :

1. This constraint is using for giving the column names , it doesn't apply to table names.
2. When we create the table , if we contain character datatypes instead of value , we can give the null .

3. When we want to doesn't accept the null then go for this constraint.
4. While creating the table to give the column names with the not null constraint.
5. After giving this constraint it doesn't accept the null value , null value is nothing but empty value.

Syn :- create table tablename (columnname datatype not null) ; Ex :-create table bank(bno number ,bname varchar2(20) not null); Ex :- insert into bank values(1,'apec'); Ex :- insert into bank values(1, null);
//wrong

3. **Primary Key :**

1. primary key is using for giving the conditions on the columns , when we declare one column with primary key it doesn't accept null values & repetative value.
2. Primary key is using for making the relations to the table , we can connection with the one table to another table for access the information.
3. primary key is very important role while making the relationships to the table.
4. primary key is applying only column names .
5. primary key does not apply to tables.
6. Using with primary key if we want to make the relationship with the another table using with another key.
7. For every table only one primary key column.

Syn :- create table tablename (columnname datatype primarykey) ; Ex :- create table student (eno number primary key) ;

4. **NULL :**

1. null is a keyword in a Oracle.
2. null is not a value , it indicates this column contains no value.
3. When we insert in the values if we doesn't find any value to the given column then go for null.
4. When we declare one column with not null constraint at the time we don't use the null key word.
5. If we try to use the null keyword it will show error, and that row doesn't insert.

Syn :- create table emp3(eno number , ename varchar2(20)) ; Ex :- insert into emp3 values(1,'apec') ; Ex :- insert into emp3 values(1, null) ; Ex :- insert into emp3 values(2, null) ;

Clauses in SQL :

clauses are using for accessing the records on the given certain conditions , give any condition regarding the condition is working.

empno	empname	salary	empjob
10	ashok	1000	Security guard
20	arunkumar	2000	employ
30	akshay	3000	Team Leader
40	saicharan	4000	HR

1. **where :**

1. syntax : select * from tablename where condition ;
2. Ex : select * from employee where empno=10 ;

empno	empname	salary	empjob
10	ashok	1000	Security guard

3. Ex : select * from employee where empno>=20 ;

empno	empname	salary	empjob
20	arunkumar	2000	employ
30	akshay	3000	Team Leader
40	saicharan	4000	HR

2. **order by :**

1. This clause is using for displaying the data in the form of orders.
2. The orders are ascending order or descending order .
3. Ascending order is nothing but small to big , descending order is nothing but big value to small.
4. syntax : select * from employee order by salary asc ;

empno	empname	salary	empjob
10	ashok	1000	Security guard
20	arunkumar	2000	employ
30	akshay	3000	Team Leader
40	saicharan	4000	HR

5. syntax : select * from employee order by salary desc ;

empno	empname	salary	empjob
40	saicharan	4000	HR
30	akshay	3000	Team Leader
20	arunkumar	2000	employ
10	ashok	1000	Security guard

3. **group by :**

1. This function is using for displaying the data in the form of selecting multiple columns depending on one column.
2. syntax : select * from employee group by salary ;
Ex : select sum(salary) , count(salary) from employee group by salary ;

4. **Having :**

1. This clause is using for finding the result of the given having condition.
2. This will display the result of who are satisfy the condition.
3. select * from employee having sno=10 ;

5. **starting with :**

1. Using with clause we can find out whether the given string is starting with the given character or not .
2. We give the character as a argument in a single code.
3. syntax : select * from employee where empname starting with('a');

empno	empname	salary	empjob
10	ashok	1000	Security guard
20	arunkumar	2000	employ
30	akshay	3000	Team Leader

Table set commands :

1. union :

1. This command is using for when we want to combining the two tables union can display , display the data of same in two tables.
2. syntax : select * from student union select * from student1 ;

2. union all :

1. Union all is nothing but combining the two tables given matched values or non-matched values.
2. syntax : select * from student union all select * from student1 ;
//combining the 2 tables with no conditions .

3. intersect :

1. intersect is using for combining the two tables as well as what are the same values contained in 2 tables.
2. intersect will display the data should containing two tables.
3. syntax : select * from table1 intersect select * from table2 ;

4. minus :

1. Minus is using for display in the 2 tables data which values does not match with the two tables that will display .
2. syntax : select * from student minus select * from student1 ;

- copying the data from one table to another table .
- This is using for when we want create 2nd table with the first table structure .
- In general if we want to create with same values means we have to create another table .
- Oracle provides one feature copying the table entire table and structure with values.
- No need to create the tables and inserting the values for the 2nd table.
- In the second table contains all the rows and values should be same with the first table name.
- When we copy the table & the structure also copied.
- When we execute the desc command both structure are same.
- syntax : create table tablename as select * from tablename ;
- Ex : create table student1 as select * from student ;

Tab :

- This command is using for displaying the all tables information available in the oracle server.
- It will display the all tablenames
- syntax : select * from tab ;

User_Objects :

- This command is using for displaying the all tables and views , which are available in the oracle server.
- It is not only display the tablenames and also all those details it will give.
- syntax : select * from user_objects ;

Opening the applications from the oracle editor :

1. sql plus ----- console editor
2. sql plusw ----- GUI editor
3. select sql+plus

Views :

1. view is nothing but table .
2. When we want to create one table with same values then go for views.

3. View occupies less memory than compared to table memory.
4. We can perform all operations on views also.
5. syntax : create view viewname as select * from tablename ;
Ex : create view student 1 as select * from student ;
6. syntax : insert into viewname values(c1value , c2value);
7. syntax : delete viewname where condition ;
8. syntax : drop view viewname ;
9. syntax : update viewname set columnname=value where condition ;

Joins :

1. Joins are used for combining the two tables information.
2. When we want to see the 2 tables information at a time then go for joins.
3. After joining the tables it will display the 2nd table information at a time in a single table .
4. After joining the tables entire information in one table.
5. **Joins are 3 types :**
 1. self join
 2. equi join
 3. non-equi join
 4. **self join :**

1. self join is nothing but combining the same table with the 2 same names.
2. syntax : select * from emp , emp ;
5. **equi join :**
 1. Using this joins , we can combine the two tables with the equal condition.
 2. When we join this 2 tables before we will give the condition.
 3. After satisfy the condition only it will show the records.
 4. syntax : select e.empno , e.name , e.job , k.empno , k.ename , k.job from emp e , emp k where k.empno=e.empno ; // e & k are alias names
6. **non-equi join :**
 1. syntax : select e.empno , e.ename , e.job , k.empno , k.ename , k.job from emp e , emp k where k.empno < > e.empno ;

Rename the Table :

1. If we want to change the table name from one into another name then go for this cmd .
2. After rename in the table old name is does not appear.
3. The new name will be apply to table.
4. syntax : rename oldtablename to newtablename ;
Ex : rename emp to emp1 ;

Opening application from sql * plus :

1. syntax : \$ applicationname
2. Ex1 : \$ notepad
3. Ex2 : \$ calc
4. Ex3 : \$ cmd //console or dos prompt opened

Executing multiple queries from a file to sql * plus editor :

1. Using this procedure we can execute group of queries in a single statement.
2. In generally editor we can execute only one query at a time.
3. Using with sql editor , we can't execute group of queries at a time .
4. If we want to execute group of queries in the form of file is possible .
5. We write the all no. of queries as a file , we save that file as filename.sql
6. sql is the execution of oracle programs
7. Steps for executing the file :
 1. open the notepad
 2. write the first query
 3. after complete the query put the(/) instead of (;)

4. after put the slash , write the 2nd query , later 2nd query put the slash.
5. like this procedure how many queries we have
6. After completing the queries , close the file , before closing the file save the file name as filename.sql
7. Then we open the sql * plus editor.
8. we give the path of the file.
9. we execute the file in the editor.
@filepath
@c://commands.sql

Bind Variables :

1. This variables are using for giving the values to the where conditions.
2. At runtime to allocate the values to the variables.
3. If we directly assign the values it will execute only one time.
4. If we want to execute 2nd time , then we have to write the entire condition.
5. **general** : select * from emp where sal=100 ;
Use bind variable : select * from emp where sal='&sal' ;
6. **Characters (use bind variable)** : select * from emp where ename='&name' ;

vsiz() function :

1. This function is using for displaying the size of the particular variable .
2. It will display the output of how many bytes accupied by the variable.
3. **syntax** : select vsiz(value) from dual ; //value=100

User environment : userenv() function :

1. This function is using for describing the details of the user.
2. This function as 2 arguments for every argument , it will display 1 output
3. The arguments are terminal & language
4. **syntax** : select userenv('terminal') ,
userenv('language') from dual ;

Set Command :

1. **set space n** : This cmd is using for giving the spaces to the columns
2. If we want to give space between the column names then go for space cmd. This default column space is "Zero"
3. The highest column space is 10
4. set the n value is 0 to 10

5. set space n ;

Ex : set space 5 ;

- giving column names to the functions after executing .
 - In general when we execute function it will display the column name like function name only.
 - If we want to give the column name to the function then you can give.
 - syntax : select function-name variable-name from anytable ;
- Ex : select max(sal) highest from dual ;

PL/SQL :

1. pl/sql is nothing but procedural language/sql , This language is using for writing the programs in the sql editor.
2. We can write the all programs like C , C++ this language also provides variables data types , program is nothing but executing set of instructions , statement is nothing but group of words .
3. We write pl/sql programs in sql * plus editor , sql * plus editor is using for writing pl/sql programs.
4. In generally once we write the program can be compile & run the programs.
5. In pl/sql programs after writing the program we save the file name as filename.sql
add.sql OR sub.sql
6. We can write the pl/sql programs in notepad also , after writing into the notepad , we save the filename as filename.sql

7. We can save the file in oracle server Or we can save the file into our own given path like c,d,e drives.
8. After saving the file we will execute the programs , After executing the program we can see the output, But in pl/sql after executing the program we can't see the output. But program is executed
9. Why because we want to see the output , we have to set the command is like "set serveroutput on" ; When serveroutput cmd is on it will display the output .
10. When serveroutput is off , at the time output is not displayed.
11. We have to said these settings before executing the pl/sql program .
12. If we said after executing the program there is no use , before executing if we have errors in the file , it will show those are called compiletime errors .
13. After clearing the errors we can executing the program.
14. **Using with the pl/sql we can do with the conditional statements.**
 1. if
 2. if else
15. control statements
 1. while
 2. do while
 3. for loop

Data Types in pl/sql :

1. number
2. varchar

- 3. varchar2
- 4. date

Structure of the pl/sql program :

In the above structure describes how to write the pl/sql programs

1. declare block :

- 1. In this block we can declare the variables , we can declare the variables with the help of some specified data type.
- 2. Every variable should have a one data type , every variable declaration ends with the semicolon.
- 3. semicolon indicates end of the statement.

2. begin block :

- 1. This block is using writing the real code of the program.
- 2. In begin part , we write the program & task of the program.
- 3. For completion of this task it will use the declare part .
- 4. Using the declare part is nothing but using the variables .

3. end block : end is nothing but indicating the our pl/sql program is over .

- 4. ' / ' is nothing but start the execution of the pl/sql program.It is for oracle server , not for pl/sql program.

How to declare the variable in pl/sql :

variablename	datatype ;
x	number ;
name	varchar(20) ;
jdate	date ;

Installing the variable in pl/sql :

variablename	datatype :=value ;
x	number:=30 ;
y	varchar(20) :='hai' ;
d	date:='10-may-13' ;

Output function in pl/sql : dbms_output.put_line() ;

1. This is output procedure in pl/sql , it is using for displaying the out to the user.
2. When we print the values Or when we want to display some information to the user using this procedure only.
3. dbms_output is a package name , put_line is a function name

Write a one pl/sql program for display the welcome message :

1. open the notepad
2. declare begin dbms_output.put_line("welcome"); end;
/
3. save filename as "file.sql"
4. close the file
- 5.

```
sql< @filepath      c:\file.sql
set serveroutput on
```

6. output :welcome

If statement :

1. It is one type of conditional statement , it is using for comparing the 2 values Or comparing the 2 variables.
2. if condition is satisfied Or condition is true then if block will be executed.
3. if condition is false then does not execute any code.
4. if condition is true we get the output .
5. if condition is false we have no output.

INTRODUCTION

SQL is divided into the following

- [Data Definition Language \(DDL\)](#)
- [Data Manipulation Language \(DML\)](#)
- [Data Retrieval Language \(DRL\)](#)
- [Transaction Control Language \(TCL\)](#)

- Data Control Language (DCL)

DDL -- create, alter, drop, truncate, rename

DML -- insert, update, delete

DRL -- select

TCL -- commit, rollback, savepoint

DCL -- grant, revoke

CREATE TABLE

SYNTAX :

Create table <table_name> (col1 datatype1, col2 datatype2 ...coln datatype_n);

Ex:

SQL> create table student (no number (2), name varchar (10), marks number (3));

INSERT

This will be used to insert the records into table.

We have two methods to insert.

- By value method
- By address method

a. **USING VALUE METHOD**

Syntax:

insert into <table_name> values (value1, value2, value3 Valuen);

Ex:

```
SQL> insert into student values (1, 'sudha', 100);
SQL> insert into student values (2, 'saketh', 200);
```

To insert a new record again you have to type entire insert command, if there are lot of records this will be difficult.

This will be avoided by using address method.

b. USING ADDRESS METHOD

Syntax:

```
insert into <table_name> values (&col1, &col2,
&col3 .... &coln);
```

This will prompt you for the values but for every insert you have to use forward slash.

Ex:

```
SQL> insert into student values (&no, '&name',
&marks);
```

```
Enter value for no: 1
Enter value for name: Jagan
Enter value for marks: 300
old 1: insert into student values(&no,
'&name', &marks)
new 1: insert into student values(1,
'Jagan', 300)
```

```
SQL> /
```

```
Enter value for no: 2
Enter value for name: Naren
Enter value for marks: 400
old 1: insert into student values(&no,
'&name', &marks)
```

```
new 1: insert into student values (2,
'Naren', 400)
```

c. **INSERTING DATA INTO SPECIFIED COLUMNS USING VALUE METHOD**

Syntax:

**insert into <table_name>(col1, col2, col3 ... Coln)
values (value1, value2, value3 ...Valuen);**

Ex:

SQL> insert into student (no, name) values (3,
'Ramesh');

SQL> insert into student (no, name) values (4,
'Madhu');

d. **INSERTING DATA INTO SPECIFIED COLUMNS USING ADDRESS METHOD**

Syntax:

**insert into <table_name>(col1, col2, col3 ... coln)
values (&col1, &col2, &col3 &coln);**

This will prompt you for the values but for every insert you have to use forward slash.

Ex:

SQL> insert into student (no, name) values (&no,
'&name');

```
Enter value for no: 5
Enter value for name: Visu
old 1: insert into student (no, name)
values(&no, '&name')
new 1: insert into student (no, name)
values(5, 'Visu')
```

SQL> /

```
Enter value for no: 6
Enter value for name: Rattu
old 1: insert into student (no, name)
values(&no, '&name')
new 1: insert into student (no, name)
values(6, 'Rattu')
```

SELECTING DATA

Syntax:

Select * from <table_name>; -- here * indicates all columns

or

Select col1, col2, ... coln from <table_name>;

Ex:

SQL> **select * from student;**

NO	NAME	MARKS
---	-----	-----
1	Sudha	100
2	Saketh	200
1	Jagan	300
2	Naren	400
3	Ramesh	
4	Madhu	
5	Visu	
6	Rattu	

SQL> **select no, name, marks from student;**

NO	NAME	MARKS
---	-----	-----
1	Sudha	100
2	Saketh	200
1	Jagan	300
2	Naren	400
3	Ramesh	

```

4    Madhu
5    Visu
6    Rattu

```

SQL> select no, name from student;

```

      NO      NAME
----  -
1      Sudha
2      Saketh
1      Jagan
2      Naren
3      Ramesh
4      Madhu
5      Visu
6      Rattu

```

CONDITIONAL SELECTIONS AND OPERATORS

We have two clauses used in this

- Where
- Order by

USING WHERE

Syntax:

select * from <table_name> where <condition>;

the following are the different types of operators used in where clause.

- Arithmetic operators
- Comparison operators
- Logical operators
- Arithmetic operators -- highest precedence +, -, *, /
- Comparison operators =, !=, >, <, >=, <=, <>

- between, not between
- in, not in
- null, not null
- like
- Logical operators
 - And
 - Or -- lowest precedence
 - not

a. **USING** =, >, <, >=, <=, !=, <>

Ex:

SQL> select * from student where no = 2;

NO	NAME	MARKS
2	Saketh	200
2	Naren	400

SQL> select * from student where no < 2;

NO	NAME	MARKS
1	Sudha	100
1	Jagan	300

SQL> select * from student where no > 2;

NO	NAME	MARKS
3	Ramesh	
4	Madhu	
5	Visu	
6	Rattu	

SQL> select * from student where no <= 2;

NO	NAME	MARKS
---	-----	-----
1	Sudha	100
2	Saketh	200
1	Jagan	300
2	Naren	400

SQL> select * from student where no >= 2;

NO	NAME	MARKS
---	-----	-----
2	Saketh	200
2	Naren	400
3	Ramesh	
4	Madhu	
5	Visu	
6	Rattu	

SQL> select * from student where no != 2;

NO	NAME	MARKS
---	-----	-----
1	Sudha	100
1	Jagan	300
3	Ramesh	
4	Madhu	
5	Visu	
6	Rattu	

SQL> select * from student where no <> 2;

NO	NAME	MARKS
---	-----	-----
1	Sudha	100
1	Jagan	300
3	Ramesh	
4	Madhu	
5	Visu	
6	Rattu	

b. USING AND

This will gives the output when all the conditions become true.

Syntax:

**select * from <table_name> where <condition1>
and <condition2> and .. <conditionnn>;**

Ex:

**SQL> select * from student where no = 2 and
marks >= 200;**

	NO	NAME	MARKS
	---	-----	-----
	2	Saketh	200
	2	Naren	400

c. USING OR

This will gives the output when either of the conditions become true.

Syntax:

**select * from <table_name> where <condition1>
and <condition2> or .. <conditionnn>;**

Ex:

**SQL> select * from student where no = 2 or marks
>= 200;**

	NO	NAME	MARKS
	---	-----	-----
	2	Saketh	200
	1	Jagan	300
	2	Naren	400

d. USING BETWEEN

This will gives the output based on the column and its lower bound, upperbound.

Syntax:

select * from <table_name> where <col> between <lower bound> and <upper bound>;

Ex:

SQL> select * from student where marks between 200 and 400;

NO	NAME	MARKS
2	Saketh	200
1	Jagan	300
2	Naren	400

e. USING NOT BETWEEN

This will gives the output based on the column which values are not in its lower bound, upperbound.

Syntax:

select * from <table_name> where <col> not between <lower bound> and <upper bound>;

Ex:

SQL> select * from student where marks not between 200 and 400;

NO	NAME	MARKS
1	Sudha	100

f. USING IN

This will gives the output based on the column and its list of values specified.

Syntax:

select * from <table_name> where <col> in (value1, value2, value3 ... valuen);

Ex:

SQL> select * from student where no in (1, 2, 3);

NO	NAME	MARKS
1	Sudha	100
2	Saketh	200
1	Jagan	300
2	Naren	400
3	Ramesh	

g. **USING NOT IN**

This will gives the output based on the column which values are not in the list of values specified.

Syntax:

select * from <table_name> where <col> not in (value1, value2, value3 ... valuen);

Ex:

SQL> select * from student where no not in (1, 2, 3);

NO	NAME	MARKS
4	Madhu	
5	Visu	
6	Rattu	

h. USING NULL

This will gives the output based on the null values in the specified column.

Syntax:

select * from <table_name> where <col> is null;

Ex:

SQL> select * from student where marks is null;

NO	NAME	MARKS
---	-----	-----
3	Ramesh	
4	Madhu	
5	Visu	
6	Rattu	

i. USING NOT NULL

This will gives the output based on the not null values in the specified column.

Syntax:

select * from <table_name> where <col> is not null;

Ex:

SQL> select * from student where marks is not null;

NO	NAME	MARKS
---	-----	-----
1	Sudha	100
2	Saketh	200
1	Jagan	300
2	Naren	400

j. USING LIKE

This will be used to search through the rows of database column based on the pattern you specify.

Syntax:

select * from <table_name> where <col> like <pattern>;

Ex:

- i. **This will give the rows whose marks are 100.**

SQL> select * from student where marks like 100;

ii.	NO	NAME	MARKS
iii.	---	-----	-----
iv.	1	Sudha	100

- v. **This will give the rows whose name start with 'S'.**

SQL> select * from student where name like 'S%';

vi.	NO	NAME	MARKS
vii.	---	-----	-----
viii.	1	Sudha	100
ix.	2	Saketh	200

- x. **This will give the rows whose name ends with 'h'.**

SQL> select * from student where name like '%h';

xi.	NO	NAME	MARKS
xii.	---	-----	-----
xiii.	2	Saketh	200
xiv.	3	Ramesh	

- xvi. **This will give the rows whose name's second letter start with 'a'.**

SQL> select * from student where name like
'_a%';

xvii.	NO	NAME	MARKS
xviii.	---	-----	-----
xix.	2	Saketh	200
xx.	1	Jagan	300
xxi.	2	Naren	400
xxii.	3	Ramesh	
xxiii.	4	Madhu	
xxiv.	6	Rattu	

xxv. **This will give the rows whose name's third letter start with 'd'.**

SQL> select * from student where name like
'__d%';

xxvi.	NO	NAME	MARKS
xxvii.	---	-----	-----
xxviii.	1	Sudha	100
xxix.	4	Madhu	

xxx. **This will give the rows whose name's second letter start with 't' from ending.**

SQL> select * from student where name like
'%_t%';

xxxi.	NO	NAME	MARKS
xxxii.	---	-----	-----
xxxiii.	2	Saketh	200
xxxiv.	6	Rattu	

xxxv. **This will give the rows whose name's third letter start with 'e' from ending.**

SQL> select * from student where name like
'%e__%';

xxxvi.	NO	NAME	MARKS
xxxvii.	---	-----	-----
xxxviii.	2	Saketh	200
xxxix.	3	Ramesh	

xl. **This will give the rows whose name cotains 2 a's.**

SQL> select * from student where name like
'%a% a %';

xli.	NO	NAME	MARKS
xlii.	---	-----	-----
xliii.	1	Jagan	300

* You have to specify the patterns in like using underscore (_).

USING ORDER BY

This will be used to ordering the columns data (ascending or descending).

Syntax:

Select * from <table_name> order by <col> desc;

By default oracle will use ascending order. If you want output in descending order you have to use desc keyword after the column.

Ex:

SQL> select * from student order by no;

	NO	NAME	MARKS
	---	-----	-----
	1	Sudha	100
	1	Jagan	300
	2	Saketh	200
	2	Naren	400
	3	Ramesh	
	4	Madhu	
	5	Visu	
	6	Rattu	

SQL> select * from student order by no desc;

	NO	NAME	MARKS
	---	-----	-----
	6	Rattu	

5	Visu	
4	Madhu	
3	Ramesh	
2	Saketh	200
2	Naren	400
1	Sudha	100
1	Jagan	300

USING DML

USING UPDATE

This can be used to modify the table data.

Syntax:

Update <table_name> set <col1> = value1, <col2> = value2 where <condition>;

Ex:

SQL> update student set marks = 500;

If you are not specifying any condition this will update entire table.

SQL> update student set marks = 500 where no = 2;

SQL> update student set marks = 500, name = 'Venu' where no = 1;

USING DELETE

This can be used to delete the table data temporarily.

Syntax:

Delete <table_name> where <condition>;

Ex:

SQL> delete student;

If you are not specifying any condition this will delete entire table.

SQL> delete student where no = 2;

USING DDL

USING ALTER

This can be used to add or remove columns and to modify the precision of the datatype.

a. ADDING COLUMN

Syntax:

alter table <table_name> add <col datatype>;

Ex:

SQL> alter table student add sdob date;

b. REMOVING COLUMN

Syntax:

alter table <table_name> drop <col datatype>;

Ex:

SQL> alter table student drop column sdob;

c. INCREASING OR DECREASING PRECISION OF A COLUMN

Syntax:

alter table <table_name> modify <col datatype>;

Ex:

SQL> alter table student modify marks number(5);

* To decrease precision the column should be empty.

d. **MAKING COLUMN UNUSED**

Syntax:

alter table <table_name> set unused column <col>;

Ex:

**SQL> alter table student set unused column
marks;**

Even though the column is unused still it will occupy memory.

e. **DROPPING UNUSED COLUMNS**

Syntax:

alter table <table_name> drop unused columns;

Ex:

SQL> alter table student drop unused columns;

* You can not drop individual unused columns of a table.

f. **RENAMING COLUMN**

Syntax:

**alter table <table_name> rename column
<old_col_name> to <new_col_name>;**

Ex:

SQL> alter table student rename column marks to smarks;

USING TRUNCATE

This can be used to delete the entire table data permanently.

Syntax:

truncate table <table_name>;

Ex:

SQL> truncate table student;

USING DROP

This will be used to drop the database object;

Syntax:

Drop table <table_name>;

Ex:

SQL> drop table student;

USING RENAME

This will be used to rename the database object;

Syntax:

rename <old_table_name> to <new_table_name>;

Ex:

SQL> rename student to stud;

USING TCL

USING COMMIT

This will be used to save the work.

Commit is of two types.

- Implicit
- Explicit

a. **IMPLICIT**

This will be issued by oracle internally in two situations.

- When any DDL operation is performed.
- When you are exiting from SQL * PLUS.

b. **EXPLICIT**

This will be issued by the user.

Syntax:

Commit or commit work;

* When ever you committed then the transaction was completed.

USING ROLLBACK

This will undo the operation.

This will be applied in two methods.

- Upto previous commit
- Upto previous rollback

Syntax:

Roll or roll work;

Or

Rollback or rollback work;

* While process is going on, if suddenly power goes then oracle will rollback the transaction.

USING SAVEPOINT

You can use savepoints to rollback portions of your current set of transactions.

Syntax:

Savepoint <savepoint_name>;

Ex:

```
SQL> savepoint s1;
SQL> insert into student values(1, 'a', 100);
SQL> savepoint s2;
SQL> insert into student values(2, 'b', 200);
SQL> savepoint s3;
SQL> insert into student values(3, 'c',
300);
SQL> savepoint s4;
SQL> insert into student values(4, 'd',
400);
```

Before rollback

SQL> select * from student;

NO	NAME	MARKS
1	a	100
2	b	200
3	c	300
4	d	400

SQL> rollback to savepoint s3;

Or

SQL> rollback to s3;

This will rollback last two records.

SQL> select * from student;

NO	NAME	MARKS
1	a	100
2	b	200

USING DCL

DCL commands are used to granting and revoking the permissions.

USING GRANT

This is used to grant the privileges to other users.

Syntax:

**Grant <privileges> on <object_name> to <user_name>
[with grant option];**

Ex:

SQL> grant select on student to sudha; -- you can give individual privilege

SQL> grant select, insert on student to sudha; -- you can give set of privileges

SQL> grant all on student to sudha; -- you can give all privileges

The sudha user has to use dot method to access the object.

SQL> select * from saketh.student;

The sudha user can not grant permission on student table to other users. To get this type of option use the following.
SQL> grant all on student to sudha with grant option;

Now sudha user also grant permissions on student table.

USING REVOKE

This is used to revoke the privileges from the users to which you granted the privileges.

Syntax:

**Revoke <privileges> on <object_name> from
<user_name>;**

Ex:

SQL> revoke select on student from sudha; -- you can revoke individual privilege

SQL> revoke select, insert on student from sudha; -- you can revoke set of privileges

SQL> revoke all on student from sudha; -- you can revoke all privileges

USING ALIASES

CREATE WITH SELECT

We can create a table using existing table [along with data].

Syntax:

Create table <new_table_name> [col1, col2, col3 ... coln] as select * from <old_table_name>;

Ex:

SQL> create table student1 as select * from student;

Creating table with your own column names.

SQL> create table student2(sno, sname, smarks) as select * from student;

Creating table with specified columns.

SQL> create table student3 as select no,name from student;

Creating table with out table data.

SQL> create table student2(sno, sname, smarks) as select * from student where 1 = 2;

In the above where clause give any condition which does not satisfy.

INSERT WITH SELECT

Using this we can insert existing table data to a another table in a single trip. But the table structure should be same.

Syntax:

Insert into <table1> select * from <table2>;

Ex:

SQL> insert into student1 select * from student;

Inserting data into specified columns

**SQL> insert into student1(no, name) select no, name
from student;**

COLUMN ALIASES

Syntax:

**Select <original_col> <alias_name> from
<table_name>;**

Ex:

**SQL> select no sno from student; or
SQL> select no "sno" from student;**

TABLE ALIASES

If you are using table aliases you can use dot method to the columns.

Syntax:

**Select <alias_name>.<col1>, <alias_name>.<col2> ...
<alias_name>.<coln> from <table_name>
<alias_name>;**

Ex:

SQL> select s.no, s.name from student s;

USING MERGE

MERGE

You can use merge command to perform insert and update in a single command.

Ex:

```
SQL> Merge into student1 s1
      Using (select *From student2) s2
      On(s1.no=s2.no)
      When matched then
      Update set marks = s2.marks
      When not matched then
      Insert (s1.no,s1.name,s1.marks)
      Values(s2.no,s2.name,s2.marks);
```

In the above the two tables are with the same structure but we can merge different structured tables also but the datatype of the columns should match.

Assume that student1 has columns like no,name,marks and student2 has columns like no, name, hno, city.

```
SQL> Merge into student1 s1
      Using (select *From student2) s2
      On(s1.no=s2.no)
      When matched then
      Update set marks = s2.hno
      When not matched then
      Insert (s1.no,s1.name,s1.marks)
      Values(s2.no,s2.name,s2.hno);
```

MULTIBLE INSERTS

We have table called DEPT with the following columns and data

DEPTNO	DNAME	LOC
10	accounting	new york
20	research	dallas
30	sales	Chicago
40	operations	boston

A. CREATE STUDENT TABLE

SQL> Create table student(no number(2),name varchar(2),marks number(3));

B.MULTI INSERT WITH ALL FIELDS

```
C.      SQL> Insert all
D.              Into student values (1,'a',100)
E.              Into student values (2,'b',200)
F.              Into student values (3,'c',300)
G.              Select *from dept where
                deptno=10;
```

-- This inserts 3 rows

H. MULTI INSERT WITH SPECIFIED FIELDS

```
I.      SQL> insert all
J.              Into student (no,name)
                values (4,'d')
K.              Into student (name,marks)
                values ('e',400)
L.              Into student values (3,'c',300)
M.              Select *from dept where
                deptno=10;
```

-- This inserts 3 rows

N. MULTI INSERT WITH DUPLICATE ROWS

```
O.      SQL> insert all
P.              Into student values (1,'a',100)
Q.              Into student values (2,'b',200)
R.              Into student values (3,'c',300)
S.              Select *from dept where deptno >
                10;
```

-- This inserts 9 rows because in the select statement retrieves 3 records (3 inserts for each row retrieved)

T.MULTI INSERT WITH CONDITIONS BASED

```
U.      SQL> Insert all
```

```

V.          When deptno > 10 then
W.          Into student1 values(1,'a',100)
X.          When dname = 'SALES' then
Y.          Into student2 values(2,'b',200)
Z.          When loc = 'NEW YORK' then
AA.         Into student3 values(3,'c',300)
BB.         Select *from dept where
            deptno>10;

```

-- This inserts 4 rows because the first condition satisfied 3 times, second condition satisfied once and the last none.

CC. **MULTI INSERT WITH CONDITIONS BASED AND ELSE**

```

DD.         SQL> Insert all
EE.          When deptno > 100 then
FF.          Into student1 values(1,'a',100)
GG.          When dname = 'S' then
HH.          Into student2 values(2,'b',200)
II.          When loc = 'NEW YORK' then
JJ.          Into student3 values(3,'c',300)
KK.          Else
LL.          Into student values(4,'d',400)
MM.          Select *from dept where
            deptno>10;

```

-- This inserts 3 records because the else satisfied 3 times

NN. **MULTI INSERT WITH CONDITIONS BASED AND FIRST**

```

OO.         SQL> Insert first
PP.          When deptno = 20 then
QQ.          Into student1 values(1,'a',100)
RR.          When dname = 'RESEARCH' then
SS.          Into student2 values(2,'b',200)
TT.          When loc = 'NEW YORK' then
UU.          Into student3 values(3,'c',300)

```

```
VV.          Select *from dept where
             deptno=20;
```

-- This inserts 1 record because the first clause avoid to check the remaining conditions once the condition is satisfied.

WW. MULTI INSERT WITH CONDITIONS BASED, FIRST AND ELSE

```
XX.          SQL> Insert first
YY.          When deptno = 30 then
ZZ.          Into student1 values(1,'a',100)
AAA.         When dname = 'R' then
BBB.         Into student2
              values(2,'b',200)
CCC.         When loc = 'NEW YORK'
              then
DDD.         Into student3
              values(3,'c',300)
EEE.         Else
FFF.         Into student
              values(4,'d',400)
GGG.         Select *from dept where
             deptno=20;
```

-- This inserts 1 record because the else clause satisfied once

HHH. MULTI INSERT WITH MULTIPLE TABLES

```
III.         SQL> Insert all
JJJ.         Into student1
              values(1,'a',100)
KKK.         Into student2
              values(2,'b',200)
LLL.         Into student3
              values(3,'c',300)
MMM.         Select *from dept where
             deptno=10;
```

-- This inserts 3 rows

** You can use multi tables with specified fields, with duplicate rows, with conditions, with first and else clauses.

FUNCTIONS

Functions can be categorized as follows.

- Single row functions
- Group functions

SINGLE ROW FUNCTIONS

Single row functions can be categorized into five.

These will be applied for each row and produces individual output for each row.

- Numeric functions
- String functions
- Date functions
- Miscellaneous functions
- Conversion functions

NUMERIC FUNCTIONS

- Abs
- Sign
- Sqrt
- Mod
- Nvl
- Power
- Exp
- Ln
- Log

- Ceil
- Floor
- Round
- Trunk
- Bitand
- Greatest
- Least
- Coalesce

A. **ABS**

Absolute value is the measure of the magnitude of value. Absolute value is always a positive number.

Syntax:

abs (value)

Ex:

SQL> select abs(5), abs(-5), abs(0), abs(null) from dual;

ABS (5)	ABS (-5)	ABS (0)	ABS (NULL)
-----	-----	-----	-----
5	-5	0	

B. **SIGN**

Sign gives the sign of a value.

Syntax:

sign (value)

Ex:

SQL> select sign(5), sign(-5), sign(0), sign(null) from dual;

SIGN (5)	SIGN (-5)	SIGN (0)	SIGN (NULL)
----------	-----------	----------	-------------

-----	-----	-----
1	-1	0

C. **SQRT**

This will give the square root of the given value.

Syntax:

sqrt (value) -- here value must be positive.

Ex:

**SQL> select sqrt(4), sqrt(0), sqrt(null), sqrt(1)
from dual;**

SQRT (4)	SQRT (0)	SQRT (NULL)	SQRT (1)
-----	-----	-----	-----
2	0		1

D. **MOD**

This will give the remainder.

Syntax:

mod (value, divisor)

Ex:

**SQL> select mod(7,4), mod(1,5), mod(null,null),
mod(0,0), mod(-7,4) from dual;**

MOD (7, 4)	MOD (1, 5)	MOD (NULL, NULL)	MOD (0, 0)
MOD (-7, 4)			
-----	-----	-----	-----
3	1		0
-3			

E. **NVL**

This will substitutes the specified value in the place of null values.

Syntax:

nvl (null_col, replacement_value)

Ex:

SQL> select * from student; -- here for 3rd row marks value is null

NO	NAME	MARKS
1	a	100
2	b	200
3	c	

SQL> select no, name, nvl(marks,300) from student;

NO	NAME	NVL (MARKS, 300)
1	a	100
2	b	200
3	c	300

SQL> select nvl(1,2), nvl(2,3), nvl(4,3), nvl(5,4) from dual;

NVL (1, 2)	NVL (2, 3)	NVL (4, 3)	NVL (5, 4)
1	2	4	5

SQL> select nvl(0,0), nvl(1,1), nvl(null,null), nvl(4,4) from dual;

NVL (0, 0)	NVL (1, 1)	NVL (null, null)	NVL (4, 4)
0	1		4

F. **POWER**

Power is the ability to raise a value to a given exponent.

Syntax:

power (value, exponent)

Ex:

**SQL> select power(2,5), power(0,0), power(1,1),
power(null,null), power(2,-5) from dual;**

POWER (2, 5)	POWER (0, 0)	POWER (1, 1)
32	1	1
.03125		

G. **EXP**

This will raise e value to the give power.

Syntax:

exp (value)

Ex:

SQL> select exp(1), exp(2), exp(0), exp(null), exp(-2) from dual;

EXP (1)	EXP (2)	EXP (0)	EXP (NULL)
2.71828183	7.3890561	1	
.135335283			

H. **LN**

This is based on natural or base e logarithm.

Syntax:

ln (value) -- here value must be greater than zero which is positive only.

Ex:

SQL> select ln(1), ln(2), ln(null) from dual;

LN (1)	LN (2)	LN (NULL)
0	.693147181	

Ln and Exp are reciprocal to each other.

EXP (3) = 20.0855369

LN (20.0855369) = 3

I. LOG

This is based on 10 based logarithm.

Syntax:

log (10, value) -- here value must be greater than zero which is positive only.

Ex:

**SQL> select log(10,100), log(10,2), log(10,1),
log(10,null) from dual;**

LOG (10, 100)	LOG (10, 2)	LOG (10, 1)
2	.301029996	0

LN (value) = LOG (EXP(1), value)

SQL> select ln(3), log(exp(1),3) from dual;

LN (3)	LOG (EXP (1) , 3)

1.09861229 1.09861229

J. CEIL

This will produce a whole number that is greater than or equal to the specified value.

Syntax:

ceil (value)

Ex:

**SQL> select ceil(5), ceil(5.1), ceil(-5), ceil(-5.1),
ceil(0), ceil(null) from dual;**

CEIL(5)	CEIL(5.1)	CEIL(-5)	CEIL(-5.1)	
CEIL(0)	CEIL(NULL)			
-----	-----	-----	-----	-----
5	6	-5	-5	0

K. FLOOR

This will produce a whole number that is less than or equal to the specified value.

Syntax:

floor (value)

Ex:

SQL> select floor(5), floor(5.1), floor(-5), floor(-5.1), floor(0), floor(null) from dual;

FLOOR(5)	FLOOR(5.1)	FLOOR(-5)	FLOOR(-5.1)	
FLOOR(0)	FLOOR(NULL)			
-----	-----	-----	-----	-----
5	5	-5	-6	0

L. ROUND

This will rounds numbers to a given number of digits of precision.

Syntax:

round (value, precision)

Ex:

**SQL> select round(123.2345), round(123.2345,2),
round(123.2354,2) from dual;**

```
ROUND(123.2345)  ROUND(123.2345,0)
ROUND(123.2345,2) ROUND(123.2354,2)
-----
123              123              123.23
123.24
```

**SQL> select round(123.2345,-1), round(123.2345,-
2), round(123.2345,-3), round(123.2345,-4) from
dual;**

```
ROUND(123.2345,-1) ROUND(123.2345,-2)
ROUND(123.2345,-3) ROUND(123.2345,-4)
-----
120                100                0
0
```

**SQL> select round(123,0), round(123,1),
round(123,2) from dual;**

```
ROUND(123,0)  ROUND(123,1)  ROUND(123,2)
-----
123          123          123
```

SQL> select round(-123,0), round(-123,1), round(-123,2) from dual;

```
ROUND (-123, 0)  ROUND (-123, 1)  ROUND (-123, 2)
-----
-123            -123            -123
```

**SQL> select round(123,-1), round(123,-2),
round(123,-3), round(-123,-1), round(-123,- 2),
round(-123,-3) from dual;**

```
ROUND (123, -1)  ROUND (123, -2)  ROUND (123, -3)
ROUND (-123, -1) ROUND (-123, -2) ROUND (-123, -3)
-----
120             100             0
120             -100            0
```

**SQL> select round(null,null), round(0,0),
round(1,1), round(-1,-1), round(-2,-2) from dual;**

```
ROUND (NULL, NULL)  ROUND (0, 0)  ROUND (1, 1)  ROUND (-1, -1)  ROUND (-2, -2)
-----
0                      0          1          0
```

M. **TRUNC**

This will truncates or chops off digits of precision from a number.

Syntax:

trunc (value, precision)

Ex:

**SQL> select trunc(123.2345), trunc(123.2345,2),
trunc(123.2354,2) from dual;**

```
TRUNC (123.2345)  TRUNC (123.2345,2)
TRUNC (123.2354,2)
-----
123              123.23              123.23
```

**SQL> select trunc(123.2345,-1), trunc(123.2345,-2),
trunc(123.2345,-3), trunc(123.2345,-4) from dual;**

```
TRUNC (123.2345,-1)  TRUNC (123.2345,-2)
TRUNC (123.2345,-3)  TRUNC (123.2345,-4)
-----
120                  100                  0
0
```

**SQL> select trunc(123,0), trunc(123,1),
trunc(123,2) from dual;**

```
TRUNC (123,0)  TRUNC (123,1)  TRUNC (123,2)
-----
123            123            123
```

**SQL> select trunc(-123,0), trunc(-123,1), trunc(-
123,2) from dual;**

```
TRUNC (-123,0)  TRUNC (-123,1)  TRUNC (-123,2)
-----
-123            -123            -123
```

**SQL> select trunc(123,-1), trunc(123,-2),
trunc(123,-3), trunc(-123,-1), trunc(-123,2), trunc(-
123,-3) from dual;**

```
TRUNC(123,-1) TRUNC(123,-2) TRUNC(123,-3)
TRUNC(-123,-1) TRUNC(-123,2) TRUNC(-123,-3)
```

```
-----
-----
120      100      0
-120      -123      0
```

**SQL> select trunc(null,null), trunc(0,0), trunc(1,1),
trunc(-1,-1), trunc(-2,-2) from dual;**

```
TRUNC(NULL,NULL) TRUNC(0,0) TRUNC(1,1) TRUNC(-1,-1) TRUNC(-2,-2)
```

```
-----
-----
0      0      1      0
```

N. **BITAND**

This will perform bitwise and operation.

Syntax:

bitand (value1, value2)

Ex:

**SQL> select bitand(2,3), bitand(0,0), bitand(1,1),
bitand(null,null), bitand(-2,-3) from dual;**

```
BITAND(2,3) BITAND(0,0) BITAND(1,1)
BITAND(NULL,NULL) BITAND(-2,-3)
```

```
-----
-----
2      0      1
-4
```

O. **GREATEST**

This will give the greatest number.

Syntax:

greatest (value1, value2, value3 ... valuen)

Ex:

**SQL> select greatest(1, 2, 3), greatest(-1, -2, -3)
from dual;**

GREATEST (1, 2, 3)	GREATEST (-1, -2, -3)
3	-1

- If all the values are zeros then it will display zero.
- If all the parameters are nulls then it will display nothing.
- If any of the parameters is null it will display nothing.

P. **LEAST**

This will give the least number.

Syntax:

least (value1, value2, value3 ... valuen)

Ex:

**SQL> select least(1, 2, 3), least(-1, -2, -3) from
dual;**

LEAST (1, 2, 3)	LEAST (-1, -2, -3)
1	-3

- If all the values are zeros then it will display zero.
- If all the parameters are nulls then it will display nothing.

- If any of the parameters is null it will display nothing.

Q. COALESCE

This will return first non-null value.

Syntax:

coalesce (value1, value2, value3 ... valuen)

Ex:

**SQL> select coalesce(1,2,3), coalesce(null,2,null,5)
from dual;**

COALESCE (1, 2, 3)	COALESCE (NULL, 2, NULL, 5)
-----	-----
1	2

STRING FUNCTIONS

- Initcap
- Upper
- Lower
- Length
- Rpad
- Lpad
- Ltrim
- Rtrim
- Trim
- Translate
- Replace
- Soundex
- Concat (' || ' Concatenation operator)
- Ascii
- Chr
- Substr

- Instr
- Decode
- Greatest
- Least
- Coalesce

A. **INITCAP**

This will capitalize the initial letter of the string.

Syntax:

initcap (string)

Ex:

SQL> select initcap('computer') from dual;

```
INITCAP
-----
Computer
```

B. **UPPER**

This will convert the string into uppercase.

Syntax:

upper (string)

Ex:

SQL> select upper('computer') from dual;

```
UPPER
-----
COMPUTER
```

C. **LOWER**

This will convert the string into lowercase.

Syntax:

lower (string)

Ex:

SQL> select lower('COMPUTER') from dual;

```
LOWER
-----
computer
```

D. **LENGTH**

This will give length of the string.

Syntax:

length (string)

Ex:

SQL> select length('computer') from dual;

```
LENGTH
-----
      8
```

E. **RPAD**

This will allows you to pad the right side of a column with any set of characters.

Syntax:

rpad (string, length [, padding_char])

Ex:

**SQL> select rpad('computer',15,'*'),
rpadd('computer',15,'*#') from dual;**

```
RPAD ( 'COMPUTER '      RPAD ( 'COMPUTER '
-----
computer*****          computer*#*#*#*
```

-- Default padding character was blank space.

F. LPAD

This will allow you to pad the left side of a column with any set of characters.

Syntax:

lpad (string, length [, padding_char])

Ex:

**SQL> select lpad('computer',15,'*'),
lpad('computer',15,'*#') from dual;**

LPAD ('COMPUTER '	LPAD ('COMPUTER '
-----	-----
*****computer	*#*#*#*computer

-- Default padding character was blank space.

G. LTRIM

This will trim off unwanted characters from the left end of string.

Syntax:

ltrim (string [,unwanted_chars])

Ex:

**SQL> select ltrim('computer','co'),
ltrim('computer','com') from dual;**

LTRIM (LTRIM
-----	-----
mputer	puter

**SQL> select ltrim('computer','puter'),
ltrim('computer','omputer') from dual;**

```
LTRIM('C      LTRIM('C
-----
computer      computer
```

-- If you haven't specify any unwanted characters it will display entire string.

H. **RTRIM**

This will trim off unwanted characters from the right end of string.

Syntax:

rtrim (string [, unwanted_chars])

Ex:

**SQL> select rtrim('computer','er'),
rtrim('computer','ter') from dual;**

```
RTRIM(      RTRIM
-----
comput      compu
```

**SQL> select rtrim('computer','comput'),
rtrim('computer','compute') from dual;**

```
RTRIM('C      RTRIM('C
-----
computer      computer
```

-- If you haven't specify any unwanted characters it will display entire string.

I. **TRIM**

This will trim off unwanted characters from the both sides of string.

Syntax:

trim (unwanted_chars from string)

Ex:

SQL> select trim('i' from 'indiani') from dual;

```
TRIM(
-----
ndian
```

SQL> select trim(leading'i' from 'indiani') from dual; -- this will work as LTRIM

```
TRIM(L
-----
ndiani
```

SQL> select trim(trailing'i' from 'indiani') from dual; -- this will work as RTRIM

```
TRIM(T
-----
Indian
```

J. TRANSLATE

This will replace the set of characters, character by character.

Syntax:

translate (string, old_chars, new_chars)

Ex:

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

```
TRANS
-----
xydxa
```

K. REPLACE

This will replace the set of characters, string by string.

Syntax:

replace (string, old_chars [, new_chars])

Ex:

**SQL> select replace('india','in','xy'),
replace('india','in') from dual;**

REPLACE	REPLACE
-----	-----
Xydia	dia

L. SOUNDEX

This will be used to find words that sound like other words, exclusively used in where clause.

Syntax:

soundex (string)

Ex:

**SQL> select * from emp where soundex(ename) =
soundex('SMIT');**

EMPNO	ENAME	JOB	MGR	HIREDATE
SAL	DEPTNO			
-----	-----	-----	-----	-----
-----	-----			
7369	SMITH	CLERK	7902	17-DEC-80
500	20			

M. CONCAT

This will be used to combine two strings only.

Syntax:

concat (string1, string2)

Ex:

SQL> select concat('computer',' operator') from dual;

```
CONCAT ( 'COMPUTER '
-----
computer operator
```

If you want to combine more than two strings you have to use concatenation operator (||).

SQL> select 'how' || ' are' || ' you' from dual;

```
'HOW' || 'ARE
-----
how are you
```

N. **ASCII**

This will return the decimal representation in the database character set of the first character of the string.

Syntax:

ascii (string)

Ex:

SQL> select ascii('a'), ascii('apple') from dual;

```
ASCII ( 'A ' )   ASCII ( 'APPLE ' )
-----
          97          97
```

O. **CHR**

This will return the character having the binary equivalent to the string in either the database character set or the national character set.

Syntax:

chr (number)

Ex:

SQL> select chr(97) from dual;

```
CHR
-----
a
```

P. SUBSTR

This will be used to extract substrings.

Syntax:

substr (string, start_chr_count [, no_of_chars])

Ex:

**SQL> select substr('computer',2),
substr('computer',2,5), substr('computer',3,7)
from dual;**

```
SUBSTR(      SUBST      SUBSTR
-----      -
computer      omput      mputer
```

- If no_of_chars parameter is negative then it will display nothing.
- If both parameters except string are null or zeros then it will display nothing.
- If no_of_chars parameter is greater than the length of the string then it ignores and calculates based on the original string length.
- If start_chr_count is negative then it will extract the substring from right end.

12	3	4	5	6	7	8	
CO	M	P	U	T	E	R	
-8	-7	-6	-5	-4	-3	-2	-1

Q. INSTR

This will allows you for searching through a string for set of characters.

Syntax:

instr (string, search_str [, start_chr_count [, occurrence]])

Ex:

**SQL> select instr('information','o',4,1),
instr('information','o',4,2) from dual;**

```
INSTR (' INFORMATION' , 'O' , 4 , 1)
INSTR (' INFORMATION' , 'O' , 4 , 2)
-----
-----
4
10
```

- If you are not specifying start_chr_count and occurrence then it will start search from the beginning and finds first occurrence only.
- If both parameters start_chr_count and occurrence are null, it will display nothing.

R. DECODE

Decode will act as value by value substitution. For every value of field, it will checks for a match in a series of if/then tests.

Syntax:

decode (value, if1, then1, if2, then2, else);

Ex:

**SQL> select sal,
decode(sal,500,'Low',5000,'High','Medium') from
emp;**

SAL	DECODE
500	Low
2500	Medium
2000	Medium
3500	Medium
3000	Medium
5000	High
4000	Medium
5000	High
1800	Medium
1200	Medium
2000	Medium
2700	Medium
2200	Medium
3200	Medium

**SQL> select decode(1,1,3), decode(1,2,3,4,4,6) from
dual;**

DECODE (1, 1, 3)	DECODE (1, 2, 3, 4, 4, 6)
3	6

- If the number of parameters are odd and different then decode will display nothing.
- If the number of parameters are even and different then decode will display last value.

- If all the parameters are null then decode will display nothing.
- If all the parameters are zeros then decode will display zero.

S. **GREATEST**

This will give the greatest string.

Syntax:

greatest (strng1, string2, string3 ... stringn)

Ex:

**SQL> select greatest('a', 'b', 'c'),
greatest('satish','srinu','saketh') from dual;**

GREAT	GREAT
-----	-----
c	srinu

- If all the parameters are nulls then it will display nothing.
- If any of the parameters is null it will display nothing.

T. **LEAST**

This will give the least string.

Syntax:

greatest (strng1, string2, string3 ... stringn)

Ex:

**SQL> select least('a', 'b', 'c'),
least('satish','srinu','saketh') from dual;**

LEAST	LEAST
-----	-----
a	saketh

- If all the parameters are nulls then it will display nothing.
- If any of the parameters is null it will display nothing.

U. **COALESCE**

This will gives the first non-null string.

Syntax:

coalesce (strng1, string2, string3 ... stringn)

Ex:

**SQL> select coalesce('a','b','c'),
coalesce(null,'a',null,'b') from dual;**

COALESCE	COALESCE
-----	-----
a	a

DATE FUNCTIONS

- Sysdate
- Current_date
- Current_timestamp
- Systimestamp
- Localtimestamp
- Dbtimezone
- Sessiontimezone
- To_char
- To_date
- Add_months
- Months_between
- Next_day
- Last_day
- Extract

- Greatest
- Least
- Round
- Trunc
- New_time
- Coalesce

Oracle default date format is DD-MON-YY.

We can change the default format to our desired format by using the following command.

SQL> alter session set nls_date_format = 'DD-MONTH-YYYY';

But this will expire once the session was closed.

A. **SYSDATE**

This will give the current date and time.

Ex:

SQL> select sysdate from dual;

```
SYSDATE
-----
24-DEC-06
```

B. **CURRENT_DATE**

This will returns the current date in the session's timezone.

Ex:

SQL> select current_date from dual;

```
CURRENT_DATE
-----
24-DEC-06
```

C. CURRENT_TIMESTAMP

This will returns the current timestamp with the active time zone information.

Ex:

SQL> select current_timestamp from dual;

```
CURRENT_TIMESTAMP
-----
24-DEC-06 03.42.41.383369 AM +05:30
```

D. SYSTIMESTAMP

This will returns the system date, including fractional seconds and time zone of the database.

Ex:

SQL> select systimestamp from dual;

```
SYSTIMESTAMP
-----
24-DEC-06 03.49.31.830099 AM +05:30
```

E. LOCALTIMESTAMP

This will returns local timestamp in the active time zone information, with no time zone information shown.

Ex:

SQL> select localtimestamp from dual;

```
LOCALTIMESTAMP
-----
24-DEC-06 03.44.18.502874 AM
```

F. DBTIMEZONE

This will returns the current database time zone in UTC format. (Coordinated Universal Time)

Ex:

SQL> select dbtimezone from dual;

```
DBTIMEZONE
-----
-07:00
```

G. SESSIONTIMEZONE

This will returns the value of the current session's time zone.

Ex:

SQL> select sessiontimezone from dual;

```
SESSIONTIMEZONE
-----
+05:30
```

H. TO_CHAR

This will be used to extract various date formats. The available date formats as follows.

Syntax:

to_char (date, format)

DATE FORMATS

```
D          --      No of days in week
DD         --      No of days in month
DDD        --      No of days in year
MM         --      No of month
MON        --      Three letter abbreviation of
month
```


MONTH	--	Fully spelled out month
RM	--	Roman numeral month
DY	--	Three letter abbreviated day
DAY	--	Fully spelled out day
Y	--	Last one digit of the year
YY	--	Last two digits of the year
YYY	--	Last three digits of the year
YYYY	--	Full four digit year
SYYYY--		Signed year
I	--	One digit year from ISO
standard		
IY	--	Two digit year from ISO
standard		
IYY	--	Three digit year from ISO
standard		
IYYY	--	Four digit year from ISO
standard		
Y, YYY	--	Year with comma
YEAR	--	Fully spelled out year
CC	--	Century
Q	--	No of quarters
W	--	No of weeks in month
WW	--	No of weeks in year
IW	--	No of weeks in year from ISO
standard		
HH	--	Hours
MI	--	Minutes
SS	--	Seconds
FF	--	Fractional seconds
AM or PM	--	Displays AM or PM depending
upon time of day		
A.M or P.M--		Displays A.M or P.M depending
upon time of day		
AD or BC	--	Displays AD or BC depending
upon the date		
A.D or B.C--		Displays AD or BC depending
upon the date		
FM	--	Prefix to month or day,
suppresses padding of month or day		
TH	--	Suffix to a number

```

SP          --      suffix to a number to be
spelled out
SPTH        --      Suffix combination of TH and
SP to be both spelled out
THSP        --      same as SPTH

```

Ex:

**SQL> select to_char(sysdate,'dd month yyyy
hh:mi:ss am dy') from dual;**

```

TO_CHAR(SYSDATE, 'DD MONTH YYYYHH:MI
-----
24 december 2006 02:03:23 pm sun

```

**SQL> select to_char(sysdate,'dd month year')
from dual;**

```

TO_CHAR(SYSDATE, 'DDMONTHYEAR')
-----
24 december two thousand six

```

**SQL> select to_char(sysdate,'dd fmonth year')
from dual;**

```

TO_CHAR(SYSDATE, 'DD FMMONTH YEAR')
-----
24 december two thousand six

```

**SQL> select to_char(sysdate,'ddth DDTH') from
dual;**

```

TO_CHAR(S
-----
24th 24TH

```

**SQL> select to_char(sysdate,'ddspth DDSPTH')
from dual;**

```

TO_CHAR(SYSDATE, 'DDSPTHDDSPTH

```

```
-----
twenty-fourth TWENTY-FOURTH
```

**SQL> select to_char(sysdate,'ddsp Ddsp DDSP ')
from dual;**

```
TO_CHAR(SYSDATE, 'DDSPDDSPDDSP')
-----
twenty-four Twenty-Four TWENTY-FOUR
```

I. ***TO_DATE***

This will be used to convert the string into data format.

Syntax:

to_date (date)

Ex:

**SQL> select
to_char(to_date('24/dec/2006','dd/mon/yyyy'), 'dd
* month * day') from dual;**

```
TO_CHAR(TO_DATE('24/DEC/20
-----
24 * december * Sunday
```

-- If you are not using to_char oracle will display output in default date format.

J. ***ADD_MONTHS***

This will add the specified months to the given date.

Syntax:

add_months (date, no_of_months)

Ex:

SQL> select add_months(to_date('11-jan-1990','dd-mon-yyyy'), 5) from dual;

```
ADD_MONTHS
-----
11-JUN-90
```

SQL> select add_months(to_date('11-jan-1990','dd-mon-yyyy'), -5) from dual;

```
ADD_MONTH
-----
11-AUG-89
```

- If no_of_months is zero then it will display the same date.
- If no_of_months is null then it will display nothing.

K. ***MONTHS_BETWEEN***

This will give difference of months between two dates.

Syntax:

months_between (date1, date2)

Ex:

SQL> select months_between(to_date('11-aug-1990','dd-mon-yyyy'), to_date('11-jan- 1990','dd-mon-yyyy')) from dual;

```
MONTHS_BETWEEN(TO_DATE('11-AUG-1990','DD-MON-
YYYY'),TO_DATE('11-JAN-1990','DD-MON-YYYY'))
```

SQL> select months_between(to_date('11-jan-1990','dd-mon-yyyy'), to_date('11-aug- 1990','dd-mon-yyyy')) from dual;

```
MONTHS_BETWEEN(TO_DATE('11-JAN-1990','DD-MON-
YYYY'),TO_DATE('11-AUG-1990','DD-MON-YYYY'))
-----
-7
```

L. ***NEXT_DAY***

This will produce next day of the given day from the specified date.

Syntax:

next_day (date, day)

Ex:

SQL> select next_day(to_date('24-dec-2006','dd-mon-yyyy'),'sun') from dual;

```
NEXT_DAY (
-----
31-DEC-06
```

-- If the day parameter is null then it will display nothing.

M. ***LAST_DAY***

This will produce last day of the given date.

Syntax:

last_day (date)

Ex:

SQL> select last_day(to_date('24-dec-2006','dd-mon-yyyy'),'sun') from dual;

```
LAST_DAY (
-----
31-DEC-06
```

N. **EXTRACT**

This is used to extract a portion of the date value.

Syntax:

extract ((year | month | day | hour | minute | second), date)

Ex:

SQL> select extract(year from sysdate) from dual;

```
EXTRACT (YEARFROMSYSDATE)
-----
2006
```

-- You can extract only one value at a time.

O. **GREATEST**

This will give the greatest date.

Syntax:

greatest (date1, date2, date3 ... daten)

Ex:

SQL> select greatest(to_date('11-jan-90','dd-mon-yy'),to_date('11-mar-90','dd-mon-yy'),to_date('11-apr-90','dd-mon-yy')) from dual;

```
GREATEST (
```

```
-----
11-APR-90
```

P. **LEAST**

This will give the least date.

Syntax:

least (date1, date2, date3 ... daten)

Ex:

SQL> select least(to_date('11-jan-90','dd-mon-yy'),to_date('11-mar-90','dd-mon-yy'),to_date('11-apr-90','dd-mon-yy')) from dual;

```
LEAST (
-----
11-JAN-90
```

Q. **ROUND**

Round will rounds the date to which it was equal to or greater than the given date.

Syntax:

round (date, (day | month | year))

If the second parameter was year then round will checks the month of the given date in the following ranges.

```
JAN  --  JUN
JUL  --  DEC
```

If the month falls between JAN and JUN then it returns the first day of the current year. If the month

falls between JUL and DEC then it returns the first day of the next year.

If the second parameter was month then round will checks the day of the given date in the following ranges.

1	--	15
16	--	31

If the day falls between 1 and 15 then it returns the first day of the current month. If the day falls between 16 and 31 then it returns the first day of the next month.

If the second parameter was day then round will checks the week day of the given date in the following ranges.

SUN	--	WED
THU	--	SUN

If the week day falls between SUN and WED then it returns the previous sunday.

If the weekday falls between THU and SUN then it returns the next sunday.

- If the second parameter was null then it returns nothing.
- If the you are not specifying the second parameter then round will resets the time to the begining of the current day in case of user specified date.

- If the you are not specifying the second parameter then round will resets the time to the begining of the next day in case of sysdate.

Ex:

SQL> select round(to_date('24-dec-04','dd-mon-yy'),'year'), round(to_date('11-mar- 06','dd-mon-yy'),'year') from dual;

```
ROUND (TO_    ROUND (TO_
-----
01-JAN-05      01-JAN-06
```

SQL> select round(to_date('11-jan-04','dd-mon-yy'),'month'), round(to_date('18-jan- 04','dd-mon-yy'),'month') from dual;

```
ROUND (TO_    ROUND (TO_
-----
01-JAN-04      01-FEB-04
```

SQL> select round(to_date('26-dec-06','dd-mon-yy'),'day'), round(to_date('29-dec- 06','dd-mon-yy'),'day') from dual;

```
ROUND (TO_    ROUND (TO_
-----
24-DEC-06      31-DEC-06
```

SQL> select to_char(round(to_date('24-dec-06','dd-mon-yy')), 'dd mon yyyy hh:mi:ss am') from dual;

```
TO_CHAR (ROUND (TO_DATE ( '
-----
24 dec 2006 12:00:00 am
```

R. **TRUNC**

Trunc will chop off the date to which it was equal to or less than the given date.

Syntax:

trunc (date, (day | month | year))

- If the second parameter was year then it always returns the first day of the current year.
- If the second parameter was month then it always returns the first day of the current month.
- If the second parameter was day then it always returns the previous sunday.
- If the second parameter was null then it returns nothing.
- If the you are not specifying the second parameter then trunk will resets the time to the begining of the current day.

Ex:

SQL> select trunc(to_date('24-dec-04','dd-mon-yy'),'year'), trunc(to_date('11-mar- 06','dd-mon-yy'),'year') from dual;

```
TRUNC (TO_      TRUNC (TO_
-----
01-JAN-04      01-JAN-06
```

SQL> select trunc(to_date('11-jan-04','dd-mon-yy'),'month'), trunc(to_date('18-jan- 04','dd-mon-yy'),'month') from dual;

```
TRUNC (TO_      TRUNC (TO_
-----
01-JAN-04      01-JAN-04
```

```
SQL> select trunc(to_date('26-dec-06','dd-mon-yy'),'day'), trunc(to_date('29-dec-06','dd-mon-yy'),'day') from dual;
```

```
TRUNC (TO_      TRUNC (TO_
-----
24-DEC-06      24-DEC-06
```

```
SQL> select to_char(trunc(to_date('24-dec-06','dd-mon-yy')), 'dd mon yyyy hh:mi:ss am') from dual;
```

```
TO_CHAR (TRUNC (TO_DATE ( '
-----
24 dec 2006 12:00:00 am
```

S. *NEW_TIME*

This will give the desired timezone's date and time.

Syntax:

new_time (date, current_timezone, desired_timezone)

Available timezones are as follows.

TIMEZONES

```
AST/ADT      -- Atlantic standard/day
light time
BST/BDT      -- Bering standard/day
light time
CST/CDT      -- Central standard/day
light time
EST/EDT      -- Eastern standard/day
light time
GMT          -- Greenwich mean time
HST/HDT      -- Alaska-Hawaii
standard/day light time
MST/MDT      -- Mountain standard/day
light time
```

```

NST      --  Newfoundland standard time
PST/PDT  --  Pacific standard/day
light time
YST/YDT  --  Yukon standard/day light
time

```

Ex:

**SQL> select
to_char(new_time(sysdate,'gmt','yst'),'dd mon
yyyy hh:mi:ss am') from dual;**

```

TO_CHAR(NEW_TIME(SYSDAT
-----
24 dec 2006 02:51:20 pm

```

**SQL> select
to_char(new_time(sysdate,'gmt','est'),'dd mon
yyyy hh:mi:ss am') from dual;**

```

TO_CHAR(NEW_TIME(SYSDAT
-----
24 dec 2006 06:51:26 pm

```

T. COALESCE

This will give the first non-null date.

Syntax:

coalesce (date1, date2, date3 ... daten)

Ex:

**SQL> select coalesce('12-jan-90','13-jan-99'),
coalesce(null,'12-jan-90','23-mar-98',null) from
dual;**

```

COALESCE (      COALESCE (
-----
12-jan-90      12-jan-90

```

MISCELLANEOUS FUNCTIONS

- Uid
- User
- Vsize
- Rank
- Dense_rank

A. **UID**

This will returns the integer value corresponding to the user currently logged in.

Ex:

SQL> select uid from dual;

```

      UID
-----
      319

```

B. **USER**

This will returns the login's user name.

Ex:

SQL> select user from dual;

```

      USER
-----
      SAKETH

```

C. **VSIZE**

This will returns the number of bytes in the expression.

Ex:

**SQL> select vsize(123), vsize('computer'),
vsize('12-jan-90') from dual;**

VSIZE (123) ----- -----	VSIZE ('COMPUTER') -----	VSIZE ('12- JAN-90 ') -----
3	8	9

D. **RANK**

This will give the non-sequential ranking.

Ex:

**SQL> select rownum,sal from (select sal from emp
order by sal desc);**

ROWNUM	SAL
1	5000
2	3000
3	3000
4	2975
5	2850
6	2450
7	1600
8	1500
9	1300
10	1250
11	1250
12	1100
13	1000
14	950
15	800

**SQL> select rank(2975) within group(order by sal
desc) from emp;**

RANK (2975) WITHINGROUP (ORDERBYSALDESC) -----
4

E. ***DENSE_RANK***

This will give the sequential ranking.

Ex:

SQL> select dense_rank(2975) within group(order by sal desc) from emp;

```
DENSE_RANK(2975) WITHIN GROUP (ORDER BY SAL DESC)
```

3

CONVERSION FUNCTIONS

- Bin_to_num
- Chartorowid
- Rowidtochar
- To_number
- To_char
- To_date

A. ***BIN_TO_NUM***

This will convert the binary value to its numerical equivalent.

Syntax:

bin_to_num(binary_bits)

Ex:

SQL> select bin_to_num(1,1,0) from dual;

```
BIN_TO_NUM(1, 1, 0)
```

6

- If all the bits are zero then it produces zero.
- If all the bits are null then it produces an error.

B. *CHARTOROWID*

This will convert a character string to act like an internal oracle row identifier or rowid.

C. *ROWIDTOCHAR*

This will convert an internal oracle row identifier or rowid to character string.

D. *TO_NUMBER*

This will convert a char or varchar to number.

E. *TO_CHAR*

This will convert a number or date to character string.

F. *TO_DATE*

This will convert a number, char or varchar to a date.

GROUP FUNCTIONS

- Sum
- Avg
- Max
- Min
- Count

Group functions will be applied on all the rows but produces single output.

A. *SUM*

This will give the sum of the values of the specified column.

Syntax:

sum (column)

Ex:

SQL> select sum(sal) from emp;

B. *SUM (SAL)*


```
C.      -----
D.      38600
```

E. **AVG**

This will give the average of the values of the specified column.

Syntax:

avg (column)

Ex:

SQL> select avg(sal) from emp;

```
F.      AVG (SAL)
G.      -----
H.      2757.14286
```

I. **MAX**

This will give the maximum of the values of the specified column.

Syntax:

max (column)

Ex:

SQL> select max(sal) from emp;

```
J.      MAX (SAL)
K.      -----
L.      5000
```

M. **MIN**

This will give the minimum of the values of the specified column.

Syntax:

min (column)

Ex:

SQL> select min(sal) from emp;

```
N.      MIN (SAL)
O.      -----
```

P. 500

Q. COUNT

This will give the count of the values of the specified column.

Syntax:

count (column)

Ex:

SQL> select count(sal),count(*) from emp;

R.	COUNT (SAL)	COUNT (*)
S.	-----	-----
T.	14	14

CONSTRAINTS

Constraints are categorized as follows.

- **Domain integrity constraints**
 - Not null
 - Check
- **Entity integrity constraints**
 - Unique
 - Primary key
- **Referential integrity constraints**
 - Foreign key

Constraints are always attached to a column not a table.

We can add constraints in three ways.

- **Column level** -- along with the column definition
- **Table level** -- after the table definition
- **Alter level** -- using alter command

While adding constraints you need not specify the name but the type only, oracle will internally name the constraint.

If you want to give a name to the constraint, you have to use the constraint clause.

NOT NULL

This is used to avoid null values.

We can add this constraint in column level only.

Ex:

```
SQL> create table student(no number(2) not null,  
name varchar(10), marks number(3));
```

```
SQL> create table student(no number(2) constraint nn  
not null, name varchar(10), marks number(3));
```

CHECK

This is used to insert the values based on specified condition.

We can add this constraint in all three levels.

Ex:

COLUMN LEVEL

```
SQL> create table student(no number(2) , name  
varchar(10), marks number(3) check (marks > 300));
```

```
SQL> create table student(no number(2) , name
```

**varchar(10), marks number(3) constraint ch
check(marks > 300));**

TABLE LEVEL

**SQL> create table student(no number(2) , name
varchar(10), marks number(3), check (marks > 300));**

**SQL> create table student(no number(2) , name
varchar(10), marks number(3), constraint ch
check(marks > 300));**

ALTER LEVEL

SQL> alter table student add check(marks>300);

**SQL> alter table student add constraint ch
check(marks>300);**

UNIQUE

This is used to avoid duplicates but it allow nulls.

We can add this constraint in all three levels.

Ex:

COLUMN LEVEL

**SQL> create table student(no number(2) unique, name
varchar(10), marks number(3));**

**SQL> create table student(no number(2) constraint un
unique, name varchar(10), marks number(3));**

TABLE LEVEL

**SQL> create table student(no number(2) , name
varchar(10), marks number(3), unique(no));**

**SQL> create table student(no number(2) , name
varchar(10), marks number(3), constraint un
unique(no));**

ALTER LEVEL

SQL> alter table student add unique(no);

SQL> alter table student add constraint un unique(no);

PRIMARY KEY

This is used to avoid duplicates and nulls. This will work as combination of unique and not null. Primary key always attached to the parent table. We can add this constraint in all three levels.

Ex:

COLUMN LEVEL

**SQL> create table student(no number(2) primary key,
name varchar(10), marks number(3));**

```
SQL> create table student(no number(2) constraint pk  
primary key, name varchar(10), marks number(3));
```

TABLE LEVEL

```
SQL> create table student(no number(2) , name  
varchar(10), marks number(3), primary key(no));
```

```
SQL> create table student(no number(2) , name  
varchar(10), marks number(3), constraint pk primary  
key(no));
```

ALTER LEVEL

```
SQL> alter table student add primary key(no);
```

```
SQL> alter table student add constraint pk primary  
key(no);
```

FOREIGN KEY

This is used to reference the parent table primary key column which allows duplicates. Foreign key always attached to the child table. We can add this constraint in table and alter levels only.

Ex:

TABLE LEVEL

```
SQL> create table emp(empno number(2), ename  
varchar(10), deptno number(2), primary key(empno),
```

foreign key(deptno) references dept(deptno));

SQL> create table emp(empno number(2), ename varchar(10), deptno number(2), constraint pk primary key(empno), constraint fk foreign key(deptno) references dept(deptno));

ALTER LEVEL

SQL> alter table emp add foreign key(deptno) references dept(deptno);

SQL> alter table emp add constraint fk foreign key(deptno) references dept(deptno);

Once the primary key and foreign key relationship has been created then you can not remove any parent record if the dependent child exists.

USING ON DELETE CASCADE

By using this clause you can remove the parent record even if child exists. Because whenever you remove parent record Oracle automatically removes all its dependent records from child table, if this clause is present while creating foreign key constraint.

Ex:

TABLE LEVEL

SQL> create table emp(empno number(2), ename

varchar(10), deptno number(2), primary key(empno), foreign key(deptno) references dept(deptno) on delete cascade);

SQL> create table emp(empno number(2), ename varchar(10), deptno number(2), constraint pk primary key(empno), constraint fk foreign key(deptno) references dept(deptno) on delete cascade);

ALTER LEVEL

SQL> alter table emp add foreign key(deptno) references dept(deptno) on delete cascade;

SQL> alter table emp add constraint fk foreign key(deptno) references dept(deptno) on delete cascade;

COMPOSITE KEYS

A composite key can be defined on a combination of columns. We can define composite keys on entity integrity and referential integrity constraints. Composite key can be defined in table and alter levels only.

Ex:

UNIQUE (TABLE LEVEL)

SQL> create table student(no number(2) , name varchar(10), marks number(3), unique(no,name));

SQL> create table student(no number(2) , name

varchar(10), marks number(3), constraint unique(no,name));

UNIQUE (ALTER LEVEL)

SQL> alter table student add unique(no,name);

SQL> alter table student add constraint unique(no,name);

PRIMARY KEY (TABLE LEVEL)

SQL> create table student(no number(2) , name varchar(10), marks number(3), primary key(no,name));

SQL> create table student(no number(2) , name varchar(10), marks number(3), constraint pk primary key(no,name));

PRIMARY KEY (ALTER LEVEL)

SQL> alter table student add primary key(no,anme);

SQL> alter table student add constraint pk primary key(no,name);

FOREIGN KEY (TABLE LEVEL)

SQL> create table emp(empno number(2), ename

```
varchar(10), deptno number(2), dname varchar(10),  
primary key(empno), foreign key(deptno,dname)  
references dept(deptno,dname));
```

```
SQL> create table emp(empno number(2), ename  
varchar(10), deptno number(2), dname varchar(10),  
constraint pk primary key(empno), constraint fk  
foreign key(deptno,dname) references  
dept(deptno,dname));
```

FOREIGN KEY (ALTER LEVEL)

```
SQL> alter table emp add foreign key(deptno,dname)  
references dept(deptno,dname);
```

```
SQL> alter table emp add constraint fk foreign  
key(deptno,dname) references dept(deptno,dname);
```

DEFERRABLE CONSTRAINTS

Each constraint has two additional attributes to support deferred checking of constraints.

- Deferred initially immediate
- Deferred initially deferred

Deferred initially immediate checks for constraint violation at the time of insert.

Deferred initially deferred checks for constraint violation at the time of commit.

Ex:

```
SQL> create table student(no number(2), name
```

**varchar(10), marks number(3), constraint un
unique(no) deferred initially immediate);**

**SQL> create table student(no number(2), name
varchar(10), marks number(3), constraint un
unique(no) deferred initially deferred);**

**SQL> alter table student add constraint un unique(no)
deferrable initially deferred;**

SQL> set constraints all immediate;

This will enable all the constraints violations at the time of inserting.

SQL> set constraints all deferred;

This will enable all the constraints violations at the time of commit.

OPERATIONS WITH CONSTRAINTS

Possible operations with constraints as follows.

- Enable
- Disable
- Enforce
- Drop

ENABLE

This will enable the constraint. Before enable, the constraint will check the existing data.

Ex:

SQL> alter table student enable constraint un;

DISABLE

This will disable the constraint.

Ex:

SQL> alter table student enable constraint un;

ENFORCE

This will enforce the constraint rather than enable for future inserts or updates.

This will not check for existing data while enforcing data.

Ex:

SQL> alter table student enforce constraint un;

DROP

This will remove the constraint.

Ex:

SQL> alter table student drop constraint un;

Once the table is dropped, constraints automatically will drop.

CASE AND DEFAULT

CASE

Case is similar to decode but easier to understand while going through coding

Ex:

SQL> Select sal, Case sal When 500 then 'low' When 5000 then 'high' Else 'medium' End case From emp;

SAL	CASE
-----	-----
500	low
2500	medium
2000	medium
3500	medium
3000	medium
5000	high
4000	medium
5000	high
1800	medium
1200	medium
2000	medium
2700	medium
2200	medium
3200	medium

DEFAULT

Default can be considered as a substitute behavior of not null constraint when applied to new rows being entered into the table.

When you define a column with the default keyword followed by a value, you are actually telling the database that, on insert if a row was not assigned a value for this column, use the default value that you have specified. Default is applied only during insertion of new rows.

Ex:

SQL> create table student(no number(2) default 11,name varchar(2));

SQL> insert into student values(1,'a');

```
SQL> insert into student(name) values('b');
SQL> select * from student;
```

NO	NAME
1	a
11	b

```
SQL> insert into student values(null, 'c');
SQL> select * from student;
```

NO	NAME
1	a
11	b

C -- Default can not override nulls.

ABSTRACT DATA TYPES

Some times you may want type which holds all types of data including numbers, chars and special characters something like this. You can not achieve this using pre-defined types.

You can define custom types which holds your desired data.

Ex:

Suppose in a table we have address column which holds hno and city information.

We will define a custom type which holds both numeric as well as char data.

CREATING ADT

```
SQL> create type addr as object(hno number(3),city
varchar(10)); /
```

CREATING TABLE BASED ON ADT

```
SQL> create table student(no number(2),name
varchar(2),address addr);
```

INSERTING DATA INTO ADT TABLES

```
SQL> insert into student values(1,'a',addr(111,'hyd'));
SQL> insert into student values(2,'b',addr(222,'bang'));
SQL> insert into student values(3,'c',addr(333,'delhi'));
```

SELECTING DATA FROM ADT TABLES

```
SQL> select * from student;
```

NO	NAME	ADDRESS (HNO, CITY)
---	----	-----
1	a	ADDR (111, 'hyd')
2	b	ADDR (222, 'bang')
3	c	ADDR (333, 'delhi')

```
SQL> select no,name,s.address.hno,s.address.city from
student s;
```

NO	NAME	ADDRESS.HNO	ADDRESS.CITY
--	----	-----	-----
1	a	111	hyd
2	b	222	bang
3	c	333	delhi

UPDATE WITH ADT TABLES

```
SQL> update student s set s.address.city = 'bombay' where
s.address.hno = 333;
```

SQL> select no,name,s.address.hno,s.address.city from student s;

NO	NAME	ADDRESS.HNO	ADDRESS.CITY
1	a	111	hyd
2	b	222	bang
3	c	333	bombay

DELETE WITH ADT TABLES

SQL> delete student s where s.address.hno = 111;

SQL> select no,name,s.address.hno,s.address.city from student s;

NO	NAME	ADDRESS.HNO	ADDRESS.CITY
2	b	222	bang
3	c	333	bombay

DROPPING ADT

SQL> drop type addr;

OBJECT VIEWS AND METHODS

OBJECT VIEWS

If you want to implement objects with the existing table, object views come into picture.

You define the object and create a view which relates this object to the existing table nothing but object view.

Object views are used to relate the user defined objects to the existing table.

Ex:

1. Assume that the table student has already been created with the following columns

```
SQL> create table student(no number(2),name  
varchar(10),hno number(3),city varchar(10));
```

2. Create the following types

```
SQL> create type addr as object(hno  
number(2),city varchar(10));/
```

```
SQL> create type stud as object(name  
varchar(10),address addr);/
```

3. Relate the objects to the student table by creating the object view

```
SQL> create view student_ov(no,stud_info) as  
select no,stud(name,addr(hno,city)) from student;
```

4. Now you can insert data into student table in two ways

- a. By regular insert

```
SQL> Insert into student  
values(1,'sudha',111,'hyd');
```

- b. By using object view

```
SQL> Insert into student_ov  
values(1,stud('sudha',addr(111,'hyd')));
```

METHODS

You can define methods which are nothing but functions in types and apply in the tables which holds the types;

Ex:

1. **Defining methods in types**

```
SQL> Create type stud as object(name
```

```
varchar(10),marks number(3), Member function
makrs_f(marks in number) return number, Pragma
restrict_references(marks_f,wnds,rnds,wnps,fnps));/
```

2. Defining type body

```
SQL> Create type body stud as Member function
marks_f(marks in number) return number is Begin
Return (marks+100); End marks_f; End;/
```

3. Create a table using stud type

```
SQL> Create table student(no number(2),info stud);
```

4. Insert some data into student table

```
SQL> Insert into student values(1,stud('sudha',100));
```

5. Using method in select

```
SQL> Select s.info.marks_f(s.info.marks) from
student s;
```

-- Here we are using the pragma restrict_references to avoid the writes to the database.

VARRAYS AND NESTED TABLES

VARRAYS

A varying array allows you to store repeating attributes of a record in a single row but with limit.

Ex:

1. We can create varrays using oracle types as well as user defined types.

- a. Varray using pre-defined types

```
SQL> Create type va as varray(5) of
varchar(10);/
```

- b. Varrays using user defined types

```
SQL> Create type addr as object(hno
```

```
number(3),city varchar(10));/
```

```
SQL> Create type va as varray(5) of addr; /
```

2. Using varray in table

```
SQL> Create table student(no number(2),name  
varchar(10),address va);
```

3. Inserting values into varray table

```
SQL> Insert into student  
values(1,'sudha',va(addr(111,'hyd')));
```

```
SQL> Insert into student  
values(2,'jagan',va(addr(111,'hyd'),addr(222,'bang')));
```

4. Selecting data from varray table

```
SQL> Select * from student;
```

-- This will display varray column data along with
varray and adt;

```
SQL> Select no,name, s.* from student s1,  
table(s1.address) s; -- This will display in general  
format
```

5. Instead of s.* you can specify the columns in varray

```
SQL> Select no,name, s.hno,s.city from student  
s1,table(s1.address) s; -- Update and delete not  
possible in varrays.
```

-- Here we used table function which will take the
varray column as input for producing output
excluding varray and types.

NESTED TABLES

A nested table is, as its name implies, a table within a table.

In this case it is a table that is represented as a column within another table.

Nested table has the same effect of varrays but has no

limit.

Ex:

1. We can create nested tables using oracle types and user defined types which has no limit
 - a. Nested tables using pre-defined types
SQL> Create type nt as table of varchar(10);/
 - b. Nested tables using user defined types
SQL> Create type addr as object(hno number(3),city varchar(10));/
SQL> Create type nt as table of addr;/
2. Using nested table in table
SQL> Create table student(no number(2),name varchar(10),address nt) nested table address store as student_temp;
3. Inserting values into table which has nested table
SQL> Insert into student values (1,'sudha',nt(addr(111,'hyd')));
SQL> Insert into student values (2,'jagan',nt(addr(111,'hyd'),addr(222,'bang')));
4. Selecting data from table which has nested table
SQL> Select * from student;
-- This will display nested table column data along with nested table and adt;
SQL> Select no,name, s.* from student s1, table(s1.address) s;
-- This will display in general format
5. Instead of s.* you can specify the columns in nested table
SQL> Select no,name, s.hno,s.city from student s1,table(s1.address) s;

6. Inserting nested table data to the existing row

SQL> Insert into table(select address from student where no=1) values(addr(555,'chennai'));

7. Update in nested tables

SQL> Update table(select address from student where no=2) s set s.city='bombay' where s.hno = 222;

8. Delete in nested table

SQL> Delete table(select address from student where no=3) s where s.hno=333;

DATA MODEL

- ALL_COLL_TYPES
- ALL_TYPES
- DBA_COLL_TYPES
- DBA_TYPES
- USER_COLL_TYPES
- USER_TYPES

FLASHBACK QUERY

Used to retrieve the data which has been already committed with out going for recovery. Flashbacks are of two types

- Time base flashback
- SCN based flashback (SCN stands for System Change Number)

Ex:

1. Using time based flashback

- a. SQL> Select *from student;
-- This will display all the rows
- b. SQL> Delete student;
- c. SQL> Commit; -- this will commit the work.
- d. SQL> Select *from student;
-- Here it will display nothing
- e. Then execute the following procedures
SQL> Exec
dbms_flashback.enable_at_time(sysdate-2/1440)
- f. SQL> Select *from student;
-- Here it will display the lost data
-- The lost data will come but the current system time was used
- g. SQL> Exec dbms_flashback.disable
-- Here we have to disable the flashback to enable it again

2. **Using SCN based flashback**

- a. Declare a variable to store SCN
SQL> Variable s number
- b. Get the SCN
SQL> Exec :s := exec
dbms_flashback.get_system_change_number
- c. To see the SCN
SQL> Print s
- d. Then execute the following procedures
SQL> Exec
dbms_flashback.enable_at_system_change_number(:s)
SQL> Exec dbms_flashback.disable

EXTERNAL TABLES

- You can use external table feature to access external files as if they are tables inside the database.
- When you create an external table, you define its structure and location within Oracle.
- When you query the table, Oracle reads the external table and returns the results just as if the data had been stored within the database.

ACCESSING EXTERNAL TABLE DATA

- To access external files from within Oracle, you must first use the create directory command to define a directory object pointing to the external file location
- who will access the external files must have the read and write privilege on the directory.

Ex:

CREATING DIRECTORY AND OS LEVEL FILE

```
SQL>
SQLplus system/manager
SQL> Create directory saketh_dir as
'/Visdb/visdb/9.2.0/external';
SQL> Grant all on directory saketh_dir to saketh;
SQL> Conn saketh/saketh
SQL> Spool dept.lst
SQL> Select deptno || ',' || dname || ',' || loc from dept;
SQL> Spool off
```

CREATING EXTERNAL TABLE

SQL> Create table dept_ext (deptno number(2), Dname varchar(14), Loc varchar(13)) Organization external (type oracle_loader Default directory saketh_dir Access parameters (records delimited by newline Fields terminated by "," (deptno number(2), Dname varchar(14), Loc varchar(13))) Location ('/Visdb/visdb/9.2.0/dept.lst'));

SELECTING DATA FROM EXTERNAL TABLE

SQL> select * from dept_ext;

This will read from dept.lst which is a operating system level file.

LIMITATIONS ON EXTERNAL TABLES

- a. You can not perform insert, update, and delete operations
- b. Indexing not possible
- c. Constraints not possible

BENEFITS OF EXTERNAL TABLES

- a. Queries of external tables complete very quickly even though a full table scan is required with each access
- b. You can join external tables to each other or to standard tables

REF DEREF VALUE

REF

- The ref function allows referencing of existing row objects.

- Each of the row objects has an object id value assigned to it.
- The object id assigned can be seen by using ref function.

DEREF

- The deref function performs opposite action.
- It takes a reference value of object id and returns the value of the row objects.

VALUE

- Even though the primary table is object table, still it displays the rows in general format.
- To display the entire structure of the object, this will be used.

Ex:

1. create vendot_adt type
SQL> Create type vendor_adt as object (vendor_code number(2), vendor_name varchar(2), vendor_address varchar(10));/
2. create object tables vendors and vendors1
SQL> Create table vendors of vendor_adt;
SQL> Create table vendors1 of vendor_adt;
3. insert the data into object tables
SQL> insert into vendors values(1, 'a', 'hyd');
SQL> insert into vendors values(2, 'b', 'bang');
SQL> insert into vendors1 values(3, 'c', 'delhi');
SQL> insert into vendors1 values(4, 'd', 'chennai');

4. create another table orders which holds the vendor_adt type also.

SQL> Create table orders (order_no number(2),
vendor_info ref vendor_adt); Or

SQL> Create table orders (order_no number(2),
vendor_info ref vendor_adt with rowid);

5. insert the data into orders table

The vendor_info column in the following syntaxes will store object id of any table which is referenced by vendor_adt object (both vendors and vendors1).

SQL> insert into orders values(11,(select ref(v) from vendors v where vendor_code = 1));

SQL> insert into orders values(12,(select ref(v) from vendors v where vendor_code = 2));

SQL> insert into orders values(13,(select ref(v1) from vendors1 v1 where vendor_code = 1));

SQL> insert into orders values(14,(select ref(v1) from vendors1 v1 where vendor_code = 1));

6. To see the object ids of vendor table

SQL> Select ref(V) from vendors v;

7. If you see the vendor_info of orders it will show only the object ids not the values, to see the values

SQL> Select deref(o.vendor_info) from orders o;

8. Even though the vendors table is object table it will not show the adt along with data, to see the data along with the adt

SQL>Select * from vendors;

This will give the data without adt.

SQL>Select value(v) from vendors v;

This will give the columns data along with the type.

REF CONSTRAINTS

- Ref can also acts as constraint.
- Even though vendors1 also holding vendor_adt, the orders table will store the object ids of vendors only because it is constrained to that table only.
- The vendor_info column in the following syntaxes will store object ids of vendors only.

SQL> Create table orders (order_no number(2),
vendor_info ref vendor_adt scope is vendors);

Or

SQL> Create table orders (order_no number(2),
vendor_info ref vendor_adt constraint fk references
vendors);

OBJECT VIEWS WITH REFERENCES

- To implement the objects and the ref constraints to the existing tables, what we can do? Simply drop the both tables and recreate with objects and ref constrains.
- But you can achieve this with out dropping the tables and without losing the data by creating object views with references.

Ex:

- **Create the following tables**

SQL> Create table student1(no number(2) primary
key,name varchar(2),marks number(3));

SQL> Create table student2(no number(2) primary
key,hno number(3),city varchar(10),id
number(2),foreign Key(id) references student1(no));

- **Insert the records into both tables**
 SQL> insert into student1(1,'a',100);
 SQL> insert into student1(2,'b',200);
 SQL> insert into student2(11,111,'hyd',1);
 SQL> insert into student2(12,222,'bang',2);
 SQL> insert into student2(13,333,'bombay',1);
- **Create the type**
 SQL> create or replace type stud as object(no
 number(2),name varchar(2),marks number(3));/
- **Generating OIDs**
 SQL> Create or replace view student1_ov of stud
 with object identifier(or id) (no) as Select * from
 Student1;
- **Generating references**
 SQL> Create or replace view student2_ov as select
 no,hno,city,make_ref(student1_ov,id) id from
 Student2;
- **Query the following**
 SQL> select *from student1_ov;
 SQL> select ref(s) from student1_ov s;
 SQL> select values(s) from student1_ov;
 SQL> select *from student2_ov;
 SQL> select deref(s.id) from student2_ov s;

PARTITIONS

- A single logical table can be split into a number of physically separate pieces based on ranges of key values. Each of the parts of the table is called a partition.
- A non-partitioned table can not be partitioned later.

TYPES

- Range partitions
- List partitions
- Hash partitions
- Sub partitions

ADVANTAGES

- Reducing downtime for scheduled maintenance, which allows maintenance operations to be carried out on selected partitions while other partitions are available to users.
- Reducing downtime due to data failure, failure of a particular partition will no way affect other partitions.
- Partition independence allows for concurrent use of the various partitions for various purposes.

ADVANTAGES OF PARTITIONS BY STORING THEM IN DIFFERENT TABLESPACES

- Reduces the possibility of data corruption in multiple partitions.
- Back up and recovery of each partition can be done independently.

DISADVANTAGES

- Partitioned tables cannot contain any columns with long or long raw datatypes, LOB types or object types.

RANGE PARTITIONS

- a. Creating range partitioned table
SQL> Create table student(no number(2),name

varchar(2)) partition by range(no) (partition p1 values less than(10), partition p2 values less than(20), partition p3 values less than(30),partition p4 values less than(maxvalue));

** if you are using maxvalue for the last partition, you can not add a partition.

b. Inserting records into range partitioned table

SQL> Insert into student values(1,'a'); -- this will go to p1

SQL> Insert into student values(11,'b'); -- this will go to p2

SQL> Insert into student values(21,'c'); -- this will go to p3

SQL> Insert into student values(31,'d'); -- this will go to p4

c. Retrieving records from range partitioned table

SQL> Select *from student;

SQL> Select *from student partition(p1);

d. Possible operations with range partitions

- Add
- Drop
- Truncate
- Rename
- Split
- Move
- Exchange

e. Adding a partition

SQL> Alter table student add partition p5 values less than(40);

f. Dropping a partition

SQL> Alter table student drop partition p4;

g. Renaming a partition

```
SQL> Alter table student rename partition p3 to p6;
```

h. Truncate a partition

```
SQL> Alter table student truncate partition p6;
```

i. Splitting a partition

```
SQL> Alter table student split partition p2 at(15) into
(partition p21,partition p22);
```

j. Exchanging a partition

```
SQL> Alter table student exchange partition p1 with
table student2;
```

k. Moving a partition

```
SQL> Alter table student move partition p21
tablespace saketh_ts;
```

LIST PARTITIONS

1. Creating list partitioned table

```
SQL> Create table student(no number(2),name
varchar(2)) partition by list(no) (partition p1
values(1,2,3,4,5), partition p2
values(6,7,8,9,10),partition p3 values(11,12,13,14,15),
partition p4 values(16,17,18,19,20));
```

2. Inserting records into list partitioned table

```
SQL> Insert into student values(1,'a'); -- this will go
to p1
```

```
SQL> Insert into student values(6,'b'); -- this will go
to p2
```

```
SQL> Insert into student values(11,'c'); -- this will go
to p3
```

```
SQL> Insert into student values(16,'d'); -- this will go
to p4
```

3. Retrieving records from list partitioned table
SQL> Select *from student;
SQL> Select *from student partition(p1);
4. Possible operations with list partitions
 - Add
 - Drop
 - Truncate
 - Rename
 - Move
 - Exchange
5. Adding a partition
SQL> Alter table student add partition p5
values(21,22,23,24,25);
6. Dropping a partition
SQL> Alter table student drop partition p4;
7. Renaming a partition
SQL> Alter table student rename partition p3 to p6;
8. Truncate a partition
SQL> Alter table student truncate partition p6;
9. Exchanging a partition
SQL> Alter table student exchange partition p1 with
table student2;
10. Moving a partition
SQL> Alter table student move partition p2
tablespace saketh_ts;

HASH PARTITIONS

- a. Creating hash partitioned table
SQL> Create table student(no number(2),name
varchar(2)) partition by hash(no) partitions 5;
Here oracle automatically gives partition names like

SYS_P1
SYS_P2
SYS_P3
SYS_P4
SYS_P5

- b. Inserting records into hash partitioned table it will insert the records based on hash function calculated by taking the partition key

```
SQL> Insert into student values(1,'a');  
SQL> Insert into student values(6,'b');  
SQL> Insert into student values(11,'c');  
SQL> Insert into student values(16,'d');
```

- c. Retrieving records from hash partitioned table

```
SQL> Select *from student;  
SQL> Select *from student partition(sys_p1);
```

- d. Possible operations with hash partitions

- Add
- Truncate
- Rename
- Move
- Exchange

- e. Adding a partition

```
SQL> Alter table student add partition p6 ;
```

- f. Renaming a partition

```
SQL> Alter table student rename partition p6 to p7;
```

- g. Truncate a partition

```
SQL> Alter table student truncate partition p7;
```

- h. Exchanging a partition

```
SQL> Alter table student exchange partition sys_p1  
with table student2;
```

i. Moving a partition

SQL> Alter table student move partition sys_p2
tablespace saketh_ts;

SUB-PARTITIONS WITH RANGE AND HASH

Subpartitions clause is used by hash only.

We can not create subpartitions with list and hash partitions.

1. Creating subpartitioned table

SQL> Create table student(no number(2),name
varchar(2),marks number(3)) Partition by range(no)
subpartition by hash(name) subpartitions 3 (Partition
p1 values less than(10),partition p2 values less
than(20));

This will create two partitions p1 and p2 with three
subpartitions for each partition

```
P1 -   SYS_SUBP1
      SYS_SUBP2
      SYS_SUBP3
P2 -   SYS_SUBP4
      SYS_SUBP5
      SYS_SUBP6
```

** if you are using maxvalue for the last partition, you
can not add a partition.

2. Inserting records into subpartitioned table

SQL> Insert into student values(1,'a'); -- this will go
to p1

SQL> Insert into student values(11,'b'); -- this will go
to p2

3. Retrieving records from subpartitioned table

SQL> Select *from student;

SQL> Select *from student partition(p1);

SQL> Select *from student subpartition(sys_subp1);

4. Possible operations with subpartitions

- Add
- Drop
- Truncate
- Rename
- Split

5. Adding a partition

SQL> Alter table student add partition p3 values less than(30);

6. Dropping a partition

SQL> Alter table student drop partition p3;

7. Renaming a partition

SQL> Alter table student rename partition p2 to p3;

8. Truncate a partition

SQL> Alter table student truncate partition p1;

9. Splitting a partition

SQL> Alter table student split partition p3 at(15) into (partition p31,partition p32);

DATA MODEL

- ALL_IND_PARTITIONS
- ALL_IND_SUBPARTITIONS
- ALL_TAB_PARTITIONS
- ALL_TAB_SUBPARTITIONS
- DBA_IND_PARTITIONS
- DBA_IND_SUBPARTITIONS
- DBA_TAB_PARTITIONS

- DBA_TAB_SUBPARTITIONS
- USER_IND_PARTITIONS
- USER_IND_SUBPARTITIONS
- USER_TAB_PARTITIONS
- USER_TAB_SUBPARTITIONS

GROUP BY AND HAVING

GROUP BY

- Using group by, we can create groups of related information.
- Columns used in select must be used with group by, otherwise it was not a group by expression.

Ex:

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

DEPTNO	SUM (SAL)
10	8750
20	10875
30	9400

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

DEPTNO	JOB	SUM (SAL)
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
20	ANALYST	6000
20	CLERK	1900
20	MANAGER	2975

30	CLERK	950
30	MANAGER	2850
30	SALESMAN	5600

HAVING

This will work as where clause which can be used only with group by because of absence of where clause in group by.

Ex:

SQL> select deptno,job,sum(sal) tsal from emp group by deptno,job having sum(sal) > 3000;

DEPTNO	JOB	TSAL
-----	-----	-----
10	PRESIDENT	5000
20	ANALYST	6000
30	SALESMAN	5600

SQL> select deptno,job,sum(sal) tsal from emp group by deptno,job having sum(sal) > 3000 order by job;

DEPTNO	JOB	TSAL
-----	-----	-----
20	ANALYST	6000
10	PRESIDENT	5000
30	SALESMAN	5600

ORDER OF EXECUTION

- Group the rows together based on group by clause.
- Calculate the group functions for each group.
- Choose and eliminate the groups based on the having clause.
- Order the groups based on the specified column.

ROLLUP GROUPING CUBE

These are the enhancements to the group by feature.

USING ROLLUP

This will give the salaries in each department in each job category along with the total salary for individual departments and the total salary of all the departments.

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

DEPTNO	JOB	SUM (SAL)
10	CLERK	1300
10	MANAGER	2450
10	PRESIDENT	5000
10		8750
20	ANALYST	6000
20	CLERK	1900
20	MANAGER	2975
20		10875
30	CLERK	950
30	MANAGER	2850
30	SALESMAN	5600
30		9400
		29025

USING GROUPING

In the above query it will give the total salary of the individual departments but with a blank in the job column and gives the total salary of all the departments with blanks in deptno and job columns.

To replace these blanks with your desired string grouping will be used

SQL> select decode(grouping(deptno),1,'All

```

Depts',deptno),decode(grouping(job),1,'All
jobs',job),sum(sal) from emp group by rollup(deptno,job);
DECODE (GROUPING (DEPTNO) , 1, 'ALLDEPTS ', DEP DECODE (GR
SUM (SAL)

```

```

-----
-----
10                                CLERK
      1300
10                                MANAGER
      2450
10                                PRESIDENT
      5000
10                                All jobs
8750
20                                ANALYST
      6000
20                                CLERK
      1900
20                                MANAGER
      2975
20                                All jobs
10875
30                                CLERK
      950
30                                MANAGER
      2850
30                                SALESMAN
      5600
30                                All jobs
9400
All Depts                        All jobs
29025

```

- Grouping will return 1 if the column which is specified in the grouping function has been used in rollup.
- Grouping will be used in association with decode.

USING CUBE

This will give the salaries in each department in each job category, the total salary for individual departments, the total salary of all the departments and the salaries in each job category.

```
SQL> select decode(grouping(deptno),1,'All
Depts',deptno),decode(grouping(job),1,'All
Jobs',job),sum(sal) from emp group by cube(deptno,job);
```

```
DECODE (GROUPING (DEPTNO) , 1, 'ALLDEPTS' , DEP DECODE (GR
SUM (SAL)
-----
-----
10 CLERK
1300
10 MANAGER
2450
10 PRESIDENT
5000
10 All Jobs
8750
20 ANALYST
6000
20 CLERK
1900
20 MANAGER
2975
20 All Jobs
10875
30 CLERK
950
30 MANAGER
2850
30 SALESMAN
5600
30 All Jobs
9400
All Depts ANALYST
6000
All Depts CLERK
4150
```


All Depts	MANAGER
8275	
All Depts	PRESIDENT
5000	
All Depts	SALESMAN
5600	
All Depts	All Jobs
29025	

SET OPERATORS

TYPES

- Union
- Union all
- Intersect
- Minus

UNION

This will combine the records of multiple tables having the same structure.

Ex:

SQL> select * from student1 union select * from student2;

UNION ALL

This will combine the records of multiple tables having the same structure but including duplicates.

Ex:

SQL> select * from student1 union all select * from student2;

INTERSECT

This will give the common records of multiple tables having the same structure.

Ex:

```
SQL> select * from student1 intersect select * from student2;
```

MINUS

This will give the records of a table whose records are not in other tables having the same structure.

Ex:

```
SQL> select * from student1 minus select * from student2;
```

VIEWS

- A view is a database object that is a logical representation of a table. It is delivered from a table but has no storage of its own and often may be used in the same manner as a table.
- A view takes the output of the query and treats it as a table, therefore a view can be thought of as a stored query or a virtual table.

TYPES

- Simple view
- Complex view

Simple view can be created from one table where as complex view can be created from multiple tables.

WHY VIEWS?

- Provides additional level of security by restricting access to a predetermined set of rows and/or columns of a table.
- Hide the data complexity.
- Simplify commands for the user.

VIEWS WITHOUT DML

- Read only view
- View with group by
- View with aggregate functions
- View with rownum
- Partition view
- View with distinct

Ex:

```
SQL> Create view dept_v as select *from dept with read only;
```

```
SQL> Create view dept_v as select deptno, sum(sal) t_sal from emp group by deptno;
```

```
SQL> Create view stud as select rownum no, name, marks from student;
```

```
SQL> Create view student as select *from student1 union select *from student2;
```

```
SQL> Create view stud as select distinct no,name from student;
```

VIEWS WITH DML

- View with not null column
 - insert with out not null column not possible
 - update not null column to null is not possible
 - delete possible
- View with out not null column which was in base table
 - insert not possible
 - update, delete possible
- View with expression
 - insert , update not possible
 - delete possible
- View with functions (except aggregate)
 - insert, update not possible
 - delete possible
- View was created but the underlying table was dropped then we will get the message like " **view has errors** ".
- View was created but the base table has been altered but still the view was with the initial definition, we have to replace the view to affect the changes.
- Complex view (view with more than one table)
 - insert not possible
 - update, delete possible (not always)

CREATING VIEW WITHOUT HAVING THE BASE TABLE

SQL> Create force view stud as select *From student;
 -- Once the base table was created then the view is validated.

VIEW WITH CHECK OPTION CONSTRAINT

SQL> Create view stud as select *from student where marks = 500 with check option constraint Ck;

- Insert possible with marks value as 500
- Update possible excluding marks column
- Delete possible

DROPPING VIEWS

SQL> drop view dept_v;

SYNONYM AND SEQUENCE

SYNONYM

A synonym is a database object, which is used as an alias for a table, view or sequence.

TYPES

- Private
- Public
- Private synonym is available to the particular user who creates.
- Public synonym is created by DBA which is available to all the users.

ADVANTAGES

- Hide the name and owner of the object.

- Provides location transparency for remote objects of a distributed database.

CREATE AND DROP

```
SQL> create synonym s1 for emp;  
SQL> create public synonym s2 for emp;  
SQL> drop synonym s1;
```

SEQUENCE

- A sequence is a database object, which can generate unique, sequential integer values.
- It can be used to automatically generate primary key or unique key values.
- A sequence can be either in an ascending or descending order.

Syntax:

Create sequence <seq_name> [increment by n] [start with n] [maxvalue n] [minvalue n] [cycle/nocycle] [cache/nocache];

By default the sequence starts with 1, increments by 1 with minvalue of 1 and with nocycle, nocache.

Cache option pre-allocates a set of sequence numbers and retains them in memory for faster access.

Ex:

```
SQL> create sequence s;  
SQL> create sequence s increment by 10 start with 100  
minvalue 5 maxvalue 200 cycle cache 20;
```

USING SEQUENCE

```
SQL> create table student(no number(2),name  
varchar(10));  
SQL> insert into student values(s.nextval, 'saketh');
```

- Initially currval is not defined and nextval is starting value.
- After that nextval and currval are always equal.

CREATING ALPHA-NUMERIC SEQUENCE

```
SQL> create sequence s start with 111234;  
SQL> Insert into student values (s.nextval || translate  
(s.nextval,'1234567890','abcdefghij'));
```

ALTERING SEQUENCE

We can alter the sequence to perform the following.

- Set or eliminate minvalue or maxvalue.
- Change the increment value.
- Change the number of cached sequence numbers.

Ex:

```
SQL> alter sequence s minvalue 5;
```

```
SQL> alter sequence s increment by 2;  
SQL> alter sequence s cache 10;
```

DROPPING SEQUENCE

```
SQL> drop sequence s;
```

JOINS

- The purpose of a join is to combine the data across tables.
- A join is actually performed by the where clause which combines the specified rows of tables.
- If a join involves in more than two tables then oracle joins first two tables based on the joins condition and then compares the result with the next table and so on.

TYPES

- Equi join
- Non-equi join
- Self join
- Natural join
- Cross join
- Outer join
- Left outer
- Right outer
- Full outer
- Inner join
- Using clause
- On clause

Assume that we have the following tables.

SQL> select * from dept;

DEPTNO	DNAME	LOC
10	mkt	hyd
20	fin	bang
30	hr	bombay

SQL> select * from emp;

DEPTNO	EMPNO	ENAME	JOB	MGR
10	111	saketh	analyst	444
20	222	sudha	clerk	333
10	333	jagan	manager	111
40	444	madhu	engineer	222

EQUI JOIN

A join which contains an '=' operator in the joins condition.

Ex:

SQL> select empno,ename,job,dname,loc from emp e,dept d where e.deptno=d.deptno;

EMPNO	ENAME	JOB	DNAME	LOC
---	---	---	---	---

	111	saketh	analyst	mkt
hyd				
	333	jagan	manager	mkt
hyd				
	222	sudha	clerk	fin
bang				

USING CLAUSE

**SQL> select empno,ename,job ,dname,loc from emp e
join dept d using(deptno);**

	EMPNO	ENAME	JOB	DNAME	LOC
----	-----	-----	-----	-----	-----
	111	saketh	analyst	mkt	
hyd					
	333	jagan	manager	mkt	
hyd					
	222	sudha	clerk	fin	
bang					

ON CLAUSE

**SQL> select empno,ename,job,dname,loc from emp e
join dept d on(e.deptno=d.deptno);**

	EMPNO	ENAME	JOB	DNAME	LOC
----	-----	-----	-----	-----	-----
	111	saketh	analyst	mkt	
hyd					
	333	jagan	manager	mkt	
hyd					
	222	sudha	clerk	fin	
bang					

NON-EQUI JOIN

A join which contains an operator other than '=' in the joins condition.

Ex:

SQL> select empno,ename,job,dname,loc from emp e,dept d where e.deptno > d.deptno;

EMPNO	ENAME	JOB	DNAME	LOC
222	sudha	clerk	mkt	hyd
444	madhu	engineer	mkt	hyd
444	madhu	engineer	fin	bang
444	madhu	engineer	hr	bombay

SELF JOIN

Joining the table itself is called self join.

Ex:

SQL> select e1.empno,e2.ename,e1.job,e2.deptno from emp e1,emp e2 where e1.empno=e2.mgr;

EMPNO	ENAME	JOB	DEPTNO
111	jagan	analyst	10
222	madhu	clerk	40
333	sudha	manager	20
444	saketh	engineer	10

NATURAL JOIN

Natural join compares all the common columns.

Ex:

**SQL> select empno,ename,job,dname,loc from emp
natural join dept;**

	EMPNO	ENAME	JOB	DNAME	
LOC	-----	-----	-----	-----	---
	111	saketh	analyst	mkt	
hyd	333	jagan	manager	mkt	
hyd	222	sudha	clerk	fin	
bang					

CROSS JOIN

This will gives the cross product.

Ex:

**SQL> select empno,ename,job,dname,loc from emp
cross join dept;**

	EMPNO	ENAME	JOB	DNAME	LOC
--	-----	-----	-----	-----	---
	111	saketh	analyst	mkt	
hyd	222	sudha	clerk	mkt	
hyd	333	jagan	manager	mkt	
hyd	444	madhu	engineer	mkt	
	111	saketh	analyst	fin	
bang	222	sudha	clerk	fin	
bang					

333	jagan	manager	fin
bang	444	madhu	engineer
bang	111	saketh	analyst
bombay	222	sudha	clerk
bombay	333	jagan	manager
bombay	444	madhu	engineer
bombay			

OUTER JOIN

Outer join gives the non-matching records along with matching records.

LEFT OUTER JOIN

This will display the all matching records and the records which are in left hand side table those that are not in right hand side table.

Ex:

**SQL> select empno,ename,job,dname,loc from emp e
left outer join dept d on(e.deptno=d.deptno);**

Or

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

EMPNO	ENAME	JOB	DNAME	LOC
111	saketh	analyst	mkt	

hyd	333	jagan	manager	mkt
bang	222	sudha	clerk	fin
	444	madhu	engineer	

RIGHT OUTER JOIN

This will display the all matching records and the records which are in right hand side table those that are not in left hand side table.

Ex:

**SQL> select empno,ename,job,dname,loc from emp e
right outer join dept d on(e.deptno=d.deptno);**

Or

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

	EMPNO	ENAME	JOB	DNAME	LOC
	-----	-----	-----	-----	---
hyd	111	saketh	analyst	mkt	
hyd	333	jagan	manager	mkt	
bang	222	sudha	clerk	fin	
bombay				hr	

FULL OUTER JOIN

This will display the all matching records and the non-matching records from both tables.

Ex:

**SQL> select empno,ename,job,dname,loc from emp e
full outer join dept d on(e.deptno=d.deptno);**

EMPNO	ENAME	JOB	DNAME	LOC
333	jagan	manager	mkt	hyd
111	saketh	analyst	mkt	hyd
222	sudha	clerk	fin	bang
444	madhu	engineer	hr	bombay

INNER JOIN

This will display all the records that have matched.

Ex:

**SQL> select empno,ename,job,dname,loc from emp
inner join dept using(deptno);**

EMPNO	ENAME	JOB	DNAME	LOC
111	saketh	analyst	mkt	hyd
333	jagan	manager	mkt	hyd
222	sudha	clerk	fin	bang

SUBQUERIES AND EXISTS

SUBQUERIES

- Nesting of queries, one within the other is termed as a subquery.
- A statement containing a subquery is called a parent query.
- Subqueries are used to retrieve data from tables that depend on the values in the table itself.

TYPES

- Single row subqueries
- Multi row subqueries
- Multiple subqueries
- Correlated subqueries

SINGLE ROW SUBQUERIES

In single row subquery, it will return one value.

Ex:

SQL> select * from emp where sal > (select sal from emp where empno = 7566);

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

MULTI ROW SUBQUERIES

- In multi row subquery, it will return more than one value.
- In such cases we should include operators like any, all, in or not in between the comparison operator and the subquery.

Ex:

SQL> select * from emp where sal > any (select sal from emp where sal between 2500 and 4000);

SAL	EMPNO COMM	ENAME DEPTNO	JOB	MGR	HIREDATE
	-----	-----	-----	-----	-----
2975	7566	JONES 20	MANAGER	7839	02-APR-81
3000	7788	SCOTT 20	ANALYST	7566	19-APR-87
5000	7839	KING 10	PRESIDENT		17-NOV-81
3000	7902	FORD 20	ANALYST	7566	03-DEC-81

SQL> select * from emp where sal > all (select sal from emp where sal between 2500 and 4000);

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

MULTIPLE SUBQUERIES

- There is no limit on the number of subqueries included in a where clause.
- It allows nesting of a query within a subquery.

Ex:

SQL> 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			
3000		20	
7902	FORD	ANALYST	7566
03-DEC-81			
3000		20	

CORRELATED SUBQUERIES

A subquery is evaluated once for the entire parent statement where as a correlated subquery is evaluated once for every row processed by the parent statement.

Ex:

SQL> select distinct deptno from emp e where 5 <= (select count(ename) from emp where e.deptno = deptno);

DEPTNO
20
30

EXISTS

Exists function is a test for existence. This is a logical test for the return of rows from a query.

Ex:

Suppose we want to display the department numbers which has more than 4 employees.

SQL> select deptno,count(*) from emp group by deptno having count(*) > 4;

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

From the above query can you want to display the names of employees ?

SQL> select deptno,ename, count(*) from emp group by deptno,ename having count(*) > 4;

no rows selected

The above query returns nothing because combination of deptno and ename never return more than one count.

The solution is to use exists which follows.

SQL> select deptno,ename from emp e1 where exists (select * from emp e2 where e1.deptno=e2.deptno group by e2.deptno having count(e2.ename) > 4) order by deptno,ename;

DEPTNO	ENAME
-----	-----
20	ADAMS
20	FORD
20	JONES

```

20      SCOTT
20      SMITH
30      ALLEN
30      BLAKE
30      JAMES
30 MARTIN
30      TURNER
30      WARD

```

NOT EXISTS

SQL> select deptno,ename from emp e1 where not exists (select * from emp e2 where e1.deptno=e2.deptno group by e2.deptno having count(e2.ename) > 4) order by deptno,ename;

```

      DEPTNO      ENAME
-----
      10  CLARK
      10  KING
      10  MILLER

```

WALKUP TREES AND INLINE VIEW

WALKUP TREES

Using hierarchical queries, you can retrieve data based on a natural hierarchical relationship between rows in a table. However, where a hierarchical relationship exists between the rows of a table, a process called tree walking enables the hierarchy to be constructed.

Ex:

SQL> select ename || '==>' || prior ename, level from

**emp start with ename = 'KING' connect by prior
empno=mgr;**

ENAME '==>' PRIORENAM	LEVEL
-----	-----
KING==>	1
JONES==>KING	2
SCOTT==>JONES	3
ADAMS==>SCOTT	4
FORD==>JONES	3
SMITH==>FORD	4
BLAKE==>KING	2
ALLEN==>BLAKE	3
WARD==>BLAKE	3
MARTIN==>BLAKE	3
TURNER==>BLAKE	3
JAMES==>BLAKE	3
CLARK==>KING	2
MILLER==>CLARK	3

- In the above Start with clause specifies the root row of the table.
- Level pseudo column gives the 1 for root , 2 for child and so on.
- Connect by prior clause specifies the columns which has parent-child relationship.

INLINE VIEW OR TOP-N ANALYSIS

In the select statement instead of table name, replacing the select statement is known as inline view.

Ex:

**SQL> Select ename, sal, rownum rank from (select
*from emp order by sal);**

ENAME	SAL	RANK
-----	-----	-----
SMITH	800	1
JAMES	950	2
ADAMS	1100	3
WARD	1250	4
MARTIN	1250	5
MILLER	1300	6
TURNER	1500	7
ALLEN	1600	8
CLARK	2450	9
BLAKE	2850	10
JONES	2975	11
SCOTT	3000	12
FORD	3000	13
KING	5000	14

LOCKS

- Locks are the mechanisms used to prevent destructive interaction between users accessing same resource simultaneously.
- Locks provides high degree of data concurrency.

TYPES

- Row level locks
- Table level locks

ROW LEVEL LOCKS

- In the row level lock a row is locked exclusively so that other cannot modify the row until the transaction holding the lock is committed or rolled back.
- This can be done by using select..for update clause.

Ex:

SQL> select * from emp where sal > 3000 for update of comm.;

TABLE LEVEL LOCKS

A table level lock will protect table data thereby guaranteeing data integrity when data is being accessed concurrently by multiple users.

A table lock can be held in several modes.

- Share lock
- Share update lock
- Exclusive lock

SHARE LOCK

- A share lock locks the table allowing other users to only query but not insert, update or delete rows in a table.
- Multiple users can place share locks on the same resource at the same time.

Ex:

SQL> lock table emp in share mode;

SHARE UPDATE LOCK

- It locks rows that are to be updated in a table.
- It permits other users to concurrently query, insert, update or even lock other rows in the same table.

- It prevents the other users from updating the row that has been locked.

Ex:

SQL> lock table emp in share update mode;

EXCLUSIVE LOCK

- Exclusive lock is the most restrictive of tables locks.
- When issued by any user, it allows the other user to only query.
- It is similar to share lock but only one user can place exclusive lock on a table at a time.

Ex:

SQL> lock table emp in share exclusive mode;

NOWAIT

- If one user locked the table without nowait then another user trying to lock the same table then he has to wait until the user who has initially locked the table issues a commit or rollback statement.
- This delay could be avoided by appending a nowait clause in the lock table command.

Ex:

SQL> lock table emp in exclusive mode nowait.

DEADLOCK

- A deadlock occurs when two users have a lock each on separate object, and they want to acquire a lock on the each other's object.
- When this happens, the first user has to wait for the second user to release the lock, but the second user will not release it until the lock on the first user's object is freed.
- In such a case, oracle detects the deadlock automatically and solves the problem by aborting one of the two transactions.

INDEXES

- Index is typically a listing of keywords accompanied by the location of information on a subject.
- We can create indexes explicitly to speed up SQL statement execution on a table.
- The index points directly to the location of the rows containing the value.

WHY INDEXES?

Indexes are most useful on larger tables, on columns that are likely to appear in where clauses as simple equality.

TYPES

- Unique index
- Non-unique index
- Btree index
- Bitmap index
- Composite index
- Reverse key index
- Function-based index

- Descending index
- Domain index
- Object index
- Cluster index
- Text index
- Index organized table
- Partition index
- Local index
- Local prefixed
- Local non-prefixed
- Global index
- Global prefixed
- Global non-prefixed

UNIQUE INDEX

- Unique indexes guarantee that no two rows of a table have duplicate values in the columns that define the index.
- Unique index is automatically created when primary key or unique constraint is created.

Ex:

SQL> create unique index stud_ind on student(sno);

NON-UNIQUE INDEX

Non-Unique indexes do not impose the above restriction on the column values.

Ex:

SQL> create index stud_ind on student(sno);

BTREE INDEX or ASCENDING INDEX

1. The default type of index used in an oracle database is the btree index.
2. A btree index is designed to provide both rapid access to individual rows and quick access to groups of rows within a range.
3. The btree index does this by performing a succession of value comparisons.
4. Each comparison eliminates many of the rows.

Ex:

SQL> create index stud_ind on student(sno);

BITMAP INDEX

This can be used for low cardinality columns: that is columns in which the number of distinct values is small when compared to the number of the rows in the table.

Ex:

SQL> create bitmap index stud_ind on student(sex);

COMPOSITE INDEX

- A composite index also called a concatenated index is an index created on multiple columns of a table.

- Columns in a composite index can appear in any order and need not be adjacent columns of the table.

Ex:

```
SQL> create bitmap index stud_ind on student(sno, sname);
```

REVERSE KEY INDEX

1. A reverse key index when compared to standard index, reverses each byte of the column being indexed while keeping the column order.
2. When the column is indexed in reverse mode then the column values will be stored in an index in different blocks as the starting value differs.
3. Such an arrangement can help avoid performance degradations in indexes where modifications to the index are concentrated on a small set of blocks.

Ex:

```
SQL> create index stud_ind on student(sno, reverse);
```

We can rebuild a reverse key index into normal index using the noreverse keyword.

Ex:

```
SQL> alter index stud_ind rebuild noreverse;
```

FUNCTION BASED INDEX

This will use result of the function as key instead of using column as the value for the key.

Ex:

SQL> create index stud_ind on student(upper(sname));

DESCENDING INDEX

1. The order used by B-tree indexes has been ascending order.
2. You can categorize data in B-tree index in descending order as well.
3. This feature can be useful in applications where sorting operations are required.

Ex:

SQL> create index stud_ind on student(sno desc);

TEXT INDEX

1. Querying text is different from querying data because words have shades of meaning, relationships to other words, and opposites.
2. You may want to search for words that are near each other, or words that are related to thers.
3. These queries would be extremely difficult if all you had available was the standard relational operators.
4. By extending SQL to include text indexes, oracle text permits you to ask very complex questions about the text.

5. To use oracle text, you need to create a text index on the column in which the text is stored.
6. Text index is a collection of tables and indexes that store information about the text stored in the column.

TYPES

There are several different types of indexes available in oracle 9i.

The first, CONTEXT is supported in oracle 8i as well as oracle 9i.

As of oracle 9i, you can use the CTXCAT text index to further enhance your text index management and query capabilities.

- CONTEXT
- CTXCAT
- CTXRULE

The CTXCAT index type supports the transactional synchronization of data between the base table and its text index.

With CONTEXT indexes, you need to manually tell oracle to update the values in the text index after data changes in base table.

CTXCAT index types do not generate score values during the text queries.

HOW TO CREATE TEXT INDEX ?

- You can create a text index via a special version of the create index command.

- For context index, specify the ctxsys.context index type and for ctxcat index, specify the ctxsys.ctxcat index type.

Ex:

Suppose you have a table called BOOKS with the following columns Title, Author, Info.

```
SQL> create index book_index on books(info)  
indextype is ctxsys.context;
```

```
SQL> create index book_index on books(info)  
indextype is ctxsys.ctxcat;
```

TEXT QUERIES

Once a text index is created on the info column of BOOKS table, text-searching capabilities increase dynamically.

CONTAINS & CATSEARCH

CONTAINS function takes two parameters - the column name and the search string.

Syntax:

```
Contains(indexed_column, search_str);
```

If you create a CTXCAT index, use the CATSEARCH function in place of CONTAINS.

CATSEARCH takes three parameters - the column name, the search string and the index set.

Syntax:

Contains(indexed_column, search_str, index_set);

HOW A TEXT QUERY WORKS ?

- When a function such as CONTAINS or CATSEARCH is used in query, the text portion of the query is processed by oracle text.
- The remainder of the query is processed just like a regular query within the database.
- The result of the text query processing and the regular query processing are merged to return a single set of records to the user.
- **SEARCHING FOR AN EXACT MATCH OF A WORD** The following queries will search for a word called 'prperty' whose score is greater than zero.

SQL> select * from books where contains(info, 'property') > 0;

SQL> select * from books where catsearch(info, 'property', null) > 0;

Suppose if you want to know the score of the 'property' in each book, if score values for individual searches range from 0 to 10 for each occurrence of the string within the text then use the score function.

SQL> select title, score(10) from books where contains(info, 'property', 10) > 0;

SEARCHING FOR AN EXACT MATCH OF MULTIPLE WORDS

The following queries will search for two words.

```
SQL> select * from books where contains(info, 'property  
AND harvests') > 0;
```

```
SQL> select * from books where catsearch(info, 'property  
AND harvests', null) > 0;
```

Instead of using AND you could have used an ampersand(&). Before using this method, set define off so the & character will not be seen as part of a variable name.

```
SQL> set define off
```

```
SQL> select * from books where contains(info, 'property  
& harvests') > 0;
```

```
SQL> select * from books where catsearch(info, 'property  
harvests', null) > 0;
```

The following queries will search for more than two words.

```
SQL> select * from books where contains(info, 'property  
AND harvests AND workers') > 0;
```

```
SQL> select * from books where catsearch(info, 'property  
harvests workers', null) > 0;
```

The following queries will search for either of the two words.

```
SQL> select * from books where contains(info, 'property
```

OR harvests') > 0;

Instead of OR you can use a vertical line (|).

```
SQL> select * from books where contains(info, 'property |
harvests') > 0;
```

```
SQL> select * from books where catsearch(info, 'property |
harvests', null) > 0;
```

In the following queries the ACCUM(accumulate) operator adds together the scores of the individual searches and compares the accumulated score to the threshold value.

```
SQL> select * from books where contains(info, 'property
ACCUM harvests') > 0;
```

```
SQL> select * from books where catsearch(info, 'property
ACCUM harvests', null) > 0;
```

Instead of OR you can use a comma(,).

```
SQL> select * from books where contains(info, 'property ,
harvests') > 0;
```

```
SQL> select * from books where catsearch(info, 'property ,
harvests', null) > 0;
```

In the following queries the MINUS operator subtracts the score of the second term's search from the score of the first term's search.

```
SQL> select * from books where contains(info, 'property
MINUS harvests') > 0;
```

```
SQL> select * from books where catsearch(info, 'property  
NOT harvests', null) > 0;
```

Instead of MINUS you can use - and instead of NOT you can use ~.

```
SQL> select * from books where contains(info, 'property -  
harvests') > 0;
```

```
SQL> select * from books where catsearch(info, 'property  
~ harvests', null) > 0;
```

SEARCHING FOR AN EXACT MATCH OF A PHRASE

The following queries will search for the phrase.

If the search phrase includes a reserved word within oracle text, the you must use curly braces ({ }) to enclose text.

```
SQL> select * from books where contains(info,  
'transactions {and} finances') > 0;
```

```
SQL> select * from books where catsearch(info,  
'transactions {and} finances', null) > 0;
```

You can enclose the entire phrase within curly braces, in which case any reserved words within the phrase will be treated as part of the search criteria.

```
SQL> select * from books where contains(info,  
'{transactions and finances}') > 0;
```

```
SQL> select * from books where catsearch(info,  
'{transactions and finances}', null) > 0;
```

SEARCHING FOR WORDS THAT ARE NEAR EACH OTHER

The following queries will search for the words that are in between the search terms.

SQL> select * from books where contains(info, 'workers NEAR harvests') > 0;

Instead of NEAR you can use ;.

SQL> select * from books where contains(info, 'workers ; harvests') > 0;

In CONTEXT index queries, you can specify the maximum number of words between the search terms.

SQL> select * from books where contains(info, 'NEAR((workers, harvests),10)' > 0;

USING WILDCARDS DURING SEARCHES

You can use wildcards to expand the list of valid search terms used during your query.

Just as in regular text-string wildcard processing, two wildcards are available.

% - percent sign; multiple-character wildcard
_ - underscore; single-character wildcard

SQL> select * from books where contains(info, 'worker%') > 0;

SQL> select * from books where contains(info, 'work____') > 0;

SEARCHING FOR WORDS THAT SHARE THE SAME STEM

Rather than using wildcards, you can use stem-expansion capabilities to expand the list of text strings.

Given the 'stem' of a word, oracle will expand the list of words to search for to include all words having the same stem.

Sample expansions are show here. Play - plays playing played playful

```
SQL> select * from books where contains(info, '$manage') > 0;
```

SEARCHING FOR FUZZY MATCHES

1. A fuzzy match expands the specified search term to include words that are spelled similarly but that do not necessarily have the same word stem. Fuzzy matches are most helpful when the text contains misspellings.
2. The misspellings can be either in the searched text or in the search string specified by the user during the query.
3. The following queries will not return anything because its search does not contain the word 'hardest'.

```
SQL> select * from books where contains(info, 'hardest') > 0;
```

1. It does, however, contains the word 'harvest'.
2. A fuzzy match will return the books containing the word 'harvest' even though 'harvest' has a different word stem than the word used as the search term.

3. To use a fuzzy match, precede the search term with a question mark, with no space between the question mark and the beginning of the search term.

```
SQL> select * from books where contains(info,  
'?hardest') > 0;
```

SEARCHING FOR WORDS THAT SOUND LIKE OTHER WORDS

1. SOUNDEX, expands search terms based on how the word sounds.
2. The SOUNDEX expansion method uses the same text-matching logic available via the SOUNDEX function in SQL.
3. To use the SOUNDEX option, you must precede the search term with an exclamation mark(!).

```
SQL> select * from books where contains(info, '!grate')  
> 0;
```

INDEX SYNCHRONIZATION

- When using CONTEXT indexes, you need to manage the text index contents;
- the text indexes are not updated when the base table is updated.
- When the table was updated, its text index is out of sync with the base table.
- To sync of the index, execute the SYNC_INDEX procedure of the CTX_DDL package.

```
SQL> exec CTX_DDL.SYNC_INDEX('book_index');
```

INDEX SETS

- Historically, problems with queries of text indexes have occurred when other criteria are used alongside text searches as part of the where clause.
- To improve the mixed query capability, oracle features index sets.
- The indexes within the index set may be structured relational columns or on text columns.
- To create an index set, use the CTX_DDL package to create the index set and add indexes to it.
- When you create a text index, you can then specify the index set it belongs to.

SQL> exec

CTX_DDL.CREATE_INDEX_SET('books_index_set');

The add non-text indexes.

SQL> exec CTX_DDL.ADD_INDEX('books_index_set',
'title_index'); Now create a CTXCAT text index.

Specify ctxsys.ctxcat as the index type, and list the index set in the parameters clause.

SQL> create index book_index on books(info) indextype
is ctxsys.ctxcat parameters('index set books_index_set');

INDEX-ORGANIZED TABLE

1. An index-organized table keeps its data sorted according to the primary key column values for the table.

2. Index-organized tables store their data as if the entire table was stored in an index.
3. An index-organized table allows you to store the entire table's data in an index.

Ex:

```
SQL> create table student (sno number(2),sname  
varchar(10),smarks number(3) constraint pk primary  
key(sno) organization index;
```

PARTITION INDEX

- Similar to partitioning tables, oracle allows you to partition indexes too.
- Like table partitions, index partitions could be in different tablespaces.

LOCAL INDEXES

- Local keyword tells oracle to create a separate index for each partition.
- In the local prefixed index the partition key is specified on the left prefix. When the underlying table is partitioned based on, say two columns then the index can be prefixed on the first column specified.
- Local prefixed indexes can be unique or non unique.
- Local indexes may be easier to manage than global indexes.

Ex:

SQL> create index stud_index on student(sno) local;

GLOBAL INDEXES

- A global index may contain values from multiple partitions.
- An index is global prefixed if it is partitioned on the left prefix of the index columns.
- The global clause allows you to create a non-partitioned index.
- Global indexes may perform uniqueness checks faster than local (partitioned) indexes.
- You cannot create global indexes for hash partitions or subpartitions.

Ex:

SQL> create index stud_index on student(sno) global;

- Similar to table partitions, it is possible to move them from one device to another.
- But unlike table partitions, movement of index partitions requires individual reconstruction of the index or each partition (only in the case of global index).

Ex:

SQL> alter index stud_ind rebuild partition p2

- Index partitions cannot be dropped manually.
- They are dropped implicitly when the data they refer to is dropped from the partitioned table.

MONITORING USE OF INDEXES

- Once you turned on the monitoring the use of indexes, then we can check whether the table is hitting the index or not.
- To monitor the use of index use the following syntax.

Syntax:

alter index index_name monitoring usage;

then check for the details in V\$OBJECT_USAGE view.

If you want to stop monitoring use the following.

Syntax:

alter index index_name nomonitoring usage;

DATA MODEL

- ALL_INDEXES
- DBA_INDEXES
- USER_INDEXES
- ALL_IND-COLUMNS
- DBA-IND_COLUMNS
- USER_IND_COLUMNS
- ALL_PART_INDEXES
- DBA_PART_INDEXES
- USER_PART_INDEXES
- V\$OBJECT_USAGE

SQL*PLUS COMMANDS

These commands does not require statement terminator and applicable to the sessions , those will be automatically cleared when session was closed.

BREAK

This will be used to breakup the data depending on the grouping.

Syntax:

Break or bre [on <column_name> on report]

COMPUTE

This will be used to perform group functions on the data.

Syntax:

Compute or comp [group_function of column_name on breaking_column_name or report]

TTITLE

This will give the top title for your report. You can on or off the ttitle.

Syntax:

Ttitle or ttit [left | center | right] title_name skip n other_characters

Ttitle or ttit [on or off]

BTITLE

This will give the bottom title for your report. You can on or off the btitle.

Syntax:

**Btitle or btit [left | center | right] title_name skip n
other_characters
Btitle or btit [on or off]**

Ex:

SQL> bre on deptno skip 1 on report

SQL> comp sum of sal on deptno

SQL> comp sum of sal on report

SQL> ttitle center 'EMPLOYEE DETAILS' skip1 center '-
-----'

SQL> btitle center '** THANKQ **'

SQL> select * from emp order by deptno; Output:

EMPLOYEE DETAILS					

SAL	EMPNO COMM	ENAME DEPTNO	JOB	MGR	HIREDATE

2450	7782	CLARK	MANAGER	7839	09-JUN-81
		10			
5000	7839	KING	PRESIDENT		17-NOV-81
1300	7934	MILLER	CLERK	7782	23-JAN-82
----- *****					

8750	sum				
	7369	SMITH	CLERK	7902	17-DEC-80
800	20				
	7876	ADAMS	CLERK	7788	23-MAY-87
1100					
	7902	FORD	ANALYST	7566	03-DEC-81
3000					
	7788	SCOTT	ANALYST	7566	19-APR-87
3000					
	7566	JONES	MANAGER	7839	02-APR-81
2975					

----- *****

10875	sum				
	7499	ALLEN	SALESMAN	7698	20-FEB-81
1600	300 30				
	7698	BLAKE	MANAGER	7839	01-MAY-81
2850					
	7654	MARTIN	SALESMAN	7698	28-SEP-81
1250	1400				
	7900	JAMES	CLERK	7698	03-DEC-81
950					
	7844	TURNER	SALESMAN	7698	08-SEP-81
1500	0				
	7521	WARD	SALESMAN	7698	22-FEB-81
1250	500				

----- *****

9400	sum	
------	-----	--

sum
29025

** THANKQ **

CLEAR

This will clear the existing buffers or break or computations or columns formatting.

Syntax:

Clear or cle buffer | bre | comp | col;

Ex:

SQL> clear buffer

Buffer cleared

SQL> clear bre

Breaks cleared

SQL> clear comp

Computes cleared

SQL> clear col

Columns cleared

CHANGE

This will be used to replace any strings in SQL statements.

Syntax:

Change or c/old_string/new_string

If the old_string repeats many times then new_string replaces the first string only.

Ex:

SQL> select * from det;

```
select * from det
      *
```

ERROR at line 1:

ORA-00942: table or view does not exist

SQL> c/det/dept

```
1* select * from dept
```

SQL> /

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

COLUMN

This will be used to increase or decrease the width of the table columns.

Syntax:

**Column or col <column_name> format
<num_format|text_format>**

Ex:

```
SQL> col deptno format 999
```

```
SQL> col dname format a10
```

SAVE

This will be used to save your current SQL statement as SQL Script file.

Syntax:

Save or sav <file_name>.[extension] replace or rep

If you want to save the filename with existing filename the you have to use replace option. By default it will take sql as the extension.

Ex:

SQL> save ss

Created file ss.sql

SQL> save ss replace

Wrote file ss.sql

EXECUTE

This will be used to execute stored subprograms or packaged subprograms.

Syntax:

Execute or exec <subprogram_name>

Ex:

SQL> exec sample_proc

SPOOL

This will record the data when you spool on, upto when you say spool off. By default it will give lst as extension.

Syntax:

Spool on | off | out | <file_name>.[Extension]

Ex:

SQL> spool on

SQL> select * from dept;

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

SQL> spool off

SQL> ed on.lst

SQL> select * from dept;

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

SQL> spool off

LIST

This will give the current SQL statement.

Syntax:

List or li [start_line_number] [end_line_number]

Ex:

```
SQL> select
      2 *
      3 from
      4 dept;
```

```
SQL> list
      1 select
      2 *
      3 from
      4 * dept
```

```
SQL> list 1
      1 * select
```

```
SQL> list 3
      3 * from
```

```
SQL> list 1 3
      1 select
      2 *
      3 * from
```

INPUT

This will insert the new line to the current SQL statement.

Syntax:

Input or in <string>

Ex:

```
SQL> select *
```

```
SQL> list
1 * select *
SQL> input from dept
SQL> list
1 select *
2 * from dept
```

APPEND

This will add a new string to the existing string in the SQL statement without any space.

Syntax:

Append or app <string>

Ex:

```
SQL> select *
SQL> list
1 * select *
SQL> append from dept
1 * select * from dept
SQL> list
1 * select * from dept
```

DELETE

This will delete the current SQL statement lines.

Syntax:

**Delete or del <start_line_number>
[<end_line_number>]**

Ex:

```
SQL> select
      2 *
      3 from
      4 dept
      5 where
      6 deptno
      7 >10;
```

```
SQL> list
      1 select
      2 *
      3 from
      4 dept
      5 where
      6 deptno
      7 * >10
```

```
SQL> del 1
```

```
SQL> list
      1 *
      2 from
      3 dept
      4 where
      5 deptno
      6 * >10
```

```
SQL> del 2
```

```
SQL> list
  1 *
  2 dept
  3 where
  4 deptno
  5 * >10
```

```
SQL> del 2 4
```

```
SQL> list
  1 *
  2 * >10
```

```
SQL> del
```

```
SQL> list
  1 *
```

VARIABLE

This will be used to declare a variable.

Syntax:

Variable or var <variable_name> <variable_type>

Ex:

```
SQL> var dept_name varchar(15)
```

```
SQL> select dname into dept_name from dept where
deptno = 10;
```

PRINT

This will be used to print the output of the variables that will be declared at SQL level.

Syntax:

Print <variable_name>

Ex:

SQL> print dept_name

```
DEPT_NAME
-----
ACCOUNTING
```

START

This will be used to execute SQL scripts.

Syntax:

start <filename_name>.sql

Ex:

SQL> start ss.sql

SQL> @ss.sql *-- this will execute sql script files only.*

HOST

This will be used to interact with the OS level from SQL.

Syntax:

Host [operation]

Ex:

```
SQL> host
```

```
SQL> host dir
```

SHOW

Using this, you can see several commands that use the set command and status.

Syntax:

Show all | <set_command>

Ex:

```
SQL> show all
```

```
appinfo is OFF and set to "SQL*Plus"
arraysize 15
autocommit OFF
autoprint OFF
autorecovery OFF
autotrace OFF
blockterminator "." (hex 2e)
btitle OFF and is the first few characters of the
next SELECT statement
cmdsep OFF
colsep " "
compatibility version NATIVE
concat "." (hex 2e)
copycommit 0
COPYTYPECHECK is ON
define "&" (hex 26)
describe DEPTH 1 LINENUM OFF INDENT ON
echo OFF
editfile "afiedt.buf"
embedded OFF
escape OFF
FEEDBACK ON for 6 or more rows
```

```
flagger OFF  
flush ON
```

SQL> show verify
verify OFF

RUN

This will runs the command in the buffer.

Syntax:

Run | /

Ex:

```
SQL> run  
SQL> /
```

STORE

This will save all the set command statuses in a file.

Syntax:

**Store set <filename>.[extension] [create] | [replace] |
[append]**

Ex:

```
SQL> store set my_settings.scmd Created file  
my_settings.scmd  
SQL> store set my_settings.cmd replace Wrote file  
my_settings.cmd
```



```
SQL> store set my_settings.cmd append Appended file to  
my_settings.cmd
```

FOLD_AFTER

This will fold the columns one after the other.

Syntax:

Column <column_name> fold_after [no_of_lines]

Ex:

```
SQL> col deptno fold_after 1  
SQL> col dname fold_after 1  
SQL> col loc fold_after 1  
SQL> set heading off  
SQL> select * from dept;
```

```
    10  
ACCOUNTING  
NEW YORK
```

```
    20  
RESEARCH  
DALLAS
```

```
    30  
SALES  
CHICAGO
```

```
    40  
OPERATIONS  
BOSTON
```

FOLD_BEFORE

This will fold the columns one before the other.

Syntax:

Column <column_name> fold_before [no_of_lines]

DEFINE

This will give the list of all the variables currently defined.

Syntax:

Define [variable_name]

Ex:

SQL> define

```
DEFINE _DATE                = "16-MAY-07" (CHAR)
DEFINE _CONNECT_IDENTIFIER = "oracle" (CHAR)
DEFINE _USER                 = "SCOTT" (CHAR)
DEFINE _PRIVILEGE            = "" (CHAR)
DEFINE _SQLPLUS_RELEASE     = "1001000200" (CHAR)
DEFINE _EDITOR               = "Notepad" (CHAR)
DEFINE _O_VERSION            = "Oracle Database 10g
Enterprise Edition Release 10.1.0.2.0 -
                                Production With the
Partitioning, OLAP and Data Mining
                                options" (CHAR)
DEFINE _O_RELEASE            = "1001000200" (CHAR)
```

SET COMMANDS

These commands does not require statement terminator and applicable to the sessions , those will be automatically cleared when session was closed.

LINESIZE

This will be used to set the linesize.
Default linesize is 80.

Syntax:

Set linesize <value>

Ex:

SQL> set linesize 100

PAGESIZE

This will be used to set the pagesize.
Default pagesize is 14.

Syntax:

Set pagesize <value>

Ex:

SQL> set pagesize 30

DESCRIBE

This will be used to see the object's structure.

Syntax:

Describe or desc <object_name>

Ex:

SQL> desc dept

Name	Null?	Type
-----	-----	-----
DEPTNO	NOT NULL	NUMBER (2)
DNAME		VARCHAR2 (14)
LOC		VARCHAR2 (13)

PAUSE

- When the displayed data contains hundreds or thousands of lines, when you select it then it will automatically scrolls and displays the last page data.
- To prevent this you can use this pause option.
- By using this it will display the data corresponding to the pagesize with a break which will continue by hitting the return key.
- By default this will be off.

Syntax:

Set pause on | off

Ex:

SQL> set pause on

FEEDBACK

- This will give the information regarding howmany rows you selected the object.
- By default the feedback message will be displayed, only when the object contains more than 5 rows.

Syntax:

Set feedback <value>

Ex:

SQL> set feedback 4

SQL> select * from dept;

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

4 rows selected.

HEADING

- If you want to display data without headings, then you can achieve with this.
- By default heading is on.

Syntax:

Set heading on | off

Ex:

SQL> set heading off

SQL> select * from dept;

10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS

30	SALES	CHICAGO
40	OPERATIONS	BOSTON

SERVEROUTPUT

- This will be used to display the output of the PL/SQL programs.
- By default this will be off.

Syntax:

Set serveroutput on | off

Ex:

SQL> set serveroutput on

TIME

- This will be used to display the time.
- By default this will be off.

Syntax:

Set time on | off

Ex:

SQL> set time on

19:56:33 SQL>

TIMING

- This will give the time taken to execute the current SQL statement.
- By default this will be off.

Syntax:

Set timing on | off

Ex:

SQL> set timing on

SQL> select * from dept;

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

Elapsed: 00:00:00.06

SQLPROMPT

This will be used to change the SQL prompt.

Syntax:

Set sqlprompt <prompt>

Ex:

SQL> set sqlprompt 'ORACLE>'

ORACLE>

SQLCASE

- This will be used to change the case of the SQL statements.
- By default the case is mixed.

Syntax:

Set sqlcase upper | mixed | lower

Ex:

SQL> set sqlcase upper

SQLTERMINATOR

- This will be used to change the terminator of the SQL statements.
- By default the terminator is ;.

Syntax:

Set sqlterminator <termination_character>

Ex:

SQL> set sqlterminator :
SQL> select * from dept:

DEFINE

- By default if the & character finds then it will treat as bind variable and ask for the input.

- Suppose you want to treat it as a normal character while inserting data, then you can prevent this by using the define option.
- By default this will be on

Syntax:

Set define on | off

Ex:

SQL>insert into dept values(50,'R&D','HYD');

```
Enter value for d:
old 1: insert into dept values(50,'R&D','HYD')
new 1: INSERT INTO DEPT VALUES(50,'R','HYD')
```

SQL> set define off

SQL>insert into dept values(50,'R&D','HYD'); -- *here it won't ask for value*

NEWPAGE

- This will show how many blank lines will be left before the report.
- By default it will leave one blank line.

Syntax:

Set newpage <value>

Ex:

SQL> set newpage 10

- The zero value for newpage does not produce zero blank lines instead it switches to a special property which produces a top-of-form character (hex 13) just before the date on each page.
- Most modern printers respond to this by moving immediately to the top of the next page, where the printing of the report will begin.

HEADSEP

- This allow you to indicate where you want to break a page title or a column heading that runs longer than one line.
- The default heading separator is vertical bar (|).

Syntax:

Set headsep <separation_char>

Ex:

SQL> select * from dept;

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

SQL> set headsep !

SQL> col dname heading 'DEPARTMENT ! NAME'

SQL> /

DEPARTMENT

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

ECHO

- When using a bind variable, the SQL statement is maintained by echo.
- By default this is off.

Syntax:

Set echo on | off

VERIFY

- When using a bind variable, the old and new statements will be maintained by verify.
- By default this is on.

Syntax:

Set verify on | off

Ex:

SQL> select * from dept where deptno = &dno;

```
Enter value for dno: 10
old   1: select * from dept where deptno = &dno
new   1: select * from dept where deptno = 10
```

DEPTNO	DNAME	LOC
-----	-----	-----
10	ACCOUNTING	NEW YORK

SQL> set verify off

SQL> select * from dept where deptno = &dno;

Enter value for dno: 20

DEPTNO	DNAME	LOC
-----	-----	-----
20	RESEARCH	DALLAS

PNO

- This will give displays the page numbers.
- By default the value would be zero.

Ex:

SQL> col hiredate new_value xtoday noprint format a1
trunc

SQL> ttitle left xtoday right 'page' sql.pno

SQL> select * from emp where deptno = 10;

09-JUN-81

page 1

EMPNO	ENAME	JOB	MGR	SAL	COMM
DEPTNO					
-----	-----	-----	-----	-----	-----
7782	CLARK	MANAGER	7839	2450	
10					
7839	KING	PRESIDENT		5000	
10					

7934	MILLER	CLERK	7782	1300
10				

- In the above noprint tells SQLPLUS not to display this column when it prints the results of the SQL statement.
- Dates that have been reformatted by TO_CHAR get a default width of about 100 characters. By changing the format to a1 trunc, you minimize this effect.
- NEW_VALUE inserts contents of the column retrieved by the SQL statement into a variable called xtoday.

SPECIAL FILES

LOGIN.sql

- If you would like SQLPLUS to define your own environmental settings, put all the required commands in a file named login.sql.
- This is a special filename that SQLPLUS always looks for whenever it starts up.
- If it finds login.sql, it executes any commands in it as if you had entered them by hand.
- You can put any command in login.sql that you can use in SQLPLUS, including SQLPLUS commands and SQL statements.
- All to them executed before SQLPLUS gives you the SQL> prompt.

GLOGIN.sql

This is used in the same ways as LOGIN.sql but to establish default SQLPLUS settings for all users of a database.

IMP QUERIES

1. To find the nth row of a table

SQL> Select *from emp where rowid = (select max(rowid) from emp where rownum <= 4); Or
SQL> Select *from emp where rownum <= 4 minus
select *from emp where rownum <= 3;

2. To find duplicate rows

SQL> Select *from emp where rowid in (select max(rowid) from emp group by empno, ename, mgr, job, hiredate, comm, deptno, sal); Or
SQL> Select empno,ename,sal,job,hiredate,comm , count(*) from emp group by empno,ename,sal,job,hiredate,comm having count(*) >=1;

3. To delete duplicate rows

SQL> Delete emp where rowid in (select max(rowid) from emp group by empno,ename,mgr,job,hiredate,sal,comm,deptno);

4. To find the count of duplicate rows

SQL> Select ename, count(*) from emp group by ename having count(*) >= 1;

5. How to display alternative rows in a table ?

SQL> select *from emp where (rowid,0) in (select rowid,mod(rownum,2) from emp);

6. Getting employee details of each department who is drawing maximum sal ?

SQL> select *from emp where (deptno,sal) in (select deptno,max(sal) from emp group by deptno);

7. **How to get number of employees in each department , in which department is having more than 2500 employees ?**

SQL> Select deptno,count(*) from emp group by deptno having count(*) >2500;

8. **To reset the time to the beginning of the day**

SQL> Select to_char(trunc(sysdate),'dd-mon-yyyy hh:mi:ss am') from dual;

9. **To find nth maximum sal**

SQL> Select *from emp where sal in (select max(sal) from (select *from emp order by sal) where rownum <= 5);

INTRODUCTION

CHARACTERSTICS

- Highly structured, readable and accessible language.
- Standard and Portable language.
- Embedded language.
- Improved execution authority.

10g FEATURES

- **Optimized compiler**

To change the optimizer settings for the entire database, set the database parameter PLSQL_OPTIMIZE_LEVEL. Valid settings are as follows

- 0 - No optimization
- 1 - Moderate optimization
- 2 - Aggressive optimization

These settings are also modifiable for the current session.

```
SQL> alter session set plsql_optimize_level=2;
```

Oracle retains optimizer settings on a module-by-module basis. When you recompile a particular module with nondefault settings, the settings will stick allowing you to recompile later on using REUSE SETTINGS.

```
SQL> Alter procedure proc compile
```

```
plsql_optimize_level=1;
```

```
SQL> Alter procedure proc compile reuse settings;
```

- **Compile-time warnings.**

Starting with oracle database 10g release 1 you can enable additional compile-time warnings to help make your programs more robust. The compiler can detect potential runtime problems with your code, such as identifying lines of code that will never be run. This process, also known as lint checking.

To enable these warnings for the entire database, set the database parameter PLSQL_WARNINGS. These settings are also modifiable for the current session.

```
SQL> alter session set plsql_warnings =
```


'enable:all'; The above can be achieved using the built-in package DBMS_WARNING.

- **Conditional compilation.**

Conditional compilation allows the compiler to allow to compile selected parts of a program based on conditions you provide with the \$IF directive.

- Support for non-sequential collections in FORALL.
- Improved datatype support.
- Backtrace an exception to its line number.

When handling an error, how can you find the line number on which the error was originally raised ?

In earlier release, the only way to do this was allow you exception to go unhandled and then view the full error trace stack.

Now you can

call **DBMS_UTILITY.FORMAT_ERROR_BACKTRACE** function to obtain that stack and manipulate it programmatically within your program.

- Set operators for nested tables.
- **built-in functions.**
 - REGEXP_LIKE
 - REGEXP_INSTR
 - REGEXP_SUBSTR
 - REGEXP_REPLACE
- **Programmer-defined quoting mechanism.**

Starting with oracle database 10g release 1, you can define your own quoting mechanism for string literals in both SQL and PL/SQL.

Use the characters q'(q followed by a single quote) to note the programmer-defined delimiter for you string literal.

Ex:

```
DECLARE
    v varchar(10) := 'computer';
BEGIN
    dbms_output.put_line(q'*v = *' || v);
    dbms_output.put_line(q'$v = $' || v);
END;
```

```
Output:
    v = computer
v = computer
```

- **Many new built-in packages.**

DBMS_SCHEDULER

Represents a major update to DBMS_JOB. DBMS_SCHEDULER provides much improved functionality for scheduling and executing jobs defined via stored procedures.

DBMS_CRYPTO

Offers the ability to encrypt and decrypt common oracle datatype, including RAWs, BLOBs, and CLOBs. It also provides globalization support for encrypting data across different character sets.

DBMS_MONITOR

Provides an API to control additional tracing and statistics gathering of sessions.

DBMS_WARNING

Provides an API into the PL/SQL compiler warnings module, allowing you to read and change settings that control which warnings are suppressed, displayed, or treated as errors.

STANDARD PACKAGE

Oracle has defined in this special package. Oracle defines quite a few identifiers in this package, including built-in exceptions, functions and subtypes. You can reference the built-in form by prefixing it with STANDARD.

The basic unit in any PL/SQL program is block. All PL/SQL programs are composed of blocks which can occur sequentially or nested.

BLOCK STRUCTURE

```
Declare
-- declarative section
Begin
-- executable section
Exception
-- exception section
End;
```

In the above declarative and exception sections are optional.

BLOCK TYPES

- Anonymous blocks
- Named blocks
 - Labeled blocks
 - Subprograms
 - Triggers

ANONYMOUS BLOCKS

Anonymous blocks implies basic block structure.

Ex:

```
BEGIN
    Dbms_output.put_line('My first
program');
END;
```

LABELED BLOCKS

Labeled blocks are anonymous blocks with a label which gives a name to the block.

Ex:

```
<<my_block>>
BEGIN
    Dbms_output.put_line('My first program');
END;
```

SUBPROGRAMS

Subprograms are procedures and functions. They can be stored in the database as stand-alone objects, as part of package or as methods of an object type.

TRIGGERS

Triggers consists of a PL/SQL block that is associated with an event that occur in the database.

NESTED BLOCKS

A block can be nested within the executable or exception section of an outer block.

IDENTIFIERS

Identifiers are used to name PL/SQL objects, such as variables, cursors, types and subprograms. Identifiers consists of a letter, optionally followed by any sequence of characters, including letters, numbers, dollar signs, underscores, and pound signs only. The maximum length for an identifier is 30 characters.

QUOTED IDENTIFIERS

If you want to make an identifier case sensitive, include characters such as spaces or use a reserved word, you can enclose the identifier in double quotation marks.

Ex:

```

                                DECLARE
    "a" number := 5;
    "A" number := 6;
BEGIN
    dbms_output.put_line('a = ' || a);
    dbms_output.put_line('A = ' || A);
END;
Output:
a = 6
A = 6

```

COMMENTS

Comments improve readability and make your program more understandable. They are ignored by the PL/SQL compiler. There are two types of comments available.

- Single line comments
- Multiline comments

SINGLE LINE COMMENTS

A single-line comment can start any point on a line with two dashes and continues until the end of the line.

Ex:

```
BEGIN
    Dbms_output.put_line('hello');    -
- sample program
END;
```

MULTILINE COMMENTS

Multiline comments start with the /* delimiter and ends with */ delimiter.

Ex:

```
BEGIN
    Dbms_output.put_line('hello');
/* sample program */
END;
```

VARIABLE DECLERATIONS

Variables can be declared in declarative section of the block;

Ex:

```
DECLARE
    a number;
    b number := 5;
```

```
c number default 6;
```

CONSTANT DECLERATIONS

To declare a constant, you include the **CONSTANT** keyword, and you must supply a default value.

Ex:

```
DECLARE
    b constant number := 5;
    c constant number default 6;
```

NOT NULL CLAUSE

You can also specify that the variable must be not null.

Ex:

```
DECLARE
    b constant number not null:= 5;
    c number not null default 6;
```

ANCHORED DECLERATIONS

PL/SQL offers two kinds of anchoring.

- Scalar anchoring
- Record anchoring

SCALAR ANCHORING

Use the **%TYPE** attribute to define your variable based on table's column of some other PL/SQL scalar variable.

Ex:

```
DECLARE
    dno dept.deptno%type;
    Subtype t_number is number;
    a t_number;
```

```

Subtype t_sno is
student.sno%type;
V_sno t_sno;

```

RECORD ANCHORING

Use the **%ROWTYPE** attribute to define your record structure based on a table.

Ex:

```

DECLARE
V_dept dept%rowtype;

```

BENEFITS OF ANCHORED DECLARATIONS

- Synchronization with database columns.
- Normalization of local variables.

PROGRAMMER-DEFINED TYPES

With the SUBTYPE statement, PL/SQL allows you to define your own subtypes or aliases of predefined datatypes, sometimes referred to as abstract datatypes.

There are two kinds of subtypes.

- Constrained
- Unconstrained

CONSTRAINED SUBTYPE

A subtype that restricts or constrains the values normally allowed by the datatype itself.

Ex:

**Subtype positive is binary_integer range
1..2147483647;**

In the above declaration a variable that is declared as positive can store only integer greater than zero even though binary_integer ranges from -2147483647..+2147483647.

UNCONSTRAINED SUBTYPE

A subtype that does not restrict the values of the original datatype in variables declared with the subtype.

Ex:

Subtype float is number;

DATATYPE CONVERSIONS

PL/SQL can handle conversions between different families among the datatypes. Conversion can be done in two ways.

- Explicit conversion
- Implicit conversion

EXPLICIT CONVERSION

This can be done using the built-in functions available.

IMPLICIT CONVERSION

PL/SQL will automatically convert between datatype families when possible.

Ex:

```
DECLARE
    a varchar(10);
BEGIN
    select deptno into a from dept where
    dname='ACCOUNTING';
END;
```

In the above variable a is char type and deptno is number type even though, oracle will automatically converts the numeric data into char type assigns to the variable.

PL/SQL can automatically convert between

- Characters and numbers
- Characters and dates

VARIABLE SCOPE AND VISIBILITY

The scope of a variable is the portion of the program in which the variable can be accessed. For PL/SQL variables, this is from the variable declaration until the end of the block. When a variable goes out of scope, the PL/SQL engine will free the memory used to store the variable.

The visibility of a variable is the portion of the program where the variable can be accessed without having to qualify the reference. The visibility is always within the scope. If it is out of scope, it is not visible.

Ex1:

```

        DECLARE
            a number;      -- scope of a
        BEGIN
            -----
        DECLARE
            b number;      -- scope of b
        BEGIN
            -----
        END;
            -----
        END;

```

Ex2:

```

DECLARE
            a number;
            b number;
        BEGIN
            -- a , b available here
        DECLARE
            b char(10);
        BEGIN
            -- a and char type b is available here
        END;
            -----
        END;

```

Ex3:

```

<<my_block>>
DECLARE
    a number;
    b number;
BEGIN
    -- a , b available here
DECLARE
    b char(10);
BEGIN
    -- a and char type b is available here
    -- number type b is available using
<<my_block>>.b

```

```
END;
```

```
-----
```

```
END;
```

PL/SQL CONTROL STRUCTURES

PL/SQL has a variety of control structures that allow you to control the behaviour of the block as it runs.

These structures include conditional statements and loops.

- If-then-else
- Case
- Case with no else
- Labeled case
- Searched case
- Simple loop
- While loop
- For loop
- Goto and Labels

IF-THEN-ELSE

Syntax:

```
If <condition1> then
    Sequence of statements;
Elsif <condition1> then
    Sequence of statements;

.....
Else
    Sequence of statements;
End if;
```

Ex:

```
DECLARE
    dno number(2);
```

```

BEGIN
    select deptno into dno from dept where
dname = 'ACCOUNTING';
    if dno = 10 then
        dbms_output.put_line('Location is NEW
YORK');
    elsif dno = 20 then
        dbms_output.put_line('Location is
DALLAS');
    elsif dno = 30 then
        dbms_output.put_line('Location is
CHICAGO');
    else
        dbms_output.put_line('Location is
BOSTON');
    end if;
END;

```

Output:

Location is NEW YORK

CASE

Syntax:

```

    Case test-variable
        When value1 then sequence of
statements;
        When value2 then sequence of
statements;
        .....
        When valuen then sequence of
statements;
        Else sequence of statements;
    End case;

```

Ex:

```

DECLARE
    dno number(2);
BEGIN

```

```

        select deptno into dno from dept where
dname = 'ACCOUNTING';
        case dno
            when 10 then

dbms_output.put_line('Location is NEW YORK');
            when 20 then

dbms_output.put_line('Location is DALLAS');
            when 30 then

dbms_output.put_line('Location is CHICAGO');
            else

dbms_output.put_line('Location is BOSTON');
        end case;
END;

```

Output:

```

        Location is NEW YORK

```

CASE WITHOUT ELSE

Syntax:

```

        Case test-variable
            When value1 then sequence of
statements;
            When value2 then sequence of
statements;
            .....
            When valuen then sequence of
statements;
        End case;

```

Ex:

```

DECLARE
        dno number(2);
BEGIN
        select deptno into dno from dept where
dname = 'ACCOUNTING';

```

```

        case dno
            when 10 then

dbms_output.put_line('Location is NEW YORK');
            when 20 then

dbms_output.put_line('Location is DALLAS');
            when 30 then

dbms_output.put_line('Location is CHICAGO');
            when 40 then

dbms_output.put_line('Location is BOSTON');
        end case;
END;

Output:
        Location is NEW YORK

```

LABELED CASE

Syntax:

```

        <<label>>
Case test-variable
        When value1 then sequence of
statements;
        When value2 then sequence of
statements;
        .....
        When valuen then sequence of
statements;
        End case;

```

Ex:

```

DECLARE
        dno number(2);
BEGIN
        select deptno into dno from dept where
dname = 'ACCOUNTING';
        <<my_case>>

```

```

        case dno
            when 10 then

dbms_output.put_line('Location is NEW YORK');
            when 20 then

dbms_output.put_line('Location is DALLAS');
            when 30 then

dbms_output.put_line('Location is CHICAGO');
            when 40 then

dbms_output.put_line('Location is BOSTON');
        end case my_case;
END;
```

Output:

Location is NEW YORK

SEARCHED CASE

Syntax:

```

        Case
            When <condition1> then sequence of
statements;
            When <condition2> then sequence of
statements;
            .....
            When <conditionn> then sequence of
statements;
        End case;
```

Ex:

```

DECLARE
        dno number(2);
BEGIN
        select deptno into dno from dept where
dname = 'ACCOUNTING';
        case dno
            when dno = 10 then
```



```

dbms_output.put_line('Location is NEW YORK');
        when dno = 20 then

dbms_output.put_line('Location is DALLAS');
        when dno = 30 then

dbms_output.put_line('Location is CHICAGO');
        when dno = 40 then

dbms_output.put_line('Location is BOSTON');
    end case;
END;

```

Output:

```

        Location is NEW YORK

```

SIMPLE LOOP

Syntax:

```

Loop
Sequence of statements;
Exit when <condition>;
End loop;

```

In the syntax exit when <condition> is equivalent to

```

If <condition> then
    Exit;
End if;

```

Ex:

```

        DECLARE
        i number := 1;
BEGIN
    loop
        dbms_output.put_line('i = ' || i);
        i := i + 1;
        exit when i > 5;
    end loop;
END;

```

Output:

```

        i = 1
i = 2
i = 3
i = 4
i = 5

```

WHILE LOOP

Syntax:

```

While <condition> loop
Sequence of statements;
End loop;

```

Ex:

```

        DECLARE
        i number := 1;
BEGIN
    While i <= 5 loop
        dbms_output.put_line('i = ' || i);
        i := i + 1;
    end loop;
END;

```

Output:

```

        i = 1
i = 2
i = 3
i = 4
i = 5

```

FOR LOOP

Syntax:

```

For <loop_counter_variable> in
low_bound..high_bound loop
Sequence of statements;
End loop;

```

Ex1:

```
BEGIN
    For i in 1..5 loop
        dbms_output.put_line('i = ' || i);
    end loop;
END;
Output:
        i = 1
i = 2
i = 3
i = 4
i = 5
```

Ex2:

```
BEGIN
    For i in reverse 1..5 loop
        dbms_output.put_line('i = ' || i);
    end loop;
END;
Output:
        i = 5
i = 4
i = 3
i = 2
i = 1
```

NULL STATEMENT

Usually when you write a statement in a program, you want it to do something.

There are cases, however, when you want to tell PL/SQL to do absolutely nothing, and that is where the NULL comes.

The NULL statement does nothing except pass control to the next executable statement.

You can use NULL statement in the following situations.

- **Improving program readability:** Sometimes, it is helpful to avoid any ambiguity inherent in an IF statement that doesn't cover all possible cases. For example, when you write an IF statement, you do not have to include an ELSE clause.
- **Nullifying a raised exception:** When you don't want to write any special code to handle an exception, you can use the NULL statement to make sure that a raised exception halts execution of the current PL/SQL block but does not propagate any exceptions to enclosing blocks.
- **Using null after a label:** In some cases, you can pair NULL with GOTO to avoid having to execute additional statements. For example, I use a GOTO statement to quickly move to the end of my program if the state of my data indicates that no further processing is required. Because I do not have to do anything at the termination of the program, I place a NULL statement after the label because at least one executable statement is required there. Even though NULL does nothing, it is still an executable statement.

GOTO AND LABELS

Syntax:

```
Goto label;
```

Where label is a label defined in the PL/SQL block. Labels are enclosed in double angle brackets. When a goto statement is evaluated, control immediately passes to the statement identified by the label.

Ex:

```
BEGIN
```

```

    For i in 1..5 loop
        dbms_output.put_line('i = ' || i);
        if i = 4 then
            goto exit_loop;
        end if;
    end loop;
    <<exit_loop>>
    Null;
END;

```

Output:

```

        i = 1
i = 2
i = 3
i = 4

```

RESTRICTIONS ON GOTO

- It is illegal to branch into an inner block, loop.
- At least one executable statement must follow.
- It is illegal to branch into an if statement.
- It is illegal to branch from one if statement to another if statement.
- It is illegal to branch from exception block to the current block.

PRAGMAS

Pragmas are compiler directives. They serve as instructions to the PL/SQL compiler. The compiler will act on the pragma during the compilation of the block.

Syntax:

```
PRGAMA instruction_to_compiler.
```

PL/SQL offers several pragmas:

- AUTONOMOUS_TRANSACTION

- EXCEPTION_INIT
- RESTRICT_REFERENCES
- SERIALLY_REUSABLE

SUBPROGRAMS

PROCEDURES

A procedure is a module that performs one or more actions.

Syntax:

```

    Procedure [schema.]name [(parameter1
[,parameter2 ...])]
                                [authid definer |
current_user] is
    -- [declarations]
    Begin
    -- executable statements
    [Exception
    -- exception handlers]
    End [name];
  
```

In the above authid clause defines whether the procedure will execute under the authority of the definer of the procedure or under the authority of the current user.

FUNCTIONS

A function is a module that returns a value.

Syntax:

```

    Function [schema.]name [(parameter1
[,parameter2 ...])]
                                Return return_datatype
                                [authid definer |
current_user]
  
```

```

                                [deterministic]
                                [parallel_enable] is
-- [declarations]
Begin
-- executable statements
[Exception
-- exception handlers]
End [name];

```

In the above authid clause defines whether the procedure will execute under the authority of the definer of the procedure or under the authority of the current user.

Deterministic clause defines, an optimization hint that lets the system use a saved copy of the function's return result, if available. The query optimizer can choose whether to use the saved copy or re-call the function.

Parallel_enable clause defines, an optimization hint that enables the function to be executed in parallel when called from within SELECT statement.

PARAMETER MODES

- In (Default)
- Out
- In out

IN

In parameter will act as pl/sql constant.

OUT

- Out parameter will act as uninitialized variable.

- You cannot provide a default value to an out parameter.
- Any assignments made to out parameter are rolled back when an exception is raised in the program.
- An actual parameter corresponding to an out formal parameter must be a variable.

IN OUT

- In out parameter will act as initialized variable.
- An actual parameter corresponding to an in out formal parameter must be a variable.

DEFAULT PARAMETERS

- Default Parameters will not allow in the beginning and middle.
- Out and In Out parameters can not have default values.

Ex:

procedure p(a in number default 5, b in number default 6, c in number default 7) - valid

procedure p(a in number, b in number default 6, c in number default 7) - valid

procedure p(a in number, b in number, c in number default 7) - valid

procedure p(a in number, b in number default 6, c in number) - invalid

procedure p(a in number default 5, b in number default 6, c in number) - invalild

procedure p(a in number default 5, b in number, c in number) - invalild

NOTATIONS

Notations are of two types.

- Positional notation
- Name notation

We can combine positional and name notation but positional notation can not be followed by the name notation.

Ex:

Suppose we have a procedure proc(a number,b number,c number) and we have one anonymous block which contains v1,v2, and v3;

SQL> exec proc (v1,v2,v3) -- Positional notation

SQL> exec proc (a=>v1,b=>v2,c=>v3) -- Named notation

FORMAL AND ACTUAL PARAMETERS

- Parametes which are in calling subprogram are actual parameters.
- Parametes which are in called subprogram are formal parameters.

- If any subprogram was called, once the call was completed then the values of formal parameters are copied to the actual parameters.

Ex1:

```
CREATE OR REPLACE PROCEDURE SAMPLE(a in number,b
out number,c in out number) is
BEGIN
    dbms_output.put_line('After call');
    dbms_output.put_line('a = ' || a || ' b = ' ||
b || ' c = ' || c);
    b := 10;
    c := 20;
    dbms_output.put_line('After assignment');
    dbms_output.put_line('a = ' || a || ' b = ' ||
b || ' c = ' || c);
END SAMPLE;
```

```
DECLARE
    v1 number := 4;
    v2 number := 5;
    v3 number := 6;
BEGIN
    dbms_output.put_line('Before call');
    dbms_output.put_line('v1 = ' || v1 || ' v2 =
' || v2 || ' v3 = ' || v3);
    sample(v1,v2,v3);
    dbms_output.put_line('After completion of
call');
    dbms_output.put_line('v1 = ' || v1 || ' v2 =
' || v2 || ' v3 = ' || v3);
END;
```

Output:

```
Before call
v1 = 4 v2 = 5 v3 = 6
After call
a = 4 b =  c = 6
After assignment
a = 4 b = 10 c = 20
```

After completion of call
v1 = 4 v2 = 10 v3 = 20

Ex2:

```
CREATE OR REPLACE FUN(a in number,b out number,c
in out number) return number IS
BEGIN
    dbms_output.put_line('After call');
    dbms_output.put_line('a = ' || a || ' b = '
|| b || ' c = ' || c);
    dbms_output.put_line('Before assignement
Result = ' || (a*nvl(b,1)*c));
    b := 5;
    c := 7;
    dbms_output.put_line('After assignment');
    dbms_output.put_line('a = ' || a || ' b = '
|| b || ' c = ' || c);
    return (a*b*c);
END FUN;

DECLARE
    v1 number := 1;
    v2 number := 2;
    v3 number := 3;
    v number;
BEGIN
    dbms_output.put_line('Before call');
    dbms_output.put_line('v1 = ' || v1 || ' v2 = '
|| v2 || ' v3 = ' || v3);
    v := fun(v1,v2,v3);
    dbms_output.put_line('After call
completed');
    dbms_output.put_line('v1 = ' || v1 || ' v2 = '
|| v2 || ' v3 = ' || v3);
    dbms_output.put_line('Result = ' || v);
END;
```

Output:

Before call

v1 = 1 v2 = 2 v3 = 3

After call

```

a = 1 b = c = 3
Before assignement Result = 3
After assignment
a = 1 b = 5 c = 7
After call completed
v1 = 1 v2 = 5 v3 = 7
Result = 35

```

RESTRICTIONS ON FORMAL PARAMETERS

- By declaring with specified size in actual parameters.
- By declaring formal parameters with %type specifier.

USING NOCOPY

- Nocopy is a hint, not a command. This means that the compiler might silently decide that it can't fulfill your request for a nocopy parameter.
- The copying from formal to actual can be restricted by issuing nocopy qualifier.
- To pass the out and in out parameters by reference use nocopy qualifier.

Ex:

```

CREATE OR REPLACE PROCEDURE PROC(a in out
nocopy number) IS
BEGIN
----
END PROC;

```

CALL AND EXEC

Call is a SQL statement, which can be used to execute subprograms like exec. Syntax: Call subprogram_name([argument_list]) [into host_variable];

- The parantheses are always required, even if the subprogram takes no arguments.
- We can not use call with out and in out parameters.
- Call is a SQL statement, it is not valid inside a PL/SQL block;
- The INTO clause is used for the output variables of functions only.
- We can not use 'exec' with out or in out parameters.
- Exec is not valid inside a PL/SQL block;

Ex1:

```
CREATE OR REPLACE PROC IS
BEGIN
    dbms_output.put_line('hello world');
END PROC;
Output:
```

```
SQL> call proc();
        hello world
```

Ex2:

```
CREATE OR REPLACE PROC(a in number,b in number) IS
BEGIN
    dbms_output.put_line('a = ' || a || ' b = '
|| b);
END PROC;
```

Output:

```
SQL> call proc(5,6);
        a = 5 b = 6
```

Ex3:

```
CREATE OR REPLACE FUNCTION FUN RETURN VARCHAR IS
BEGIN
    return 'hello world';
END FUN;
```

Output:

```
SQL> variable v varchar(20)

SQL> call fun() into :v;

SQL> print v
      hello world
```

CALL BY REFERENCE AND CALL BY VALUE

- In parameters by default call by reference where as out and in out call by value.
- When parameter passed by reference, a pointer to the actual parameter is passed to the corresponding formal parameter.
- When parameter passed by value it copies the value of the actual parameter to the formal parameter.
- Call by reference is faster than the call by value because it avoids the copying.

SUBPROGRAMS OVERLOADING

- Possible with different number of parameters.
- Possible with different types of data.
- Possible with same type with objects.
- Can not be possible with different types of modes.
- We can overload local subprograms also.

Ex:

```
SQL> create or replace type t1 as object(a number);/
```

```
SQL> create or replace type t1 as object(a number);/
```

```
DECLARE
  i t1 := t1(5);
  j t2 := t2(5);
  PROCEDURE P(m t1) IS
```

```

BEGIN
    dbms_output.put_line('a = ' || m.a);
END P;
PROCEDURE P(n t2) IS
BEGIN
    dbms_output.put_line('b = ' || n.b);
END P;
PROCEDURE PRODUCT(a number,b number) IS
BEGIN
    dbms_output.put_line('Product of a,b = '
|| a * b);
END PRODUCT;
PROCEDURE PRODUCT(a number,b number,c
number) IS
BEGIN
    dbms_output.put_line('Product of a,b = '
|| a * b * c);
END PRODUCT;
BEGIN
    p(i);
    p(j);
    product(4,5);
    product(4,5,6);
END;

```

Output:

```

a = 5
b = 5
Product of a,b = 20
Product of a,b = 120

```

BENEFITS OF OVERLOADING

- Supporting many data combinations
- Fitting the program to the user.

RESTRICTIONS ON OVERLOADING

- Overloaded programs with parameter lists that differ only by name must be called using named notation.
- The parameter list of overloaded programs must differ by more than parameter mode.
- All of the overloaded programs must be defined within the same PL/SQL scope or block.
- Overloaded functions must differ by more than their return type.

IMPORTANT POINTS ABOUT SUBPROGRAMS

- When a stored subprogram is created, it is stored in the data dictionary.
- The subprogram is stored in compile form which is known as p-code in addition to the source text.
- The p-code has all of the references in the subprogram evaluated, and the source code is translated into a form that is easily readable by PL/SQL engine.
- When the subprogram is called, the p-code is read from the disk, if necessary, and executed.
- Once it reads from the disk, the p-code is stored in the shared pool portion of the system global area (SGA), where it can be accessed by multiple users as needed.
- Like all of the contents of the shared pool, p-code is aged out of the shared pool according to a least recently used (LRU) algorithm.
- Subprograms can be local.
- Local subprograms must be declared in the declarative section of PL/SQL block and called from the executable section.
- Subprograms can not have the declarative section separately.

- Stored subprograms can have local subprograms;
- Local subprograms also can have local subprograms.
- If the subprogram contains a variable with the same name as the column name of the table then use the dot method to differentiate (subprogram_name.sal).
- Subprograms can be invalidated.

PROCEDURES Vs FUNCTIONS

- Procedures may return through out and in out parameters where as function must return.
- Procedures can not have return clause where as functions must.
- We can use call statement directly for executing procedure where as we need to declare a variable in case of functions.
- Functions can use in select statements where as procedures can not.
- Functions can call from reports environment where as procedures can not.
- We can use exec for executing procedures where as functions can not.
- Function can be used in dbms_output where as procedure can not.
- Procedure call is a standalone executable statement where as function call is a part of an executable statement.

STORED Vs LOCAL SUBPROGRAMS

- The stored subprogram is stored in compiled p-code in the database, when the procedure is called it does not have to be compiled. The local subprogram is

compiled as part of its containing block. If the containing block is anonymous and is run multiple times, the subprogram has to be compiled each time.

- Stored subprograms can be called from any block submitted by a user who has execute privileges on the subprogram. Local subprograms can be called only from the block containing the subprogram.
- By keeping the stored subprogram code separate from the calling block, the calling block is shorter and easier to understand. The local subprogram and the calling block are one and the same, which can lead to part confusion. If a change to the calling block is made, the subprogram will be recompiled as of the recompilation of the containing block.
- The compiled p-code can be pinned in the shared pool using the DBMS_SHARED_POOL Package. This can improve performance. Local subprograms cannot be pinned in the shared pool by themselves.
- Stand alone stored subprograms can not be overloaded, but packaged subprograms can be overloaded within the same package.
- Local subprograms can be overloaded within the same block.

Ex1:

```
CREATE OR REPLACE PROCEDURE P IS
BEGIN
    dbms_output.put_line('Stored subprogram');
END;
```

Output:

```
SQL> exec p
Stored subprogram
```

Ex2:

```

DECLARE
    PROCEDURE P IS
    BEGIN
        dbms_output.put_line('Local
subprogram');
    END;
BEGIN
    p;
END;

```

```

Output:
Local subprogram

```

COMPILING SUBPROGRAMS

SQL> Alter procedure P1 compile;

SQL> Alter function F1 compile;

SUBPROGRAMS DEPENDENCIES

- A stored subprogram is marked as invalid in the data dictionary if it has compile errors.
- A stored subprogram can also become invalid if a DDL operation is performed on one of its dependent objects.
- If a subprogram is invalidated, the PL/SQL engine will automatically attempt to recompile in the next time it is called.
- If we have two procedures like P1 and P2 in which P1 depends on P2. If we compile P2 then P1 is invalidated.

SUBPROGRAMS DEPENDENCIES IN REMOTE DATABASES

- We will call remote subprogram using connect string like P1@ORACLE;
- If we have two procedures like P1 and P2 in which P1 depends on P2 but P2 was in remote database. If we compile P2 it will not invalidate P1 immediately because the data dictionary does not track remote dependencies.
- Instead the validity of remote objects is checked at runtime. When P1 is called, the remote data dictionary is queried to determine the status of P2.
- P1 and P2 are compared to see if P1 needs to be recompiled, there are two different methods of comparison
 - Timestamp Model
 - Signature Model

TIMESTAMP MODEL

- This is the default model used by oracle.
- With this model, the timestamps of the last modifications of the two objects are compared.
- The last_ddl_time field of user_objects contains the timestamp.
- If the base object has a newer timestamp than the dependent object, the dependent object will be recompiled.

ISSUES WITH THIS MODEL

- If the objects are in different time zones, the comparison is invalid.
- When P1 is in a client side PL/SQL engine such as oracle forms, in this case it may not be possible to

recompile P1, because the source for it may not be included with the forms.

SIGNATURE MODEL

- When a procedure is created, a signature is stored in the data dictionary in addition to the p-code.
- The signature encodes the types and order of the parameters.
- When P1 is compiled the first time, the signature of P2 is included. Thus, P1 only needs to be recompiled when the signature of P2 changes.
- In order to use the signature model, the parameter `REMOTE_DEPENDENCIES_MODE` must be set to `SIGNATURE`. This is a parameter in the database initialization file.

THREE WAYS OF SETTING THIS MODE

- Add the line ***REMOTE_DEPENDENCIES_MODE=SIGNATURE*** to the database initialization file. The next time the database is started, the mode will be set to `SIGNATURE` for all sessions.
- **Alter system set remote_dependencies_mode = signature;** This will affect the entire database (all sessions) from the time the statement is issued. You must have the `ALTER SYSTEM` privilege to issue this command.
- **Alter session set remote_dependencies_mode = signature;** This will only affect your session

ISSUES WITH THIS MODEL

- Signatures don't get modified if the default values of formal parameters are changed.
- Suppose P2 has a default value for one of its parameters, and P1 is using this default value. If the default in the specification for P2 is changed, P1 will not be recompiled by default. The old value for the default parameter will still be used until P1 is manually recompiled.
- If P1 is calling a packaged procedure P2, and a new overloaded version of P2 is added to the remote package, the signature is not changed. P1 will still use the old version (not the new overloaded one) until P1 is recompiled manually.

FORWARD DECLARATION

Before going to use the procedure in any other subprogram or other block , you must declare the prototype of the procedure in declarative section.

Ex1:

```
DECLARE
    PROCEDURE P1 IS
    BEGIN
        dbms_output.put_line('From procedure
p1');
        p2;
    END P1;
    PROCEDURE P2 IS
    BEGIN
        dbms_output.put_line('From procedure
p2');
        p3;
    END P2;
    PROCEDURE P3 IS
```

```

        BEGIN
            dbms_output.put_line('From procedure
p3');
        END P3;
BEGIN
    p1;
END;

```

Output:

p2;

*

ERROR at line 5:

ORA-06550: line 5, column 1:

PLS-00313: 'P2' not declared in this scope

ORA-06550: line 5, column 1:

PL/SQL: Statement ignored

ORA-06550: line 10, column 1:

PLS-00313: 'P3' not declared in this scope

ORA-06550: line 10, column 1:

PL/SQL: Statement ignored

Ex2:

```

DECLARE
    PROCEDURE P2;    -- forward declaration
    PROCEDURE P3;
    PROCEDURE P1 IS
    BEGIN
        dbms_output.put_line('From procedure
p1');
        p2;
    END P1;
    PROCEDURE P2 IS
    BEGIN
        dbms_output.put_line('From procedure
p2');
        p3;
    END P2;
    PROCEDURE P3 IS
    BEGIN
        dbms_output.put_line('From procedure
p3');

```

```

        END P3;
BEGIN
    p1;
END;

```

Output:

```

From procedure p1
From procedure p2
From procedure p3

```

PRIVILEGES AND STORED SUBPROGRAMS

EXECUTE PRIVILEGE

- For stored subprograms and packages the relevant privilege is EXECUTE.
- If user A had the procedure called emp_proc then user A grants execute privilege on procedure to user B with the following command.

SQL> Grant execute on emp_proc to user B.

- Then user B can run the procedure by issuing
SQL> Exec user A.emp_proc

userA created the following procedure

```

CREATE OR REPLACE PROCEDURE P IS
    cursor is select *from student1;
BEGIN
    for v in c loop
        insert into student2
values (v.no,v.name,v.marks);
    end loop;
END P;

```

userA granted execute privilege to userB using

SQL> grant execute on p to userB

Then userB executed the procedure

SQL> Exec userA.p

If suppose userB also having student2 table then which

table will populate whether userA's or userB's.

The answer is userA's student2 table only because by default the procedure will execute under the privilege set of its owner.

The above procedure is known as definer's procedure.

HOW TO POPULATE USER B's TABLE

- Oracle introduces Invoker's and Definer's rights.
- By default it will use the definer's rights.
- An invoker's rights routine can be created by using AUTHID clause to populate the userB's table.
- It is valid for stand-alone subprograms, package specifications, and object type specifications only.

userA created the following procedure

```
CREATE OR REPLACE PROCEDURE P
AUTHID CURRENT_USER IS
    cursor is select *from student1;
BEGIN
    for v in c loop
        insert into student2
values (v.no,v.name,v.marks);
    end loop;
END P;
```

Then grant execute privilege on p to userB.

Executing the procedure by userB, which populates userB's table.

The above procedure is called invoker's procedure.

Instead of current_user of authid clause, if you use definer then it will be called definer' procedure.

STORED SUBPROGRAMS AND ROLES

we have two users saketh and sudha in which saketh has student table and sudha does not. Sudha is going to create a procedure based on student table owned by saketh. Before doing this saketh must grant the permissions on this table to sudha.

SQL> conn saketh/saketh

SQL> grant all on student to sudha;

then sudha can create procedure

SQL> conn sudha/sudha

```
CREATE OR REPLACE PROCEDURE P IS
    cursor c is select *from saketh.student;
BEGIN
    for v in c loop
        dbms_output.put_line('No = ' || v.no);
    end loop;
END P;
```

here procedure will be created.

If the same privilege was granted through a role it wont create the procedure.

Examine the following code

SQL> conn saketh/saketh

SQL> create role saketh_role;

SQL> grant all on student to saketh_role;

SQL> grant saketh_role to sudha;

then conn sudha/sudha

```
CREATE OR REPLACE PROCEDURE P IS
    cursor c is select *from saketh.student;
BEGIN
    for v in c loop
        dbms_output.put_line('No = ' || v.no);
    end loop;
END P;
```

- The above code will raise error instead of creating procedure .
- This is because of early binding which PL/SQL uses by default in which references are evaluated in compile time but when you are using a role this will affect immediately.

ISSUES WITH INVOKER'S RIGHTS

- In an invoker's rights routine, external references in SQL statements will be resolved using the caller's privilege set.
- But references in PL/SQL statements are still resolved under the owner's privilege set.

TRIGGERS, VIEWS AND INVOKER'S RIGHTS

- A database trigger will always be executed with definer's rights and will execute under the privilege set of the schema that owns the triggering table.
- This is also true for PL/SQL function that is called from a view. In this case, the function will execute under the privilege set of the view's owner.

PACKAGES

- A package is a container for related objects.
- It has specification and body.
- Each of them is stored separately in data dictionary.

PACKAGE SYNTAX

```
Create or replace package <package_name> is
```

```

    -- package specification includes subprograms
signatures,
        cursors and global or public
variables.
End <package_name>;
Create or replace package body <package_name> is
    -- package body includes body for all the
subprograms declared
        in the spec, private Variables and
cursors.
    Begin
    -- initialization section
    Exception
    -- Exception handling section
    End <package_name>;

```

IMPORTANT POINTS ABOUT PACKAGES

- The first time a packaged subprogram is called or any reference to a packaged variable or type is made, the package is instantiated.
- Each session will have its own copy of packaged variables, ensuring that two sessions executing subprograms in the same package use different memory locations.
- In many cases initialization needs to be run the first time the package is instantiated within a session. This can be done by adding initialization section to the package body after all the objects.
- Packages are stored in the data dictionary and can not be local.
- Packaged subprograms has an advantage over stand alone subprogram.
- When ever any reference to package, the whole package p-code was stored in shared pool of SGA.

- Package may have local subprograms.
- You can include authid clause inside the package spec not in the body.
- The execution section of a package is know as initialization section.
- You can have an exception section at the bottom of a package body.
- Packages subprograms are not invalidated.

COMPILING PACKAGES

- SQL> Alter package PKG compile;
- SQL> Alter package PKG compile specification;
- SQL> Alter package PKG compile body;

PACKAGE DEPENDENCIES

- The package body depends on the some objects and the package header.
- The package header does not depend on the package body, which is an advantage of packages.
- We can change the package body with out changing the header.

PACKAGE RUNTIME STATE

Package runtime state is differ for the following packages.

- Serially reusable packages
- Non serially reusable packages

SERIALLY REUSABLE PACKAGES

To force the oracle to use serially reusable version then include PRAGMA SERIALLY_REUSABLE in both

package spec and body,
Examine the following package.

```
CREATE OR REPLACE PACKAGE PKG IS
pragma serially_reusable;
procedure emp_proc;
END PKG;
CREATE OR REPLACE PACKAGE BODY PKG IS
pragma serially_reusable;
cursor c is select ename from emp;
PROCEDURE EMP_PROC IS
v_ename emp.ename%type;
v_flag boolean := true;
v_numrows number := 0;
BEGIN
if not c%isopen then
    open c;
end if;
while v_flag loop
    fetch c into v_ename;
    v_numrows := v_numrows + 1;
    if v_numrows = 5 then
        v_flag := false;
    end if;
    dbms_output.put_line('Ename = ' ||
v_ename);
    end loop;
END EMP_PROC;
END PKG;
```

```
SQL> exec pkg.emp_proc
Ename = SMITH
Ename = ALLEN
Ename = WARD
Ename = JONES
Ename = MARTIN
```

```
SQL> exec pkg.emp_proc
Ename = SMITH
Ename = ALLEN
```

```
Ename = WARD
Ename = JONES
Ename = MARTIN
```

- The above package displays the same output for each execution even though the cursor is not closed.
- Because the serially reusable version resets the state of the cursor each time it was called.

NON SERIAL Y REUSABLE PACKAGES

This is the default version used by the oracle, examine the following package.

```
CREATE OR REPLACE PACKAGE PKG IS
procedure emp_proc;
END PKG;
CREATE OR REPLACE PACKAGE BODY IS
cursor c is select ename from emp;
PROCEDURE EMP_PROC IS
v_ename emp.ename%type;
v_flag boolean := true;
v_numrows number := 0;
BEGIN
if not c%isopen then
    open c;
end if;
while v_flag loop
    fetch c into v_ename;
    v_numrows := v_numrows + 1;
    if v_numrows = 5 then
        v_flag := false;
    end if;
    dbms_output.put_line('Ename = ' ||
v_ename);
    end loop;
END EMP_PROC;
END PKG;
```

```
SQL> exec pkg.emp_proc
Ename = SMITH
Ename = ALLEN
Ename = WARD
Ename = JONES
Ename = MARTIN
```

```
SQL> exec pkg.emp_proc

Ename = BLAKE
Ename = CLARK
Ename = SCOTT
Ename = KING
Ename = TURNER
```

- The above package displays the different output for each execution even though the cursor is not closed.
- Because the non-serially reusable version remains the state of the cursor over database calls.

DEPENDENCIES OF PACKAGE RUNTIME STATE

Dependencies can exist between package state and anonymous blocks.

Examine the following program

```
Create this package in first session
CREATE OR REPLACE PACKAGE PKG IS
v number := 5;
procedure p;
END PKG;
CREATE OR REPLACE PACKAGE BODY PKG IS
PROCEDURE P IS
BEGIN
dbms_output.put_line('v = ' || v);
v := 10;
dbms_output.put_line('v = ' || v);
END P;
END PKG;
```


Connect to second session, run the following code.

```
BEGIN  
pkg.p;  
END;
```

The above code will work.

Go back to first session and recreate the package using create.

Then connect to second session and run the following code again.

```
BEGIN  
pkg.p;  
END;
```

This above code will not work because of the following.

- The anonymous block depends on pkg. This is compile time dependency.
- There is also a runtime dependency on the packaged variables, since each session has its own copy of packaged variables.
- Thus when pkg is recompiled the runtime dependency is followed, which invalidates the block and raises the oracle error.
- Runtime dependencies exist only on package state. This includes variables and cursors declared in a package.
- If the package had no global variables, the second execution of the anonymous block would have succeeded.

PURITY LEVELS

In general, calls to subprograms are procedural, they cannot be called from SQL statements.

However, if a stand-alone or packaged function meets certain restrictions, it can be called during execution of a SQL statement.

User-defined functions are called the same way as built-in functions but it must meet different restrictions.

These restrictions are defined in terms of purity levels.

There are four types of purity levels.

WNDS -- Writes No Database State

RNDS -- Reads No Database State

WNPS -- Writes No Package State

RNPS -- Reads No Package State

In addition to the preceding restrictions, a user-defined function must also meet the following requirements to be called from a SQL statement.

- The function has to be stored in the database, either stand-alone or as part of a package.
- The function can take only in parameters.
- The formal parameters must use only database types, not PL/SQL types such as boolean or record.
- The return type of the function must also be a database type.
- The function must not end the current transaction with commit or rollback, or rollback to a savepoint prior to the function execution.
- It also must not issue any alter session or alter system commands.

RESTRICT REFERENCES

For packaged functions, however, the `RESTRICT_REFERENCES` pragma is required to specify the purity level of a given function.

Syntax:

**PRAGMA
RESTRICT_REFERENCES(subprogram_name or
package_name, WNDS [,WNPS] [,RNDS] [,RNPS]);**

Ex:

```
CREATE OR REPLACE PACKAGE PKG IS
function fun1 return varchar;
pragma restrict_references(fun1,wnds);
function fun2 return varchar;
pragma restrict_references(fun2,wnds);
END PKG;

CREATE OR REPLACE PACKAGE BODY PKG IS
FUNCTION FUN1 return varchar IS
BEGIN
    update dept set deptno = 11;
    return 'hello';
END FUN1;
FUNCTION FUN2 return varchar IS
BEGIN
    update dept set dname ='aa';
    return 'hello';
END FUN2;
END PKG;
```

The above package body will not be created, it will give the following errors.

PLS-00452: Subprogram 'FUN1' violates its associated pragma

PLS-00452: Subprogram 'FUN2' violates its associated pragma

```

CREATE OR REPLACE PACKAGE BODY PKG IS
FUNCTION FUN1 return varchar IS
BEGIN
    return 'hello';
END FUN1;
FUNCTION FUN2 return varchar IS
BEGIN
    return 'hello';
END FUN2;
END PKG;

```

Now the package body will be created.

DEFAULT

- If there is no RESTRICT_REFERENCES pragma associated with a given packaged function, it will not have any purity level asserted.
- However, you can change the default purity level for a package.
- The DEFAULT keyword is used instead of the subprogram name in the pragma.

Ex:

```

CREATE OR REPLACE PACKAGE PKG IS
pragma restrict_references(default,wnds);
function fun1 return varchar;
function fun2 return varchar;
END PKG;
CREATE OR REPLACE PACKAGE BODY PKG IS
FUNCTION FUN1 return varchar IS
BEGIN
    update dept set deptno = 11;
    return 'hello';
END FUN1;
FUNCTION FUN2 return varchar IS
BEGIN

```

```

        update dept set dname ='aa';
        return 'hello';
    END FUN2;
END PKG;

```

The above package body will not be created, it will give the following errors because the pragma will apply to all the functions.

PLS-00452: Subprogram 'FUN1' violates its associated pragma

PLS-00452: Subprogram 'FUN2' violates its associated pragma

```

CREATE OR REPLACE PACKAGE BODY PKG IS
FUNCTION FUN1 return varchar IS
BEGIN
        return 'hello';
    END FUN1;
FUNCTION FUN2 return varchar IS
BEGIN
        return 'hello';
    END FUN2;
END PKG;

```

Now the package body will be created.

TRUST

If the TRUST keyword is present, the restrictions listed in the pragma are not enforced.

Rather, they are trusted to be true.

Ex:

```

CREATE OR REPLACE PACKAGE PKG IS
function fun1 return varchar;
pragma restrict_references(fun1,wnds,trust);
function fun2 return varchar;
pragma restrict_references(fun2,wnds,trust);
END PKG;

```

```

CREATE OR REPLACE PACKAGE BODY PKG IS
FUNCTION FUN1 return varchar IS
BEGIN
    update dept set deptno = 11;
    return 'hello';
END FUN1;
FUNCTION FUN2 return varchar IS
BEGIN
    update dept set dname ='aa';
    return 'hello';
END FUN2;
END PKG;

```

The above package will be created successfully.

IMPORTANT POINTS ABOUT RESTRICT_REFERENCES

- This pragma can appear anywhere in the package specification, after the function declaration.
- It can apply to only one function definition.
- For overload functions, the pragma applies to the nearest definition prior to the pragma.
- This pragma is required only for packages functions not for stand-alone functions.
- The Pragma can be declared only inside the package specification.
- The pragma is checked at compile time, not runtime.
- It is possible to specify without any purity levels when trust or combination of default and trust keywords are present.

PINNING IN THE SHARED POOL

The shared pool is the portion of the SGS that contains, among other things, the p-code of compiled subprograms as they are run.

The first time a stored a store subprogram is called, the *p-code* is loaded from disk into the shared pool.

Once the object is no longer referenced, it is free to be aged out.

Objects are aged out of the shared pool using an *LRU(Least Recently Used)* algorithm.

The DBMS_SHARED_POOL package allows you to pin objects in the shared pool.

When an object is pinned, it will never be aged out until you request it, no matter how full the pool gets or how often the object is accessed.

This can improve performance, as it takes time to reload a package from disk.

DBMS_SHARED_POOL has four procedures

- KEEP
- UNKEEP
- SIZES
- ABORTED_REQUEST_THRESHOLD

KEEP

The DBMS_SHARED_POOL.KEEP procedure is used to pin objects in the pool.

Syntax:

PROCEDURE KEEP(object_name varchar2,flag char default 'P');

Here the flag represents different types of flag values for different types of objects.

P -- Package, function or procedure
Q -- Sequence
R -- Trigger
C -- SQL Cursor
T -- Object type
JS -- Java source
JC -- Java class
JR -- Java resource
JD -- Java shared data

UNKEEP

UNKEEP is the only way to remove a kept object from the shared pool, without restarting the database. Kept objects are never aged out automatically.

Syntax:

PROCEDURE UNKEEP(object_name varchar2, flag char default 'P');

SIZES

SIZES will echo the contents of the shared pool to the screen.

Syntax:

PROCEDURE SIZES(minsize number);

Objects with greater than the minsize will be returned. SIZES uses DBMS_OUTPUT to return the data.

ABORTED_REQUEST_THRESHOLD

When the database determines that there is not enough memory in the shared pool to satisfy a given request, it will begin aging objects out until there is enough memory. If enough objects are aged out, this can have a performance impact on other database sessions. The `ABORTED_REQUEST_THRESHOLD` can be used to remedy this.

Syntax:

PROCEDURE

`ABORTED_REQUEST_THRESHOLD(threshold_size number);`

Once this procedure is called, Oracle will not start aging objects from the pool unless at least `threshold_size` bytes is needed.

DATA MODEL FOR SUBPROGRAMS AND PACKAGES

- . `USER_OBJECTS`
- . `USER_SOURCE`
- . `USER_ERRORS`
- . `DBA_OBJECTS`
- . `DBA_SOURCE`
- . `DBA_ERRORS`
- . `ALL_OBJECTS`
- . `ALL_SOURCE`
- . `ALL_ERRORS`

CURSORS

Cursor is a pointer to memory location which is called as context area which contains the information necessary for processing, including the number of rows processed by the statement, a pointer to the parsed representation of the statement, and the active set which is the set of rows returned by the query. Cursor contains two parts

- Header
- Body

Header includes cursor name, any parameters and the type of data being loaded.

Body includes the select statement.

Ex:

**Cursor c(dno in number) return dept%rowtype is
select *from dept;**

In the above

Header - cursor c(dno in number) return dept%rowtype

Body - select *from dept

CURSOR TYPES

- Implicit (SQL)
- Explicit
 - Parameterized cursors
 - REF cursors

CURSOR STAGES

- Open

- Fetch
- Close

CURSOR ATTRIBUTES

- %found
- %notfound
- %rowcount
- %isopen
- %bulk_rowcount
- %bulk_exceptions

CURSOR DECLARATION

Syntax:

Cursor <cursor_name> is select statement;

Ex:

Cursor c is select *from dept;

CURSOR LOOPS

- Simple loop
- While loop
- For loop

SIMPLE LOOP

Syntax:

```
Loop
    Fetch <cursor_name> into
<record_variable>;
    Exit when <cursor_name> % notfound;
```

```

                                <statements>;
End loop;

```

Ex:

```

DECLARE
    cursor c is select * from student;
    v_stud student%rowtype;
BEGIN
    open c;
    loop
        fetch c into v_stud;
        exit when c%notfound;
        dbms_output.put_line('Name = ' ||
v_stud.name);
    end loop;
    close c;
END;

```

Output:

```

Name = saketh
Name = srinu
Name = satish
Name = sudha

```

WHILE LOOP

Syntax:

```

    While <cursor_name> % found loop
        Fetch <cursor_name> nto
<record_variable>;
                                <statements>;
    End loop;

```

Ex:

```

DECLARE

```

```

        cursor c is select * from student;
        v_stud student%rowtype;
BEGIN
    open c;
    fetch c into v_stud;
    while c%found loop
        fetch c into v_stud;
        dbms_output.put_line('Name = ' ||
v_stud.name);
    end loop;
    close c;
END;
Output:
Name = saketh
Name = srinu
Name = satish
Name = sudha

```

FOR LOOP

Syntax:

```

        for <record_variable> in <cursor_name> loop
            <statements>;
        End loop;

```

Ex:

```

DECLARE
        cursor c is select * from student;
BEGIN
    for v_stud in c loop
        dbms_output.put_line('Name = ' ||
v_stud.name);
    end loop;
END;

Output:
Name = saketh

```

```
Name = srinu
Name = satish
Name = sudha
```

PARAMETERIZED CURSORS

- This was used when you are going to use the cursor in more than one place with different values for the same where clause.
- Cursor parameters must be in mode.
- Cursor parameters may have default values.
- The scope of cursor parameter is within the select statement.

Ex:

```
DECLARE
    cursor c(dno in number) is select * from
dept where deptno = dno;
    v_dept dept%rowtype;
BEGIN
    open c(20);
    loop
        fetch c into v_dept;
        exit when c%notfound;
        dbms_output.put_line('Dname = ' ||
v_dept.dname ||
                                ' Loc = ' ||
v_dept.loc);
    end loop;
    close c;
END;
```

Output:

```
Dname = RESEARCH Loc = DALLAS
```

PACKAGED CURSORS WITH HEADER IN SPEC AND BODY IN PACKAGE BODY

- cursors declared in packages will not close automatically.
- In packaged cursors you can modify the select statement without making any changes to the cursor header in the package specification.
- Packaged cursors must be defined in the package body itself, and then use it as global for the package.
- You can not define the packaged cursor in any subprograms.
- Cursor declaration in package without body needs the return clause.

Ex:

```
CREATE OR REPLACE PACKAGE PKG IS
    cursor c return dept%rowtype is select * from
dept;
    procedure proc is
END PKG;

CREATE OR REPLACE PACKAGE BODY PKG IS
    cursor c return dept%rowtype is select *
from dept;
    PROCEDURE PROC IS
BEGIN
    for v in c loop
        dbms_output.put_line('Deptno = ' || v.deptno
|| ' Dname = ' ||
                                v.dname || 'Loc = '
|| v.loc);
    end loop;
END PROC;
```

```
END PKG;
```

Output:

```
SQL> exec pkg.proc
```

```
      Deptno = 10 Dname = ACCOUNTING   Loc = NEW
YORK
```

```
      Deptno = 20 Dname = RESEARCH     Loc =
DALLAS
```

```
      Deptno = 30 Dname = SALES        Loc =
CHICAGO
```

```
      Deptno = 40 Dname = OPERATIONS   Loc =
BOSTON
```

```
CREATE OR REPLACE PACKAGE BODY PKG IS
  cursor c return dept%rowtype is select * from
dept where deptno > 20;
PROCEDURE PROC IS
BEGIN
  for v in c loop
    dbms_output.put_line('Deptno = ' || v.deptno
|| ' Dname = ' ||
                                v.dname || ' Loc = '
|| v.loc);
  end loop;
END PROC;
END PKG;
```

Output:

```
      SQL> exec pkg.proc
      Deptno = 30 Dname = SALES Loc = CHICAGO
      Deptno = 40 Dname = OPERATIONS
Loc = BOSTON
```

REF CURSORS AND CURSOR VARIABLES

- This is unconstrained cursor which will return different types depends upon the user input.
- Ref cursors can not be closed implicitly.

- Ref cursor with return type is called strong cursor.
- Ref cursor with out return type is called weak cursor.
- You can declare ref cursor type in package spec as well as body.
- You can declare ref cursor types in local subprograms or anonymous blocks.
- Cursor variables can be assigned from one to another.
- You can declare a cursor variable in one scope and assign another cursor variable with different scope, then you can use the cursor variable even though the assigned cursor variable goes out of scope.
- Cursor variables can be passed as a parameters to the subprograms.
- Cursor variables modes are in or out or in out.
- Cursor variables can not be declared in package spec and package body (excluding subprograms).
- You can not user remote procedure calls to pass cursor variables from one server to another.
- Cursor variables can not use for update clause.
- You can not assign nulls to cursor variables.
- You can not compare cursor variables for equality, inequality and nullity.

Ex:

```
CREATE OR REPLACE PROCEDURE REF_CURSOR(TABLE_NAME
IN VARCHAR) IS
type t is ref cursor;
c t;
v_dept dept%rowtype;
      type r is record(ename emp.ename%type,job
emp.job%type,
```

```

sal emp.sal%type);
    v_emp r;
    v_stud student.name%type;
BEGIN
    if table_name = 'DEPT' then
        open c for select * from dept;
    elsif table_name = 'EMP' then
        open c for select ename,job,sal from
emp;

        elsif table_name = 'STUDENT' then
            open c for select name from student;
        end if;
    loop
        if table_name = 'DEPT' then
            fetch c into v_dept;
            exit when c%notfound;
            dbms_output.put_line('Deptno = ' ||
v_dept.deptno ||
        ' Dname = ' || v_dept.dname || ' Loc = '
|| v_dept.loc);
            elsif table_name = 'EMP' then
                fetch c into v_emp;
                exit when c%notfound;
                dbms_output.put_line('Ename = ' ||
v_emp.ename ||
                    ' Job = ' || v_emp.job || '
Sal = ' || v_emp.sal);
            elsif table_name = 'STUDENT' then
                fetch c into v_stud;
                exit when c%notfound;
                dbms_output.put_line('Name = ' ||
v_stud);
            end if;
        end loop;
        close c;
    END;

```

Output:

```
SQL> exec ref_cursor('DEPT')
```

```

Deptno = 10 Dname = ACCOUNTING Loc = NEW YORK
Deptno = 20 Dname = RESEARCH Loc = DALLAS
Deptno = 30 Dname = SALES Loc = CHICAGO
Deptno = 40 Dname = OPERATIONS Loc = BOSTON

```

```
SQL> exec ref_cursor('EMP')
```

```

Ename = SMITH Job = CLERK Sal = 800
Ename = ALLEN Job = SALESMAN Sal = 1600
Ename = WARD Job = SALESMAN Sal = 1250
Ename = JONES Job = MANAGER Sal = 2975
Ename = MARTIN Job = SALESMAN Sal = 1250
Ename = BLAKE Job = MANAGER Sal = 2850
Ename = CLARK Job = MANAGER Sal = 2450
Ename = SCOTT Job = ANALYST Sal = 3000
Ename = KING Job = PRESIDENT Sal = 5000
Ename = TURNER Job = SALESMAN Sal = 1500
Ename = ADAMS Job = CLERK Sal = 1100
Ename = JAMES Job = CLERK Sal = 950
Ename = FORD Job = ANALYST Sal = 3000
Ename = MILLER Job = CLERK Sal = 1300

```

```
SQL> exec ref_cursor('STUDENT')
```

```

Name = saketh
Name = srinu
Name = satish
Name = sudha

```

CURSOR EXPRESSIONS

- You can use cursor expressions in explicit cursors.
- You can use cursor expressions in dynamic SQL.
- You can use cursor expressions in REF cursor declarations and variables.
- You can not use cursor expressions in implicit cursors.

- Oracle opens the nested cursor defined by a cursor expression implicitly as soon as it fetches the data containing the cursor expression from the parent or outer cursor.
- Nested cursor closes if you close explicitly.
- Nested cursor closes whenever the outer or parent cursor is executed again or closed or canceled.
- Nested cursor closes whenever an exception is raised while fetching data from a parent cursor.
- Cursor expressions can not be used when declaring a view.
- Cursor expressions can be used as an argument to table function.
- You can not perform bind and execute operations on cursor expressions when using the cursor expressions in dynamic SQL.

USING NESTED CURSORS OR CURSOR EXPRESSIONS

Ex:

```
DECLARE
cursor c is select ename,cursor(select dname from
dept d
                                where e.empno =
d.deptno) from emp e;
type t is ref cursor;
c1 t;
c2 t;
v1 emp.ename%type;
v2 dept.dname%type;
BEGIN
```

```

open c;
loop
    fetch c1 into v1;
    exit when c1%notfound;
    fetch c2 into v2;
    exit when c2%notfound;
    dbms_output.put_line('Ename = ' || v1 ||
' Dname = ' || v2);
end loop;
end loop;
close c;
END;

```

CURSOR CLAUSES

- Return
- For update
- Where current of
- Bulk collect

RETURN

Cursor c return dept%rowtype is select *from dept;

Or

Cursor c1 is select *from dept;

Cursor c return c1%rowtype is select *from dept;

Or

Type t is record(deptno dept.deptno%type, dname dept.dname%type);

Cursor c return t is select deptno, dname from dept;

FOR UPDATE AND WHERE CURRENT OF

Normally, a select operation will not take any locks on the rows being accessed.

This will allow other sessions connected to the database to change the data being selected.

The result set is still consistent. At open time, when the active set is determined, oracle takes a snapshot of the table. Any changes that have been committed prior to this point are reflected in the active set.

Any changes made after this point, even if they are committed, are not reflected unless the cursor is reopened, which will evaluate the active set again.

However, if the FOR UPDATE clause is present, exclusive row locks are taken on the rows in the active set before the open returns.

These locks prevent other sessions from changing the rows in the active set until the transaction is committed or rolled back.

If another session already has locks on the rows in the active set, then SELECT ... FOR UPDATE operation will wait for these locks to be released by the other session.

There is no time-out for this waiting period. The SELECT...FOR UPDATE will hang until the other session releases the lock. To handle this situation, the NOWAIT clause is available.

Syntax:

Select ...from ... for update of column_name [wait n];

If the cursor is declared with the FOR UPDATE clause, the WHERE CURRENT OF clause can be used in an

update or delete statement.

Syntax:

Where current of cursor;

Ex:

```
DECLARE
    cursor c is select * from dept for update
of dname;
BEGIN
    for v in c loop
        update dept set dname = 'aa' where
current of c;
        commit;
    end loop;
END;
```

BULK COLLECT

- This is used for array fetches
- With this you can retrieve multiple rows of data with a single roundtrip.
- This reduces the number of context switches between the pl/sql and sql engines.
- Reduces the overhead of retrieving data.
- You can use bulk collect in both dynamic and static sql.
- You can use bulk collect in select, fetch into and returning into clauses.
- SQL engine automatically initializes and extends the collections you reference in the bulk collect clause.
- Bulk collect operation empties the collection referenced in the into clause before executing the query.

- You can use the limit clause of bulk collect to restrict the no of rows retrieved.
- You can fetch into multiple collections with one column each.
- Using the returning clause we can return data to the another collection.

BULK COLLECT IN FETCH

Ex:

```
DECLARE
    Type t is table of dept%rowtype;
    nt t;
    Cursor c is select *from dept;
BEGIN
    Open c;
    Fetch c bulk collect into nt;
    Close c;
    For i in nt.first..nt.last loop
        dbms_output.put_line('Dname = ' ||
nt(i).dname || ' Loc = ' || nt(i).loc);
    end loop;
END;
```

Output:

```
Dname = ACCOUNTING Loc = NEW YORK
Dname = RESEARCH Loc = DALLAS
Dname = SALES Loc = CHICAGO
Dname = OPERATIONS Loc = BOSTON
```

BULK COLLECT IN SELECT

Ex:

```
DECLARE
    Type t is table of dept%rowtype;
```



```

        Nt t;
BEGIN
    Select * bulk collect into nt from dept;
    for i in nt.first..nt.last loop
        dbms_output.put_line('Dname = ' ||
nt(i).dname || ' Loc = ' || nt(i).loc);
    end loop;
END;

```

Output:

```

Dname = ACCOUNTING Loc = NEW YORK
Dname = RESEARCH Loc = DALLAS
Dname = SALES Loc = CHICAGO
Dname = OPERATIONS Loc = BOSTON

```

LIMIT IN BULK COLLECT

You can use this to limit the number of rows to be fetched.

Ex:

```

DECLARE
    Type t is table of dept%rowtype;
    nt t;
    Cursor c is select *from dept;
BEGIN
    Open c;
    Fetch c bulk collect into nt limit 2;
    Close c;
    For i in nt.first..nt.last loop
        dbms_output.put_line('Dname = ' ||
nt(i).dname || ' Loc = ' || nt(i).loc);
    end loop;
END;

```

Output:

```

Dname = ACCOUNTING Loc = NEW YORK
Dname = RESEARCH Loc = DALLAS

```

MULTIPLE FETCHES IN INTO CLAUSE

Ex1:

```

DECLARE
    Type t is table of dept.dname%type;
    nt t;
    Type t1 is table of dept.loc%type;
    nt1 t;
    Cursor c is select dname,loc from dept;
BEGIN
    Open c;
    Fetch c bulk collect into nt,nt1;
    Close c;
    For i in nt.first..nt.last loop
        dbms_output.put_line('Dname = '
|| nt(i));
    end loop;
    For i in nt1.first..nt1.last loop
        dbms_output.put_line('Loc = ' ||
nt1(i));
    end loop;
END;
```

Output:

```

Dname = ACCOUNTING
Dname = RESEARCH
Dname = SALES
Dname = OPERATIONS
Loc = NEW YORK
Loc = DALLAS
Loc = CHICAGO
Loc = BOSTON
```

Ex2:

```

DECLARE
    type t is table of dept.dname%type;
    type t1 is table of dept.loc%type;
    nt t;
    nt1 t1;
BEGIN
```

```

        Select dname,loc bulk collect into nt,nt1
from dept;
        for i in nt.first..nt.last loop
            dbms_output.put_line('Dname = ' ||
nt(i));
        end loop;
        for i in nt1.first..nt1.last loop
            dbms_output.put_line('Loc = ' ||
nt1(i));
        end loop;
END;

```

Output:

```

Dname = ACCOUNTING
Dname = RESEARCH
Dname = SALES
Dname = OPERATIONS
Loc = NEW YORK
Loc = DALLAS
Loc = CHICAGO
Loc = BOSTON

```

RETURNING CLAUSE IN BULK COLLECT

You can use this to return the processed data to the output variables or typed variables.

Ex:

```

DECLARE
    type t is table of number(2);
    nt t := t(1,2,3,4);
    type t1 is table of varchar(2);
    nt1 t1;
    type t2 is table of student%rowtype;
    nt2 t2;
BEGIN
    select name bulk collect into nt1 from
student;
    forall v in nt1.first..nt1.last

```

```

        update student set no = nt(v) where name
= nt1(v) returning
                                no,name,marks
bulk collect into nt2;
    for v in nt2.first..nt2.last loop
        dbms_output.put_line('Marks = ' ||
nt2(v));
    end loop;
END;

```

Output:

```

Marks = 100
Marks = 200
Marks = 300
Marks = 400

```

POINTS TO REMEMBER

- Cursor name can be up to 30 characters in length.
- Cursors declared in anonymous blocks or subprograms closes automatically when that block terminates execution.
- %bulk_rowcount and %bulk_exceptions can be used only with forall construct.
- Cursor declarations may have expressions with column aliases.
- These expressions are called virtual columns or calculated columns.

SQL IN PL/SQL

The only statements allowed directly in pl/sql are DML and TCL.

BINDING

Binding a variable is the process of identifying the storage location associated with an identifier in the program.

Types of binding

- Early binding
- Late binding
- Binding during the compiled phase is early binding.
- Binding during the runtime phase is late binding.
- In early binding compile phase will take longer because of binding work but the execution is faster.
- In late binding it will shorten the compile phase but lengthens the execution time.
- PL/SQL by default uses early binding.
- Binding also involves checking the database for permissions to access the object Referenced.

DYNAMIC SQL

- If you use DDL in pl/sql it validates the permissions and existence if requires during compile time which makes invalid.
- We can avoid this by using Dynamic SQL.
- Dynamic SQL allows you to create a SQL statement dynamically at runtime.

Two techniques are available for Dynamic SQL.

- Native Dynamic SQL
- DBMS_SQL package

USING NATIVE DYNAMIC SQL

USING EXECUTE IMMEDIATE

Ex:

```
BEGIN
    Execute immediate 'create table student(no
number(2),name varchar(10))';
or
    Execute immediate ('create table student(no
number(2),name varchar(10))');
END;
```

USING EXECUTE IMMEDIATE WITH PL/SQL VARIABLES

Ex:

```
DECLARE
    v varchar(100);
BEGIN
    v := 'create table student(no number(2),name
varchar(10))';
    execute immediate v;
END;
```

USING EXECUTE IMMEDIATE WITH BIND VARIABLES AND USING CLAUSE

Ex:

```
DECLARE
    v varchar(100);
BEGIN
    v := 'insert into student
values(:v1,:v2,:v3)';
    execute immediate v using 6,'f',600;
END;
```

EXECUTING QUERIES WITH OPEN FOR AND USING CLAUSE

Ex:

```
CREATE OR REPLACE PROCEDURE P(smarks in number) IS
    s varchar(100) := 'select *from student
where marks > :m';
    type t is ref cursor;
    c t;
    v student%rowtype;
BEGIN
    open c for s using smarks;
    loop
        fetch c into v;
        exit when c%notfound;
        dbms_output.put_line('Student Marks = '
|| v.marks);
    end loop;
    close c;
END;
```

Output:

```
SQL> exec p(100)
```

```
Student Marks = 200
```

```
Student Marks = 300
```

```
Student Marks = 400
```

QUERIES WITH EXECUTE IMMEDIATE

Ex:

```
DECLARE
    d_name dept.dname%type;
    lc dept.loc%type;
    v varchar(100);
```

```

BEGIN
    v := 'select dname from dept where deptno
= 10';
    execute immediate v into d_name;
    dbms_output.put_line('Dname = ' || d_name);
    v := 'select loc from dept where dname =
:dn';
    execute immediate v into lc using d_name;
    dbms_output.put_line('Loc = ' || lc);
END;

```

Output:

```

Dname = ACCOUNTING
Loc = NEW YORK
VARIABLE NAMES

```

Ex:

```

DECLARE
    Marks number(3) := 100;
BEGIN
    Delete student where marks = marks;      --
this will delete all the rows

in the student table
END;

```

This can be avoided by using the labeled blocks.

```

<<my_block>>
DECLARE
    Marks number(3) := 100;
BEGIN
    Delete student where marks = my_block.marks;
-- delete rows which has a

marks of 100
END;

```


GETTING DATA INTO PL/SQL VARIABLES

Ex:

```
DECLARE
    V1 number;
    V2 varchar(2);
BEGIN
    Select no,name into v1,v2 from student where
marks = 100;
END;
```

DML AND RECORDS

Ex:

```
CREATE OR REPLACE PROCEDURE P(srow in
student%rowtype) IS
BEGIN
insert into student values srow;
END P;
DECLARE
    s student%rowtype;
BEGIN
    s.no := 11;
    s.name := 'aa';
    s.marks := 100;
    p(s);
END;
```

RECORD BASED INSERTS

Ex:

```
DECLARE
    srow student%rowtype;
```

```

BEGIN
    srow.no := 7;
    srow.name := 'cc';
    srow.marks := 500;
    insert into student values srow;
END;

```

RECORD BASED UPDATES

Ex:

```

DECLARE
    srow student%rowtype;
BEGIN
    srow.no := 6;
    srow.name := 'cc';
    srow.marks := 500;
    update student set row=srow where no =
srow.no;
END;

```

USING RECORDS WITH RETURNING CLAUSE

Ex:

```

DECLARE
    srow student%rowtype;
    sreturn student%rowtype;
BEGIN
    srow.no := 8;
    srow.name := 'dd';
    srow.marks := 500;
    insert into student values srow returning
no,name,marks into sreturn;
    dbms_output.put_line('No = ' || sreturn.no);
    dbms_output.put_line('No = ' ||
sreturn.name);

```

```

        dbms_output.put_line('No = ' ||
sreturn.marks);
END;

```

Output:

```

No = 8
No = dd
No = 500

```

FORALL STATEMENT

This can be used to get the data from the database at once by reducing the number of context switches which is a transfer of control between PL/SQL and SQL engine.

Syntax:

```

Forall index_var in
        [ Lower_bound..upper_bound |
        Indices of indexing_collection |
        Values of indexing_collection ]
        SQL statement;

```

FORALL WITH NON-SEQUENTIAL ARRAYS

Ex:

```

DECLARE
        type t is table of student.no%type index by
binary_integer;
        ibt t;
BEGIN
        ibt(1) := 1;
        ibt(10) := 2;
        forall i in ibt.first..ibt.last
                update student set marks = 900 where
no = ibt(i);
END;

```

The above program will give error like 'element at index [2] does not exist'. You can rectify it in one of the two following ways.

USAGE OF INDICES OF TO AVOID THE ABOVE BEHAVIOUR

Ex:

```
DECLARE
    type t is table of student.no%type index by
    binary_integer;
    ibt t;
    type t1 is table of boolean index by
    binary_integer;
    ibt1 t1;
BEGIN
    ibt(1) := 1;
    ibt(10) := 2;
    ibt(100) := 3;
    ibt1(1) := true;
    ibt1(10) := true;
    ibt1(100) := true;
    forall i in indices of ibt1
        update student set marks = 900
where no = ibt(i);
END;
```

USAGE OF INDICES OF TO AVOID THE ABOVE BEHAVIOUR

Ex:

```
DECLARE
    type t is table of student.no%type index by
    binary_integer;
    ibt t;
```

```

        type t1 is table of pls_integer index by
binary_integer;
        ibt1 t1;
BEGIN
        ibt(1) := 1;
        ibt(10) := 2;
        ibt(100) := 3;
        ibt1(11) := 1;
        ibt1(15) := 10;
        ibt1(18) := 100;
        forall i in values of ibt1
                update student set marks = 567
where no = ibt(i);
END;

```

POINTS ABOUT BULK BINDS

- Passing the entire PL/SQL table to the SQL engine in one step is known as bulk bind.
- Bulk binds are done using the forall statement.
- If there is an error processing one of the rows in bulk DML operation, only that row is rolled back.

POINTS ABOUT RETURNING CLAUSE

- This will be used only with DML statements to return data into PL/SQL variables.
- This will be useful in situations like , when performing insert or update or delete if you want to know the data of the table which has been effected by the DML.
- With out going for another SELECT using RETURNING clause we will get the data which will avoid a call to RDBMS kernel.

COLLECTIONS

- Collections are also composite types, in that they allow you to treat several variables as a unit.
- A collection combines variables of the same type.

TYPES

- Varrays
- Nested tables
- Index - by tables (Associate arrays)

VARRAYS

- A varray is datatype very similar to an array.
- A varray has a fixed limit on its size, specified as part of the declaration.
- Elements are inserted into varray starting at index 1, up to maximum length declared in the varray type.
- The maximum size of the varray is 2 giga bytes.

Syntax:

Type <TYPE_NAME> is varray | varying array (<LIMIT>) of <ELEMENT_TYPE>;

Ex1:

```
DECLARE
    type t is varray(10) of varchar(2);
    va t := t('a','b','c','d');
    flag boolean;
BEGIN
    dbms_output.put_line('Limit = ' ||
va.limit);
    dbms_output.put_line('Count = ' ||
va.count);
```

```

        dbms_output.put_line('First Index = ' ||
va.first);
        dbms_output.put_line('Last Index = ' ||
va.last);
        dbms_output.put_line('Next Index = ' ||
va.next(2));
        dbms_output.put_line('Previous Index = ' ||
va.prior(3));
        dbms_output.put_line('VARRAY ELEMENTS');
        for i in va.first..va.last loop
            dbms_output.put_line('va[' || i || ']'
= ' || va(i));
        end loop;
        flag := va.exists(3);
        if flag = true then
            dbms_output.put_line('Index 3 exists
with an element ' || va(3));
        else
            dbms_output.put_line('Index 3 does not
exists');
        end if;
        va.extend;
        dbms_output.put_line('After extend of one
index, Count = ' || va.count);
        flag := va.exists(5);
        if flag = true then
            dbms_output.put_line('Index 5 exists
with an element ' || va(5));
        else
            dbms_output.put_line('Index 5 does not
exists');
        end if;
        flag := va.exists(6);
        if flag = true then
            dbms_output.put_line('Index 6 exists
with an element ' || va(6));
        else
            dbms_output.put_line('Index 6 does not
exists');
        end if;
        va.extend(2);

```

```

        dbms_output.put_line('After extend of two
indexes, Count = ' || va.count);
        dbms_output.put_line('VARRAY ELEMENTS');
        for i in va.first..va.last loop
            dbms_output.put_line('va[' || i || ']
= ' || va(i));
        end loop;
        va(5) := 'e';
        va(6) := 'f';
        va(7) := 'g';
        dbms_output.put_line('AFTER ASSINGNING
VALUES TO EXTENDED ELEMENTS,
VARRAY ELEMENTS');
        for i in va.first..va.last loop
            dbms_output.put_line('va[' || i || ']
= ' || va(i));
        end loop;
        va.extend(3,2);
        dbms_output.put_line('After extend of three
indexes,
Count = ' || va.count);
        dbms_output.put_line('VARRAY ELEMENTS');
        for i in va.first..va.last loop
            dbms_output.put_line('va[' || i || ']
= ' || va(i));
        end loop;
        va.trim;
        dbms_output.put_line('After trim of one
index, Count = ' || va.count);
        va.trim(3);
        dbms_output.put_line('After trim of three
indexs, Count = ' || va.count);
        dbms_output.put_line('AFTER TRIM, VARRAY
ELEMENTS');
        for i in va.first..va.last loop
            dbms_output.put_line('va[' || i || ']
= ' || va(i));
        end loop;
        va.delete;

```



```

        dbms_output.put_line('After delete of entire
varray,
                                Count = ' ||
va.count);
END;
```

Output:

Limit = 10

Count = 4

First Index = 1

Last Index = 4

Next Index = 3

Previous Index = 2

VARRAY ELEMENTS

va[1] = a

va[2] = b

va[3] = c

va[4] = d

Index 3 exists with an element c

After extend of one index, Count = 5

Index 5 exists with an element

Index 6 does not exists

After extend of two indexes, Count = 7

VARRAY ELEMENTS

va[1] = a

va[2] = b

va[3] = c

va[4] = d

va[5] =

va[6] =

va[7] =

AFTER ASSINGNING VALUES TO EXTENDED ELEMENTS,

VARRAY ELEMENTS

va[1] = a

va[2] = b

va[3] = c

va[4] = d

va[5] = e

va[6] = f

va[7] = g

After extend of three indexes, Count = 10

```
VARRAY ELEMENTS
```

```
va[1] = a
va[2] = b
va[3] = c
va[4] = d
va[5] = e
va[6] = f
va[7] = g
va[8] = b
va[9] = b
va[10] = b
```

```
After trim of one index, Count = 9
```

```
After trim of three indexes, Count = 6
```

```
AFTER TRIM, VARRAY ELEMENTS
```

```
va[1] = a
va[2] = b
va[3] = c
va[4] = d
va[5] = e
va[6] = f
```

```
After delete of entire varray, Count = 0
```

Ex2:

```
DECLARE
```

```
    type t is varray(4) of student%rowtype;
    va t := t(null,null,null,null);
```

```
BEGIN
```

```
    for i in 1..va.count loop
        select * into va(i) from student where sno
= i;
        dbms_output.put_line('Sno = ' || va(i).sno
|| ' Sname = ' || va(i).sname);
    end loop;
```

```
END;
```

Output:

```
Sno = 1 Sname = saketh
Sno = 2 Sname = srinu
```

```
Sno = 3 Sname = divya
Sno = 4 Sname = manogni
```

Ex3:

```
DECLARE
    type t is varray(4) of student.smarks%type;
    va t := t(null,null,null,null);
BEGIN
    for i in 1..va.count loop
        select smarks into va(i) from student
where sno = i;
        dbms_output.put_line('Smarks = ' ||
va(i));
    end loop;
END;
```

Output:

```
Smarks = 100
Smarks = 200
Smarks = 300
Smarks = 400
```

Ex4:

```
DECLARE
    type r is record(c1 student.sname%type,c2
student.smarks%type);
    type t is varray(4) of r;
    va t := t(null,null,null,null);
BEGIN
    for i in 1..va.count loop
        select sname,smarks into va(i) from
student where sno = i;
        dbms_output.put_line('Sname = ' ||
va(i).c1 || ' Smarks = '
|| va(i).c2);
    end loop;
```

```
END;
```

Output:

```
Sname = saketh Smarks = 100
Sname = srinu Smarks = 200
Sname = divya Smarks = 300
Sname = manogni Smarks = 400
```

Ex5:

```
DECLARE
    type t is varray(1) of addr;
    va t := t(null);
    cursor c is select * from employ;
    i number := 1;
BEGIN
    for v in c loop
        select address into va(i) from employ
where ename = v.ename;
        dbms_output.put_line('Hno = ' ||
va(i).hno || ' City = '
|| va(i).city);
    end loop;
END;
```

Output:

```
Hno = 11 City = hyd
Hno = 22 City = bang
Hno = 33 City = kochi
```

Ex6:

```
DECLARE
    type t is varray(5) of varchar(2);
    va1 t;
    va2 t := t();
BEGIN
    if va1 is null then
```

```

        dbms_output.put_line('va1 is null');
    else
        dbms_output.put_line('va1 is not null');
    end if;
    if va2 is null then
        dbms_output.put_line('va2 is null');
    else
        dbms_output.put_line('va2 is not null');
    end if;
END;
```

Output:

```

va1 is null
va2 is not null
```

NESTED TABLES

- A nested table is thought of a database table which has no limit on its size.
- Elements are inserted into nested table starting at index 1.
- The maximum size of the varray is 2 giga bytes.

Syntax:

Type <TYPE_NAME> is table of <TABLE_TYPE>;

Ex1:

```

DECLARE
    type t is table of varchar(2);
    nt t := t('a','b','c','d');
    flag boolean;
BEGIN
    if nt.limit is null then
        dbms_output.put_line('No limit to
Nested Tables');
```

```

        else
            dbms_output.put_line('Limit = ' ||
nt.limit);
        end if;
        dbms_output.put_line('Count = ' ||
nt.count);
        dbms_output.put_line('First Index = ' ||
nt.first);
        dbms_output.put_line('Last Index = ' ||
nt.last);
        dbms_output.put_line('Next Index = ' ||
nt.next(2));
        dbms_output.put_line('Previous Index = ' ||
nt.prior(3));
        dbms_output.put_line('NESTED TABLE
ELEMENTS');
        for i in 1..nt.count loop
            dbms_output.put_line('nt[' || i || ']'
= ' || nt(i));
        end loop;
        flag := nt.exists(3);
        if flag = true then
            dbms_output.put_line('Index 3 exists
with an element ' || nt(3));
        else
            dbms_output.put_line('Index 3 does not
exists');
        end if;
        nt.extend;
        dbms_output.put_line('After extend of one
index, Count = ' || nt.count);
        flag := nt.exists(5);
        if flag = true then
            dbms_output.put_line('Index 5 exists
with an element ' || nt(5));
        else
            dbms_output.put_line('Index 5 does not
exists');
        end if;
        flag := nt.exists(6);
        if flag = true then

```

```

        dbms_output.put_line('Index 6 exists
with an element ' || nt(6));
    else
        dbms_output.put_line('Index 6 does not
exists');
    end if;
    nt.extend(2);
    dbms_output.put_line('After extend of two
indexes, Count = ' || nt.count);
    dbms_output.put_line('NESTED TABLE
ELEMENTS');
    for i in 1..nt.count loop
        dbms_output.put_line('nt[' || i || ']'
= ' || nt(i));
    end loop;
    nt(5) := 'e';
    nt(6) := 'f';
    nt(7) := 'g';
    dbms_output.put_line('AFTER ASSINGNING
VALUES TO EXTENDED ELEMENTS,

NESTED TABLE ELEMENTS');
    for i in 1..nt.count loop
        dbms_output.put_line('nt[' || i ||
'] = ' || nt(i));
    end loop;
    nt.extend(5,2);
    dbms_output.put_line('After extend of five
indexes, Count = ' || nt.count);
    dbms_output.put_line('NESTED TABLE
ELEMENTS');
    for i in 1..nt.count loop
        dbms_output.put_line('nt[' || i || ']'
= ' || nt(i));
    end loop;
    nt.trim;
    dbms_output.put_line('After trim of one
index, Count = ' || nt.count);
    nt.trim(3);
    dbms_output.put_line('After trim of three
indexs, Count = ' || nt.count);

```

```

        dbms_output.put_line('AFTER TRIM, NESTED
TABLE ELEMENTS');
        for i in 1..nt.count loop
            dbms_output.put_line('nt[' || i ||
'] = ' || nt(i));
        end loop;
        nt.delete(1);
        dbms_output.put_line('After delete of first
index, Count = ' || nt.count);
        dbms_output.put_line('NESTED TABLE
ELEMENTS');
        for i in 2..nt.count+1 loop
            dbms_output.put_line('nt[' || i || ']'
= ' || nt(i));
        end loop;
        nt.delete(4);
        dbms_output.put_line('After delete of
fourth index, Count = ' || nt.count);
        dbms_output.put_line('NESTED TABLE
ELEMENTS');
        for i in 2..3 loop
            dbms_output.put_line('nt[' || i || ']'
= ' || nt(i));
        end loop;
        for i in 5..nt.count+2 loop
            dbms_output.put_line('nt[' || i || ']'
= ' || nt(i));
        end loop;
        nt.delete;
        dbms_output.put_line('After delete of
entire nested table,

Count =
' || nt.count);
END;

```

Output:

No limit to Nested Tables

Count = 4

First Index = 1

Last Index = 4

Next Index = 3


```

Previous Index = 2
NESTED TABLE ELEMENTS
nt[1] = a
nt[2] = b
nt[3] = c
nt[4] = d
Index 3 exists with an element c
After extend of one index, Count = 5
Index 5 exists with an element
Index 6 does not exists
After extend of two indexes, Count = 7
NESTED TABLE ELEMENTS
nt[1] = a
nt[2] = b
nt[3] = c
nt[4] = d
nt[5] =
nt[6] =
nt[7] =
AFTER ASSINGNING VALUES TO EXTENDED ELEMENTS,
NESTED TABLE ELEMENTS
nt[1] = a
nt[2] = b
nt[3] = c
nt[4] = d
nt[5] = e
nt[6] = f
nt[7] = g
After extend of five indexes, Count = 12
NESTED TABLE ELEMENTS
nt[1] = a
nt[2] = b
nt[3] = c
nt[4] = d
nt[5] = e
nt[6] = f
nt[7] = g
nt[8] = b
nt[9] = b
nt[10] = b
nt[11] = b

```

```

nt[12] = b
After trim of one index, Count = 11
After trim of three indexes, Count = 8
AFTER TRIM, NESTED TABLE ELEMENTS
nt[1] = a
nt[2] = b
nt[3] = c
nt[4] = d
nt[5] = e
nt[6] = f
nt[7] = g
nt[8] = b
After delete of first index, Count = 7
NESTED TABLE ELEMENTS
nt[2] = b
nt[3] = c
nt[4] = d
nt[5] = e
nt[6] = f
nt[7] = g
nt[8] = b
After delete of fourth index, Count = 6
NESTED TABLE ELEMENTS
nt[2] = b
nt[3] = c
nt[5] = e
nt[6] = f
nt[7] = g
nt[8] = b
After delete of entire nested table, Count = 0

```

Ex2:

```

DECLARE
    type t is table of student%rowtype;
    nt t := t(null,null,null,null);
BEGIN
    for i in 1..nt.count loop

```

```

        select * into nt(i) from student where sno
= i;
        dbms_output.put_line('Sno = ' || nt(i).sno
|| ' Sname = ' || nt(i).sname);
    end loop;
END;

```

Output:

```

Sno = 1 Sname = saketh
Sno = 2 Sname = srinu
Sno = 3 Sname = divya
Sno = 4 Sname = manogni

```

Ex3:

```

DECLARE
    type t is table of student.smarks%type;
    nt t := t(null,null,null,null);
BEGIN
    for i in 1..nt.count loop
        select smarks into nt(i) from student
where sno = i;
        dbms_output.put_line('Smarks = ' ||
nt(i));
    end loop;
END;

```

Output:

```

Smarks = 100
Smarks = 200
Smarks = 300
Smarks = 400

```

Ex4:

```

DECLARE
    type r is record(c1 student.sname%type,c2
student.smarks%type);
    type t is table of r;

```

```

        nt t := t(null,null,null,null);
BEGIN
    for i in 1..nt.count loop
        select sname,smarks into nt(i) from
student where sno = i;
        dbms_output.put_line('Sname = ' ||
nt(i).c1 || ' Smarks = ' || nt(i).c2);
    end loop;
END;

```

Output:

```

Sname = saketh Smarks = 100
Sname = srinu Smarks = 200
Sname = divya Smarks = 300
Sname = manogni Smarks = 400

```

Ex5:

```

DECLARE
    type t is table of addr;
    nt t := t(null);
    cursor c is select * from employ;
    i number := 1;
BEGIN
    for v in c loop
        select address into nt(i) from employ
where ename = v.ename;
        dbms_output.put_line('Hno = ' ||
nt(i).hno || ' City = ' || nt(i).city);
    end loop;
END;

```

Output:

```

Hno = 11 City = hyd
Hno = 22 City = bang
Hno = 33 City = kochi

```

Ex6:

```

DECLARE

```

```

type t is varray(5) of varchar(2);
nt1 t;
nt2 t := t();
BEGIN
    if nt1 is null then
        dbms_output.put_line('nt1 is null');
    else
        dbms_output.put_line('nt1 is not null');
    end if;
    if nt2 is null then
        dbms_output.put_line('nt2 is null');
    else
        dbms_output.put_line('nt2 is not null');
    end if;
END;

Output:
nt1 is null
nt2 is not null

```

INDEX-BY TABLES

- An index-by table has no limit on its size.
- Elements are inserted into index-by table whose index may start non-sequentially including negative integers.

Syntax:

**Type <TYPE_NAME> is table of <TABLE_TYPE>
index by binary_integer;**

Ex:

```

DECLARE
    type t is table of varchar(2) index by
binary_integer;
    ibt t;

```

```

        flag boolean;
BEGIN
    ibt(1) := 'a';
    ibt(-20) := 'b';
    ibt(30) := 'c';
    ibt(100) := 'd';
    if ibt.limit is null then
        dbms_output.put_line('No limit to Index
by Tables');
    else
        dbms_output.put_line('Limit = ' ||
ibt.limit);
    end if;
    dbms_output.put_line('Count = ' ||
ibt.count);
    dbms_output.put_line('First Index = ' ||
ibt.first);
    dbms_output.put_line('Last Index = ' ||
ibt.last);
    dbms_output.put_line('Next Index = ' ||
ibt.next(2));
    dbms_output.put_line('Previous Index = ' ||
ibt.prior(3));
    dbms_output.put_line('INDEX BY TABLE
ELEMENTS');
    dbms_output.put_line('ibt[-20] = ' || ibt(-
20));
    dbms_output.put_line('ibt[1] = ' ||
ibt(1));
    dbms_output.put_line('ibt[30] = ' ||
ibt(30));
    dbms_output.put_line('ibt[100] = ' ||
ibt(100));
    flag := ibt.exists(30);
    if flag = true then
        dbms_output.put_line('Index 30 exists
with an element ' || ibt(30));
    else
        dbms_output.put_line('Index 30 does not
exists');
    end if;

```

```

        flag := ibt.exists(50);
        if flag = true then
            dbms_output.put_line('Index 50 exists
with an element ' || ibt(30));
        else
            dbms_output.put_line('Index 50 does not
exists');
        end if;
        ibt.delete(1);
        dbms_output.put_line('After delete of first
index, Count = ' || ibt.count);
        ibt.delete(30);
        dbms_output.put_line('After delete of index
thirty, Count = ' || ibt.count);
        dbms_output.put_line('INDEX BY TABLE
ELEMENTS');
        dbms_output.put_line('ibt[-20] = ' || ibt(-
20));
        dbms_output.put_line('ibt[100] = ' ||
ibt(100));
        ibt.delete;
        dbms_output.put_line('After delete of
entire
                                index-by table,
Count = ' || ibt.count);
END;

```

Output:

No limit to Index by Tables

Count = 4

First Index = -20

Last Index = 100

Next Index = 30

Previous Index = 1

INDEX BY TABLE ELEMENTS

ibt[-20] = b

ibt[1] = a

ibt[30] = c

ibt[100] = d

Index 30 exists with an element c

Index 50 does not exists

```

After delete of first index, Count = 3
After delete of index thirty, Count = 2
INDEX BY TABLE ELEMENTS
ibt[-20] = b
ibt[100] = d
After delete of entire index-by table, Count = 0

```

DIFFERENCES AMONG COLLECTIONS

- Varrays has limit, nested tables and index-by tables has no limit.
- Varrays and nested tables must be initialized before assignment of elements, in index-by tables we can directly assign elements.
- Varrays and nested tables stored in database, but index-by tables can not.
- Nested tables and index-by tables are PL/SQL tables, but varrays can not.
- Keys must be positive in case of nested tables and varrays, in case of index-by tables keys can be positive or negative.
- Referencing nonexistent elements raises `SUBSCRIPT_BEYOND_COUNT` in both nested tables and varrays, but in case of index-by tables `NO_DATA_FOUND` raises.
- Keys are sequential in both nested tables and varrays, non-sequential in index-by tables.
- Individual indexes can be deleted in both nested tables and index-by tables, but in varrays can not.
- Individual indexes can be trimmed in both nested tables and varrays, but in index-by tables can not.
- Individual indexes can be extended in both nested tables and varrays, but in index-by tables can not.

MULTILEVEL COLLECTIONS

Collections of more than one dimension which is a collection of collections, known as multilevel collections.

Syntax:

Type <TYPE_NAME1> is table of <TABLE_TYPE> index by binary_integer; Type <TYPE_NAME2> is varray(<LIMIT>) | table | of <TYPE_NAME1> | index by binary_integer;

Ex1:

```
DECLARE
    type t1 is table of varchar(2) index by
binary_integer;
    type t2 is varray(5) of t1;
    va t2 := t2();
    c number := 97;
    flag boolean;
BEGIN
    va.extend(4);
    dbms_output.put_line('Count = ' ||
va.count);
    dbms_output.put_line('Limit = ' ||
va.limit);
    for i in 1..va.count loop
        for j in 1..va.count loop
            va(i)(j) := chr(c);
            c := c + 1;
        end loop;
    end loop;
    dbms_output.put_line('VARRAY ELEMENTS');
    for i in 1..va.count loop
        for j in 1..va.count loop
            dbms_output.put_line('va[' || i || ']['
|| j || '] = ' || va(i)(j));
```

```

        end loop;
    end loop;
    dbms_output.put_line('First index = ' ||
va.first);
    dbms_output.put_line('Last index = ' ||
va.last);
    dbms_output.put_line('Next index = ' ||
va.next(2));
    dbms_output.put_line('Previous index = '
|| va.prior(3));
    flag := va.exists(2);
    if flag = true then
        dbms_output.put_line('Index 2
exists');
    else
        dbms_output.put_line('Index 2
exists');
    end if;
    va.extend;
    va(1)(5) := 'q';
    va(2)(5) := 'r';
    va(3)(5) := 's';
    va(4)(5) := 't';
    va(5)(1) := 'u';
    va(5)(2) := 'v';
    va(5)(3) := 'w';
    va(5)(4) := 'x';
    va(5)(5) := 'y';
    dbms_output.put_line('After extend of one
index, Count = ' || va.count);
    dbms_output.put_line('VARRAY ELEMENTS');
    for i in 1..va.count loop
        for j in 1..va.count loop
            dbms_output.put_line('va[' || i ||
'][][' || j || '] = ' || va(i)(j));
        end loop;
    end loop;
    va.trim;
    dbms_output.put_line('After trim of one
index, Count = ' || va.count);
    va.trim(2);

```

```

        dbms_output.put_line('After trim of two
indexes, Count = ' || va.count);
        dbms_output.put_line('VARRAY ELEMENTS');
        for i in 1..va.count loop
            for j in 1..va.count loop
                dbms_output.put_line('va[' || i ||
'][' || j || '] = ' || va(i)(j));
            end loop;
        end loop;
        va.delete;
        dbms_output.put_line('After delete of
entire varray, Count = ' || va.count);
END;

```

Output:

Count = 4

Limit = 5

VARRAY ELEMENTS

va[1][1] = a

va[1][2] = b

va[1][3] = c

va[1][4] = d

va[2][1] = e

va[2][2] = f

va[2][3] = g

va[2][4] = h

va[3][1] = i

va[3][2] = j

va[3][3] = k

va[3][4] = l

va[4][1] = m

va[4][2] = n

va[4][3] = o

va[4][4] = p

First index = 1

Last index = 4

Next index = 3

Previous index = 2

Index 2 exists

After extend of one index, Count = 5

VARRAY ELEMENTS

```

va[1][1] = a
va[1][2] = b
va[1][3] = c
va[1][4] = d
va[1][5] = q
va[2][1] = e
va[2][2] = f
va[2][3] = g
va[2][4] = h
va[2][5] = r
va[3][1] = i
va[3][2] = j
va[3][3] = k
va[3][4] = l
va[3][5] = s
va[4][1] = m
va[4][2] = n
va[4][3] = o
va[4][4] = p
va[4][5] = t
va[5][1] = u
va[5][2] = v
va[5][3] = w
va[5][4] = x
va[5][5] = y
After trim of one index, Count = 4
After trim of two indexes, Count = 2
VARRAY ELEMENTS
va[1][1] = a
va[1][2] = b
va[2][1] = e
va[2][2] = f
After delete of entire varray, Count = 0

```

Ex2:

```

DECLARE
    type t1 is table of varchar(2) index by
binary_integer;

```

```

        type t2 is table of t1;
        nt t2 := t2();
        c number := 65;
        v number := 1;
        flag boolean;
BEGIN
        nt.extend(4);
        dbms_output.put_line('Count = ' ||
nt.count);
        if nt.limit is null then
            dbms_output.put_line('No limit to
Nested Tables');
        else
            dbms_output.put_line('Limit = ' ||
nt.limit);
        end if;
        for i in 1..nt.count loop
            for j in 1..nt.count loop
                nt(i)(j) := chr(c);
                c := c + 1;
                if c = 91 then
                    c := 97;
                end if;
            end loop;
        end loop;
        dbms_output.put_line('NESTED TABLE
ELEMENTS');
        for i in 1..nt.count loop
            for j in 1..nt.count loop
                dbms_output.put_line(
                    'nt[' || i || '][' || j
|| '] = ' || nt(i)(j));
            end loop;
        end loop;
        dbms_output.put_line('First index = ' ||
nt.first);
        dbms_output.put_line('Last index = ' ||
nt.last);
        dbms_output.put_line('Next index = ' ||
nt.next(2));

```

```

        dbms_output.put_line('Previous index = '
|| nt.prior(3));
        flag := nt.exists(2);
        if flag = true then
            dbms_output.put_line('Index 2
exists');
        else
            dbms_output.put_line('Index 2
exists');
        end if;
        nt.extend(2);
        nt(1)(5) := 'Q';
        nt(1)(6) := 'R';
        nt(2)(5) := 'S';
        nt(2)(6) := 'T';
        nt(3)(5) := 'U';
        nt(3)(6) := 'V';
        nt(4)(5) := 'W';
        nt(4)(6) := 'X';
        nt(5)(1) := 'Y';
        nt(5)(2) := 'Z';
        nt(5)(3) := 'a';
        nt(5)(4) := 'b';
        nt(5)(5) := 'c';
        nt(5)(6) := 'd';
        nt(6)(1) := 'e';
        nt(6)(2) := 'f';
        nt(6)(3) := 'g';
        nt(6)(4) := 'h';
        nt(6)(5) := 'i';
        nt(6)(6) := 'j';
        dbms_output.put_line('After extend of one
index, Count = ' || nt.count);
        dbms_output.put_line('NESTED TABLE
ELEMENTS');
        for i in 1..nt.count loop
            for j in 1..nt.count loop
                dbms_output.put_line(
                    'nt[' || i || '][' || j ||
'] = ' || nt(i)(j));
            end loop;
        end loop;

```

```

        end loop;
        nt.trim;
        dbms_output.put_line('After trim of one
indexe, Count = ' || nt.count);
        nt.trim(2);
        dbms_output.put_line('After trim of two
indexes, Count = ' || nt.count);
        dbms_output.put_line('NESTED TABLE
ELEMENTS');
        for i in 1..nt.count loop
            for j in 1..nt.count loop
                dbms_output.put_line(
                    'nt[' || i || '][' || j
|| '] = ' || nt(i)(j));
            end loop;
        end loop;
        nt.delete(2);
        dbms_output.put_line(
            'After delete of second
index, Count = ' || nt.count);
        dbms_output.put_line('NESTED TABLE
ELEMENTS');
        loop
            exit when v = 4;
            for j in 1..nt.count+1 loop
                dbms_output.put_line(
                    'nt[' || v || '][' || j
|| '] = ' || nt(v)(j));
            end loop;
            v := v + 1;
            if v= 2 then
                v := 3;
            end if;
        end loop;
        nt.delete;
        dbms_output.put_line('After delete of
entire nested table,
Count = ' || nt.count);
END;

```

Output:

```

Count = 4
No limit to Nested Tables
NESTED TABLE ELEMENTS
nt[1][1] = A
nt[1][2] = B
nt[1][3] = C
nt[1][4] = D
nt[2][1] = E
nt[2][2] = F
nt[2][3] = G
nt[2][4] = H
nt[3][1] = I
nt[3][2] = J
nt[3][3] = K
nt[3][4] = L
nt[4][1] = M
nt[4][2] = N
nt[4][3] = O
nt[4][4] = P
First index = 1
Last index = 4
Next index = 3
Previous index = 2
Index 2 exists
After extend of one index, Count = 6
NESTED TABLE ELEMENTS
nt[1][1] = A
nt[1][2] = B
nt[1][3] = C
nt[1][4] = D
nt[1][5] = Q
nt[1][6] = R
nt[2][1] = E
nt[2][2] = F
nt[2][3] = G
nt[2][4] = H
nt[2][5] = S
nt[2][6] = T
nt[3][1] = I
nt[3][2] = J
nt[3][3] = K

```



```

nt[3][4] = L
nt[3][5] = U
nt[3][6] = V
nt[4][1] = M
nt[4][2] = N
nt[4][3] = O
nt[4][4] = P
nt[4][5] = W
nt[4][6] = X
nt[5][1] = Y
nt[5][2] = Z
nt[5][3] = a
nt[5][4] = b
nt[5][5] = c
nt[5][6] = d
nt[6][1] = e
nt[6][2] = f
nt[6][3] = g
nt[6][4] = h
nt[6][5] = i
nt[6][6] = j
After trim of one indexe, Count = 5
After trim of two indexes, Count = 3
NESTED TABLE ELEMENTS
nt[1][1] = A
nt[1][2] = B
nt[1][3] = C
nt[2][1] = E
nt[2][2] = F
nt[2][3] = G
nt[3][1] = I
nt[3][2] = J
nt[3][3] = K
After delete of second index, Count = 2
NESTED TABLE ELEMENTS
nt[1][1] = A
nt[1][2] = B
nt[1][3] = C
nt[3][1] = I
nt[3][2] = J
nt[3][3] = K

```

After delete of entire nested table, Count = 0

Ex3:

```

DECLARE
    type t1 is table of varchar(2) index by
binary_integer;
    type t2 is table of t1 index by
binary_integer;
    ibt t2;
    flag boolean;
BEGIN
    dbms_output.put_line('Count = ' ||
ibt.count);
    if ibt.limit is null then
        dbms_output.put_line('No limit to
Index-by Tables');
    else
        dbms_output.put_line('Limit = ' ||
ibt.limit);
    end if;
    ibt(1)(1) := 'a';
    ibt(4)(5) := 'b';
    ibt(5)(1) := 'c';
    ibt(6)(2) := 'd';
    ibt(8)(3) := 'e';
    ibt(3)(4) := 'f';
    dbms_output.put_line('INDEX-BY TABLE
ELEMENTS');
    dbms_output.put_line('ibt([1][1] = ' ||
ibt(1)(1));
    dbms_output.put_line('ibt([4][5] = ' ||
ibt(4)(5));
    dbms_output.put_line('ibt([5][1] = ' ||
ibt(5)(1));
    dbms_output.put_line('ibt([6][2] = ' ||
ibt(6)(2));
    dbms_output.put_line('ibt([8][3] = ' ||
ibt(8)(3));

```

```

        dbms_output.put_line('ibt([3][4] = ' ||
ibt(3)(4));
        dbms_output.put_line('First Index = ' ||
ibt.first);
        dbms_output.put_line('Last Index = ' ||
ibt.last);
        dbms_output.put_line('Next Index = ' ||
ibt.next(3));
        dbms_output.put_line('Prior Index = ' ||
ibt.prior(8));
        ibt(1)(2) := 'g';
        ibt(1)(3) := 'h';
        ibt(1)(4) := 'i';
        ibt(1)(5) := 'k';
        ibt(1)(6) := 'l';
        ibt(1)(7) := 'm';
        ibt(1)(8) := 'n';
        dbms_output.put_line('Count = ' ||
ibt.count);
        dbms_output.put_line('INDEX-BY TABLE
ELEMENTS');
        for i in 1..8 loop
            dbms_output.put_line('ibt[1][' || i
|| '] = ' || ibt(1)(i));
        end loop;
        dbms_output.put_line('ibt([4][5] = ' ||
ibt(4)(5));
        dbms_output.put_line('ibt([5][1] = ' ||
ibt(5)(1));
        dbms_output.put_line('ibt([6][2] = ' ||
ibt(6)(2));
        dbms_output.put_line('ibt([8][3] = ' ||
ibt(8)(3));
        dbms_output.put_line('ibt([3][4] = ' ||
ibt(3)(4));
        flag := ibt.exists(3);
        if flag = true then
            dbms_output.put_line('Index 3 exists');
        else
            dbms_output.put_line('Index 3 exists');
        end if;

```

```

        ibt.delete(1);
        dbms_output.put_line('After delete of first
index, Count = ' || ibt.count);
        ibt.delete(4);
        dbms_output.put_line('After delete of
fourth index, Count = ' || ibt.count);
        dbms_output.put_line('INDEX-BY TABLE
ELEMENTS');
        dbms_output.put_line('ibt([5][1] = ' ||
ibt(5)(1));
        dbms_output.put_line('ibt([6][2] = ' ||
ibt(6)(2));
        dbms_output.put_line('ibt([8][3] = ' ||
ibt(8)(3));
        dbms_output.put_line('ibt([3][4] = ' ||
ibt(3)(4));
        ibt.delete;
        dbms_output.put_line('After delete
of entire index-by table, Count =
' || ibt.count);
END;

```

Output:

```

Count = 0
No limit to Index-by Tables
INDEX-BY TABLE ELEMENTS
ibt([1][1] = a
ibt([4][5] = b
ibt([5][1] = c
ibt([6][2] = d
ibt([8][3] = e
ibt([3][4] = f
First Index = 1
Last Index = 8
Next Index = 4
Prior Index = 6
Count = 6
INDEX-BY TABLE ELEMENTS
ibt[1][1] = a
ibt[1][2] = g
ibt[1][3] = h

```

```

ibt[1][4] = i
ibt[1][5] = k
ibt[1][6] = l
ibt[1][7] = m
ibt[1][8] = n
ibt([4][5] = b
ibt([5][1] = c
ibt([6][2] = d
ibt([8][3] = e
ibt([3][4] = f
Index 3 exists
After delete of first index, Count = 5
After delete of fourth index, Count = 4
INDEX-BY TABLE ELEMENTS
ibt([5][1] = c
ibt([6][2] = d
ibt([8][3] = e
ibt([3][4] = f
After delete of entire index-by table, Count = 0

```

Ex4:

```

DECLARE
    type t1 is table of varchar(2) index by
binary_integer;
    type t2 is table of t1 index by
binary_integer;
    type t3 is table of t2;
    nt t3 := t3();
    c number := 65;
BEGIN
    nt.extend(2);
    dbms_output.put_line('Count = ' ||
nt.count);
    for i in 1..nt.count loop
        for j in 1..nt.count loop
            for k in 1..nt.count loop
                nt(i)(j)(k) := chr(c);
                c := c + 1;
            end loop;
        end loop;
    end loop;

```

```

                end loop;
            end loop;
        end loop;
        dbms_output.put_line('NESTED TABLE
ELEMENTS');
        for i in 1..nt.count loop
            for j in 1..nt.count loop
                for k in 1..nt.count loop
                    dbms_output.put_line(
                        'nt[' || i || '][' || j ||
'][' || k || '] = ' || nt(i)(j)(k));
                end loop;
            end loop;
        end loop;
END;

```

Output:

Count = 2

NESTED TABLE ELEMENTS

nt[1][1][1] = A

nt[1][1][2] = B

nt[1][2][1] = C

nt[1][2][2] = D

nt[2][1][1] = E

nt[2][1][2] = F

nt[2][2][1] = G

nt[2][2][2] = H

OBJECTS USED IN THE EXAMPLES

SQL> select * from student;

SNO	SNAME	SMARKS
1	saketh	100
2	srinu	200
3	divya	300
4	manogni	400

```
SQL> create or replace type addr as object(hno
number(2),city varchar(10));/
SQL> select * from employ;
```

ENAME	JOB	ADDRESS (HNO, CITY)
-----	-----	-----
-		
Ranjit	clerk	ADDR (11, 'hyd')
Satish	manager	ADDR (22, 'bang')
Srinu	engineer	ADDR (33, 'kochi')

ERROR HANDLING

- PL/SQL implements error handling with exceptions and exception handlers.
- Exceptions can be associated with oracle errors or with your own user-defined errors.
- By using exceptions and exception handlers, you can make your PL/SQL programs robust and able to deal with both unexpected and expected errors during execution.

ERROR TYPES

- **Compile-time errors**
- **Runtime errors**
- Errors that occur during the compilation phase are detected by the PL/SQL engine and reported back to the user, we have to correct them.

- Runtime errors are detected by the PL/SQL runtime engine which can programmatically raise and caught by exception handlers.
- Exceptions are designed for run-time error handling, rather than compile-time error handling.

HANDLING EXCEPTIONS

- When exception is raised, control passes to the exception section of the block.
- The exception section consists of handlers for some or all of the exceptions.
- An exception handler contains the code that is executed when the error associated with the exception occurs, and the exception is raised.

Syntax:

```
EXCEPTION
    When exception_name then
        Sequence_of_statements;
    When exception_name then
        Sequence_of_statements;
    When others then
        Sequence_of_statements;
END;
```

EXCEPTION TYPES

- Predefined exceptions
- User-defined exceptions

PREDEFINED EXCEPTIONS

- Oracle has predefined several exceptions that corresponds to the most common oracle errors.
- Like the predefined types, the identifiers of these exceptions are defined in the STANDARD package.
- Because of this, they are already available to the program, it is not necessary to declare them in the declarative section.

Ex1:

```

DECLARE
    a number;
    b varchar(2);
    v_marks number;
    cursor c is select * from student;
    type t is varray(3) of varchar(2);
    va t := t('a','b');
    val t;
BEGIN
    -- NO_DATA_FOUND
    BEGIN
        select smarks into v_marks from
student where sno = 50;
    EXCEPTION
        when no_data_found then
            dbms_output.put_line('Invalid
student number');
    END;
    -- CURSOR_ALREADY_OPEN
    BEGIN
        open c;
        open c;
    EXCEPTION
        when cursor_already_open then
            dbms_output.put_line('Cursor is
already opened');

```

```

        END;

        -- INVALID_CURSOR
        BEGIN
            close c;
            open c;
            close c;
            close c;
            EXCEPTION
                when invalid_cursor then
                    dbms_output.put_line('Cursor is
already closed');
            END;
        -- TOO_MANY_ROWS
        BEGIN
            select smarks into v_marks from
student where sno > 1;
            EXCEPTION
                when too_many_rows then
                    dbms_output.put_line('Too many values
are coming to marks variable');
            END;
        -- ZERO_DIVIDE
        BEGIN
            a := 5/0;
            EXCEPTION
                when zero_divide then
                    dbms_output.put_line('Divided by zero
- invalid operation');
            END;
        -- VALUE_ERROR
        BEGIN
            b := 'saketh';
            EXCEPTION
                when value_error then
                    dbms_output.put_line('Invalid string
length');
            END;
        -- INVALID_NUMBER
        BEGIN

```

```

        insert into student
values('a','srinu',100);
    EXCEPTION
        when invalid_number then
            dbms_output.put_line('Invalid
number');
    END;
    -- SUBSCRIPT_OUTSIDE_LIMIT
BEGIN
    va(4) := 'c';
    EXCEPTION
        when subscript_outside_limit then
            dbms_output.put_line('Index is
greater than the limit');
    END;
    -- SUBSCRIPT_BEYOND_COUNT
BEGIN
    va(3) := 'c';
    EXCEPTION
        when subscript_beyond_count then
            dbms_output.put_line('Index is
greater than the count');
    END;
    -- COLLECTION_IS_NULL
BEGIN
    val(1) := 'a';
    EXCEPTION
        when collection_is_null then
            dbms_output.put_line('Collection is
empty');
    END;
    --
END;

```

Output:

```

Invalid student number
Cursor is already opened
Cursor is already closed
Too many values are coming to marks variable
Divided by zero - invalid operation
Invalid string length

```

```
Invalid number
Index is greater than the limit
Index is greater than the count
Collection is empty
```

Ex2:

```
DECLARE
    c number;
BEGIN
    c := 5/0;
EXCEPTION
    when zero_divide then
        dbms_output.put_line('Invalid
Operation');
    when others then
        dbms_output.put_line('From OTHERS
handler: Invalid Operation');
END;

Output:
Invalid Operation
```

USER-DEFINED EXCEPTIONS

- A user-defined exception is an error that is defined by the programmer.
- User-defined exceptions are declared in the declarative section of a PL/SQL block.
- Just like variables, exceptions have a type EXCEPTION and scope.
- RAISING EXCEPTIONS User-defined exceptions are raised explicitly via the RAISE statement.

Ex:

```
DECLARE
```

```

        e exception;
BEGIN
    raise e;
EXCEPTION
    when e then
        dbms_output.put_line('e is
raised');
END;
Output:
e is raised

```

BULIT-IN ERROR FUNCTIONS

SQLCODE AND SQLERRM

- SQLCODE returns the current error code, and SQLERRM returns the current error message text;
- For user-defined exception SQLCODE returns 1 and SQLERRM returns "user-deifned exception".
- SQLERRM wiil take only negative value except 100. If any positive value other than 100 returns non-oracle exception.

Ex1:

```

DECLARE
    e exception;
    v_dname varchar(10);
BEGIN
    -- USER-DEFINED EXCEPTION
    BEGIN
        raise e;
    EXCEPTION
        when e then

dbms_output.put_line(SQLCODE || ' ' || SQLERRM);
END;

```

```

        -- PREDEFINED EXCEPTION
BEGIN
        select dname into v_dname from dept
where deptno = 50;
        EXCEPTION
        when no_data_found then

dbms_output.put_line(SQLCODE || ' ' || SQLERRM);
        END;
END;

```

Output:

```

1 User-Defined Exception
100 ORA-01403: no data found

```

Ex2:

```

BEGIN
        dbms_output.put_line(SQLERRM(100));
        dbms_output.put_line(SQLERRM(0));
        dbms_output.put_line(SQLERRM(1));
        dbms_output.put_line(SQLERRM(-100));
        dbms_output.put_line(SQLERRM(-500));
        dbms_output.put_line(SQLERRM(200));
        dbms_output.put_line(SQLERRM(-900));
END;

```

Output:

```

ORA-01403: no data found
ORA-0000: normal, successful completion
User-Defined Exception
ORA-00100: no data found
ORA-00500: Message 500 not found;  product=RDBMS;
facility=ORA
-200: non-ORACLE exception
ORA-00900: invalid SQL statement

```

DBMS_UTILITY.FORMAT_ERROR_STACK

The built-in function, like `SQLERRM`, returns the message associated with the current error.

It differs from `SQLERRM` in two ways:

Its length is not restricted; it will return the full error message string.

You can not pass an error code number to this function; it cannot be used to return the message for a random error code.

Ex:

```
DECLARE
    v number := 'ab';
BEGIN
    null;
EXCEPTION
    when others then

dbms_output.put_line(dbms_utility.format_error_stack);
END;
```

Output:

declare

*

ERROR at line 1:

ORA-06502: PL/SQL: numeric or value error:
character to number conversion error

ORA-06512: at line 2

DBMS_UTILITY.FORMAT_CALL_STACK

- This function returns a formatted string showing the execution call stack inside your PL/SQL application.

- Its usefulness is not restricted to error management;
- you will also find its handy for tracing the execution of your code.
- You may not use this function in exception block.

Ex:

```
BEGIN

dbms_output.put_line(dbms_utility.format_call_stack);
END;
```

Output:

```
----- PL/SQL Call Stack -----
```

Object_handle	line_number
object_name	
69760478	2
anonymous block	

DBMS_UTILITY.FORMAT_ERROR_BACKTRACE

- It displays the execution stack at the point where an exception was raised.
- Thus , you can call this function with an exception section at the top level of your stack and still find out where the error was raised deep within the call stack.

Ex:

```
CREATE OR REPLACE PROCEDURE P1 IS
BEGIN
    dbms_output.put_line('from procedure 1');
    raise value_error;
END P1;
```



```

CREATE OR REPLACE PROCEDURE P2 IS
BEGIN
    dbms_output.put_line('from procedure 2');
    p1;
END P2;

CREATE OR REPLACE PROCEDURE P3 IS
BEGIN
    dbms_output.put_line('from procedure 3');
    p2;
EXCEPTION
    when others then

dbms_output.put_line(dbms_utility.format_error_backtrace);
END P3;
Output:
SQL> exec p3

from procedure 3
from procedure 2
from procedure 1
ORA-06512: at "SAKETH.P1", line 4
ORA-06512: at "SAKETH.P2", line 4
ORA-06512: at "SAKETH.P3", line 4

```

EXCEPTION_INIT PRAGMA

- Using this you can associate a named exception with a particular oracle error.
- This gives you the ability to trap this error specifically, rather than via an OTHERS handler.

Syntax:

**PRAGMA EXCEPTION_INIT(exception_name,
oracle_error_number);**

Ex:

```
DECLARE
    e exception;
    pragma exception_init(e,-1476);
    c number;
BEGIN
    c := 5/0;
EXCEPTION
    when e then
        dbms_output.put_line('Invalid
Operation');
END;
```

Output:
Invalid Operation

RAISE_APPLICATION_ERROR

You can use this built-in function to create your own error messages, which can be more descriptive than named exceptions.

Syntax:

**RAISE_APPLICATION_ERROR(error_number,
error_message,, [keep_errors_flag]);**

- The Boolean parameter keep_errors_flag is optional.
- If it is TRUE, the new error is added to the list of errors already raised.
- If it is FALSE, which is default, the new error will replace the current list of errors.

Ex:

```
DECLARE
    c number;
BEGIN
    c := 5/0;

EXCEPTION
    when zero_divide then
        raise_application_error(-
20222,'Invalid Operation');
END;
```

Output:

```
DECLARE
*
ERROR at line 1:
ORA-20222: Invalid Operation
ORA-06512: at line 7
```

EXCEPTION PROPAGATION

Exceptions can occur in the declarative, the executable, or the exception section of a PL/SQL block.

EXCEPTION RAISED IN THE EXECUTABLE SECTION

Exceptions raised in executable section can be handled in current block or outer block.

Ex1:

```
DECLARE
    e exception;
BEGIN
    BEGIN
        raise e;
```

```

        END;
    EXCEPTION
        when e then
            dbms_output.put_line('e is raised');
END;

```

Output:
e is raised

Ex2:

```

DECLARE
    e exception;
BEGIN
    BEGIN
        raise e;
    END;
END;

```

Output:
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined
exception
ORA-06512: at line 5

EXCEPTION RAISED IN THE DECLARATIVE SECTION

Exceptions raised in the declarative section must be handled in the outer block.

Ex1:

```

DECLARE
    c number(3) := 'abcd';
BEGIN
    dbms_output.put_line('Hello');
EXCEPTION

```

```

        when others then
            dbms_output.put_line('Invalid string
length');
END;

```

Output:

```

ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error:
character to number conversion error
ORA-06512: at line 2

```

Ex2:

```

BEGIN
    DECLARE
        c number(3) := 'abcd';
    BEGIN
        dbms_output.put_line('Hello');
    EXCEPTION
        when others then
            dbms_output.put_line('Invalid
string length');
    END;
EXCEPTION
    when others then
        dbms_output.put_line('From outer block:
Invalid string length');
END;

```

Output:

```

From outer block: Invalid string length

```

EXCEPTION RAISED IN THE EXCEPTION SECTION

Exceptions raised in the declarative section must be handled in the outer block.

Ex1:

```

DECLARE
    e1 exception;
    e2 exception;
BEGIN
    raise e1;
EXCEPTION
    when e1 then
        dbms_output.put_line('e1 is
raised');
        raise e2;
    when e2 then
        dbms_output.put_line('e2 is
raised');
END;

```

Output:

e1 is raised

```

DECLARE

```

```

*

```

```

ERROR at line 1:

```

```

ORA-06510: PL/SQL: unhandled user-defined
exception

```

```

ORA-06512: at line 9

```

```

ORA-06510: PL/SQL: unhandled user-defined
exception

```

Ex2:

```

DECLARE
    e1 exception;
    e2 exception;
BEGIN
    BEGIN
        raise e1;
    EXCEPTION
        when e1 then

```

```

                                dbms_output.put_line('e1 is
raised');
                                raise e2;
                                when e2 then
                                    dbms_output.put_line('e2 is
raised');
                                END;
EXCEPTION
    when e2 then
        dbms_output.put_line('From outer block:
e2 is raised');
END;

```

Output:
e1 is raised
From outer block: e2 is raised

Ex3:

```

DECLARE
    e exception;
BEGIN
    raise e;
EXCEPTION
    when e then
        dbms_output.put_line('e is
raised');
        raise e;
END;

```

Output:
e is raised
DECLARE
*
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined
exception
ORA-06512: at line 8
ORA-06510: PL/SQL: unhandled user-defined
exception

RESTRICTIONS

You can not pass exception as an argument to a subprogram.

DATABASE TRIGGERS

- Triggers are similar to procedures or functions in that they are named PL/SQL blocks with declarative, executable, and exception handling sections.
- A trigger is executed implicitly whenever the triggering event happens.
- The act of executing a trigger is known as firing the trigger.

RESTRICTIONS ON TRIGGERES

- Like packages, triggers must be stored as stand-alone objects in the database and cannot be local to a block or package.
- A trigger does not accept arguments.

USE OF TRIGGERS

- Maintaining complex integrity constraints not possible through declarative constraints enable at table creation.
- Auditing information in a table by recording the changes made and who made them.
- Automatically signaling other programs that action needs to take place when chages are made to a table.
- Perform validation on changes being made to tables.
- Automate maintenance of the database.

TYPES OF TRIGGERS

- DML Triggers
- Instead of Triggers
- DDL Triggers
- System Triggers
- Suspend Triggers

CATEGORIES

Timing -- Before or After

Level -- Row or Statement

- Row level trigger fires once for each row affected by the triggering statement.
- Row level trigger is identified by the FOR EACH ROW clause.
- Statement level trigger fires once either before or after the statement.

DML TRIGGER SYNTAX

```
Create or replace trigger <trigger_name>
Before | after on insert or update or delete
[For each row]
Begin
    -- trigger body
End <trigger_name>;
```

DML TRIGGERS

- A DML trigger is fired on an INSERT, UPDATE, or DELETE operation on a database table.

- It can be fired either before or after the statement executes, and can be fired once per affected row, or once per statement.
- The combination of these factors determines the types of the triggers.
- These are a total of 12 possible types (3 statements * 2 timing * 2 levels).

ORDER OF DML TRIGGER FIRING

- Before statement level
- Before row level
- After row level
- After statement level

Ex:

Suppose we have a following table.

SQL> select * from student;

NO	NAME	MARKS
1	a	100
2	b	200
3	c	300
4	d	400

Also we have triggering_firing_order table with firing_order as the field.

```
CREATE OR REPLACE TRIGGER TRIGGER1
  before insert on student
BEGIN
  insert into trigger_firing_order
values('Before Statement Level');
```

```

END TRIGGER1;

CREATE OR REPLACE TRIGGER TRIGGER2
  before insert on student
  for each row
BEGIN
  insert into trigger_firing_order
values('Before Row Level');
END TRIGGER2;

CREATE OR REPLACE TRIGGER TRIGGER3
  after insert on student
BEGIN
  insert into trigger_firing_order values('After
Statement Level');
END TRIGGER3;

CREATE OR REPLACE TRIGGER TRIGGER4
  after insert on student
  for each row
BEGIN
  insert into trigger_firing_order values('After
Row Level');
END TRIGGER4;

```

Output:

```
SQL> select * from trigger_firing_order;
```

no rows selected

```
SQL> insert into student values(5,'e',500);
```

1 row created.

```
SQL> select * from trigger_firing_order;
```

```
FIRING_ORDER
```

```
-----
```

```
Before Statement Level
```

```
Before Row Level
```

```
After Row Level
```

After Statement Level

```
SQL> select * from student;
```

NO	NAME	MARKS
1	a	100
2	b	200
3	c	300
4	d	400
5	e	500

CORRELATION IDENTIFIERS IN ROW-LEVEL TRIGGERS

- Inside the trigger, you can access the data in the row that is currently being processed.
- This is accomplished through two correlation identifiers - **:old** and **:new**.
- A correlation identifier is a special kind of PL/SQL bind variable.
- The colon in front of each indicates that they are bind variables, in the sense of host variables used in embedded PL/SQL, and indicates that they are not regular PL/SQL variables.
- The PL/SQL compiler will treat them as records of type **Triggering_table%ROWTYPE**.
- Although syntactically they are treated as records, in reality they are not. :old and :new are also known as pseudorecords, for this reason.

TRIGGERING STATEMENT	:OLD	:NEW
-----	-----	-----

```

INSERT          all fields are NULL.
values that will be inserted

When
the statement is completed.

UPDATE          original values for      new
values that will be updated
                the row before the      when
the statement is completed.
                update.

DELETE          original values before
all fields are NULL.
                the row is deleted.

```

Ex:

Suppose we have a table called marks with fields no, old_marks, new_marks.

```

CREATE OR REPLACE TRIGGER OLD_NEW
  before insert or update or delete on student
  for each row
BEGIN
  insert into marks
values(:old.no,:old.marks,:new.marks);
END OLD_NEW;

```

Output:

SQL> select * from student;

NO	NAME	MARKS
----	-----	-----
1	a	100
2	b	200
3	c	300
4	d	400
5	e	500

SQL> select * from marks;
no rows selected

SQL> insert into student values(6,'f',600);

1 row created.

SQL> select * from student;

NO	NAME	MARKS
----	-----	-----
1	a	100
2	b	200
3	c	300
4	d	400
5	e	500
6	f	600

SQL> select * from marks;

NO	OLD_MARKS	NEW_MARKS
----	-----	-----
		600

SQL> update student set marks=555 where no=5;
1 row updated.

SQL> select * from student;

NO	NAME	MARKS
----	-----	-----
1	a	100
2	b	200
3	c	300
4	d	400
5	e	555
6	f	600

SQL> select * from marks;

NO	OLD_MARKS	NEW_MARKS
----	-----	-----
		600
5	500	555

SQL> delete student where no = 2;
1 row deleted.

SQL> select * from student;

NO	NAME	MARKS
1	a	100
3	c	300
4	d	400
5	e	555
6	f	600

SQL> select * from marks;

NO	OLD_MARKS	NEW_MARKS
		600
5	500	555
2	200	

REFERENCING CLAUSE

If desired, you can use the REFERENCING clause to specify a different name for :old and :new.

This clause is found after the triggering event, before the WHEN clause.

Syntax:

REFERENCING [old as old_name] [new as new_name]

Ex:

```
CREATE OR REPLACE TRIGGER REFERENCE_TRIGGER
before insert or update or delete on student
```

```

        referencing old as old_student new as
new_student
    for each row
BEGIN
    insert into marks

values (:old_student.no, :old_student.marks, :new_stu
dent.marks);
END REFERENCE_TRIGGER;

```

WHEN CLAUSE

WHEN clause is valid for row-level triggers only.

If present, the trigger body will be executed only for those rows that meet the condition specified by the WHEN clause.

Syntax:

WHEN trigger_condition;

Where trigger_condition is a Boolean expression.

It will be evaluated for each row. The :new and :old records can be referenced inside trigger_condition as well, but like REFERENCING, the colon is not used there. The colon is only valid in the trigger body.

Ex:

```

CREATE OR REPLACE TRIGGER WHEN_TRIGGER
    before insert or update or delete on student
    referencing old as old_student new as
new_student
    for each row
    when (new_student.marks > 500)
BEGIN
    insert into marks

```



```
values (:old_student.no, :old_student.marks, :new_student.marks);
END WHEN_TRIGGER;
```

TRIGGER PREDICATES

There are three Boolean functions that you can use to determine what the operation is.

The predicates are

- INSERTING
- UPDATING
- DELETING

Ex:

```
CREATE OR REPLACE TRIGGER PREDICATE_TRIGGER
    before insert or update or delete on
student
BEGIN
    if inserting then
        insert into predicates values('I');
    elsif updating then
        insert into predicates values('U');
    elsif deleting then
        insert into predicates values('D');
    end if;
END PREDICATE_TRIGGER;
```

Output:

```
SQL> delete student where no=1;
```

```
1 row deleted.
```

```
SQL> select * from predicates;
```

```
MSG
```

D

SQL> insert into student values (7,'g',700);

1 row created.

SQL> select * from predicates;

MSG

D

I

SQL> update student set marks = 777 where no=7;

1 row updated.

SQL> select * from predicates;

MSG

D

I

U

INSTEAD-OF TRIGGERS

Instead-of triggers fire instead of a DML operation.
Also, instead-of triggers can be defined only on views.

Instead-of triggers are used in two cases:

- To allow a view that would otherwise not be modifiable to be modified.
- To modify the columns of a nested table column in a view.

