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 datatypen*);

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 .. <*conditionn*>;

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 .. <*conditionn*>;

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;

NO	NAME	MARKS

```
1   Sudha          100
```

ii) This will give the rows whose name start with 'S'.

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

NO NAME          MARKS
--- -------      ---------
1   Sudha         100
2   Saketh        200
```

iii) This will give the rows whose name ends with 'h'.

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

NO NAME          MARKS
--- -------      ---------
2   Saketh        200
3   Ramesh
```

iV) This will give the rows whose name's second letter start with 'a'.

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

NO NAME          MARKS
--- -------      --------
2   Saketh        200
1   Jagan         300
2   Naren         400
3   Ramesh
4   Madhu
6   Rattu
```

V) This will give the rows whose name's third letter start with 'd'.

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

NO NAME        MARKS
--- -------    ---------
1   Sudha       100
4   Madhu
```

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

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

NO NAME        MARKS
--- -------    ---------
2   Saketh      200
6   Rattu
```

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

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

NO NAME        MARKS
--- -------    ---------
2   Saketh      200
3   Ramesh
```

Viii) This will give the rows whose name cotains 2 a's.

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

NO NAME        MARKS
--- -------    ----------
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.

d) 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.

e) 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 form 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 <orginal_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

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

-- This inserts 3 rows
```

c) MULTI INSERT WITH SPECIFIED FIELDS

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

-- This inserts 3 rows
```

d) MULTI INSERT WITH DUPLICATE ROWS

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

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

e) MULTI INSERT WITH CONDITIONS BASED

```
SQL> Insert all
        When deptno > 10 then
        Into student1 values(1,'a',100)
        When dname = 'SALES' then
        Into student2 values(2,'b',200)
        When loc = 'NEW  YORK' then
        Into student3 values(3,'c',300)
        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.
```

f) MULTI INSERT WITH CONDITIONS BASED AND ELSE

```
SQL> Insert all
        When deptno > 100 then
        Into student1 values(1,'a',100)
        When dname = 'S' then
        Into student2 values(2,'b',200)
        When loc = 'NEW YORK' then
        Into student3 values(3,'c',300)
        Else
        Into student values(4,'d',400)
```

```
Select *from dept where deptno>10;
```

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

g) MULTI INSERT WITH CONDITIONS BASED AND FIRST

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

h) MULTI INSERT WITH CONDITIONS BASED, FIRST AND ELSE

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

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

i) MULTI INSERT WITH MULTIBLE TABLES

```
SQL> Insert all
       Into student1 values(1,'a',100)
       Into student2 values(2,'b',200)
       Into student3 values(3,'c',300)
       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) POWER(NULL,NULL) POWER(2,-5)
-------------  -------------  ----- --------- ----------------------  --------------
     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)   EXP(-2)
---------- ---------- ---------- ------------ ----------
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) LOG(10,NULL)
-----------  ----------  ----------  ----------
         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          1           0           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          1           0             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,'*'), rpad('computer',15,'*#') from dual;

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

-- Default padding character was blank space.

f) LPAD

This will allows 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
                ----------  -------  --------
                omputer  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 orginal string length.
- If *start_chr_count* is negative then it will extract the substring from right end.

1	2	3	4	5	6	7	8
C	O	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 fmmonth 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'))
---------------------------------------------------------------------------------
                                                7
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 chops 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

d) 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)WITHINGROUP(ORDERBYSALDESC)

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;

            SUM(SAL)
            ----------
               38600
```

b) AVG

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

Syntax: avg (*column*)

Ex:

```
SQL> select avg(sal) from emp;

            AVG(SAL)
```

```
---------------
2757.14286
```

c) MAX

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

Syntax: max (*column*)

Ex:

```
SQL> select max(sal) from emp;

 MAX(SAL)
----------
  5000
```

d) MIN

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

Syntax: min (*column*)

Ex:

```
SQL> select min(sal) from emp;

MIN(SAL)
----------
  500
```

e) COUNT

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

Syntax: count (*column*)

Ex:

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

COUNT(SAL)	COUNT(*)
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 childs exists.

USING ON DELTE CASCADE

By using this clause you can remove the parent record even it childs exists.
Because when ever 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
     un unique(no,name));
```

UNIQUE (ALTER LEVEL)

```
SQL> alter table student add unique(no,name);
SQL> alter table student add constraint un 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 user 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 with in oracle.
When you query the table, oracle reads the external table and returns the results just as if the data had been stored with in 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
Users 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
a) Indexing not possible
b) Constraints not possible

BENEFITS OF EXTERNAL TABLES

a) Queries of external tables complete very quickly even though a full table scan id 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 wih 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:

a) 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));
```

b) 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);
```

c) Create the type

```
SQL> create or replace type stud as object(no number(2),name varchar(2),marks
     number(3));/
```

d) Generating OIDs

```
SQL> Create or replace view student1_ov of stud with object identifier(or id) (no) as
     Select * from Student1;
```

e) Generating references

```
SQL> Create or replace view student2_ov as select no,hno,city,make_ref(student1_ov,id)
     id from Student2;
```

d) Query the following

```
SQL> select *from student1_ov;
SQL> select ref(s) from student1_ov s;
SQL> select values(s) from student1_ov;
SQ> 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

a) 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));
```

b) 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
```

c) Retrieving records from list partitioned table

```
SQL> Select *from student;
SQL> Select *from student partition(p1);
```

d) Possible operations with list partitions

- Add
- Drop
- Truncate
- Rename
- Move
- Exchange

e) Adding a partition

SQL> Alter table student add partition p5 values(21,22,23,24,25);

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) Exchanging a partition

SQL> Alter table student exchange partition p1 with table student2;

j) 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.

a) 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.

b) 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**

c) Retrieving records from subpartitioned table

SQL> **Select *from student;**

SQL> **Select *from student partition(p1);**

SQL> **Select *from student subpartition(sys_subp1);**

d) Possible operations with subpartitions

- Add
- Drop
- Truncate
- Rename
- Split

e) Adding a partition

SQL> **Alter table student add partition p3 values less than(30);**

f) Dropping a partition

SQL> **Alter table student drop partition p3;**

g) Renaming a partition

SQL> **Alter table student rename partition p2 to p3;**

h) Truncate a partition

SQL> **Alter table student truncate partition p1;**

i) 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 wih the total salary fot 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 bty n] [start with n] [maxvalue n] [minvalue n] [cycle/nocycle] [cache/nocache];

By defalult the sequence starts with 1, increments by 1 with minvalue of 1 and with nocycle, nocache.

Cache option pre-alloocates 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;

     EMPNO ENAME      JOB       MGR     DEPTNO
---------- ---------- ---------- ---------- ----------
       111 saketh     analyst         444         10
```

222	sudha	clerk	333	20
333	jagan	manager	111	10
444	madhu	engineer	222	40

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       hyd
       111 saketh     analyst    fin       bang
       222 sudha      clerk      fin       bang
       333 jagan      manager    fin       bang
       444 madhu      engineer   fin       bang
       111 saketh     analyst    hr        bombay
       222 sudha      clerk      hr        bombay
       333 jagan      manager    hr        bombay
       444 madhu      engineer   hr        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	hyd
222	sudha	clerk	fin	bang
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
111	saketh	analyst	mkt	hyd
333	jagan	manager	mkt	hyd
222	sudha	clerk	fin	bang
			hr	bombay

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
---------- ---------- --------- ---------- ----------- ---------- ---------- ----------
      7788 SCOTT      ANALYST         7566 19-APR-87         3000                    20
      7839 KING       PRESIDENT            17-NOV-81         5000                    10
      7902 FORD       ANALYST         7566 03-DEC-81         3000                    20
```

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 comparision operator and the subquery.

Ex:

```
SQL> select * from emp where sal > any (select sal from emp where sal between 2500 and
     4000);

     EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM     DEPTNO
---------- ---------- --------- ---------- ----------- --------- ---------- ----------
      7566 JONES      MANAGER         7839 02-APR-81        2975                    20
      7788 SCOTT      ANALYST         7566 19-APR-87        3000                    20
      7839 KING       PRESIDENT            17-NOV-81        5000                    10
      7902 FORD       ANALYST         7566 03-DEC-81        3000                    20

SQL> select * from emp where sal > all (select sal from emp where sal between 2500 and
     4000);

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

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 tow 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

The default type of index used in an oracle database is the btree index. A btree index is designed to provide both rapid access to individual rows and quick access to groups of rows within a range. The btree index does this by performing a succession of value comparisons. 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 snall 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

A reverse key index when compared to standard index, reverses each byte of the column being indexed while keeping the column order. 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. 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.
view

Ex:

```
SQL> create index stud_ind on student(upper(sname));
```

DESCENDING INDEX

The order used by B-tree indexes has been ascending order. You can categorize data in B-tree index in descending order as well. This feature can be useful in applications where sorting operations are required.

Ex:

```
SQL> create index stud_ind on student(sno desc);
```

TEXT INDEX

Querying text is different from querying data because words have shades of meaning, relationships to other words, and opposites. You may want to search for words that are near each other, or words that are related to thers. These queries would be extremely difficult if all you had available was the standard relational operators. By extending SQL to include text indexes, oracle text permits you to ask very complex questions about the text.

To use oracle text, you need to create a *text index* on the column in which the text is stored. 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 fo 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 comman. 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 QEURY 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 hae 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

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. The misspellings can be either in the searched text or in the search string specified by the user during the query.

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;

It does, however, contains the word 'harvest'. A fuzzy match will return the books containing the word 'harvest' even though 'harvest' has a different word stem thant the word used as the search term.

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

SOUNDEX, expands search terms based on how the word sounds. The SOUNDEX expansion method uses the same text-matching logic available via the SOUNDEX function in SQL.

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

An index-organized table keeps its data sorted according to the primary key column values for the table. Index-organized tables store their data as if the entire table was stored in an index. 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 separte index for each partition.**
- **In the local prefixed index the partition key is specified on the left prefix. When the underlying table is partitioned baes 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 follwing 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 COMMNANDS

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

     EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM     DEPTNO
---------- ---------- --------- ---------- --------- ---------- ---------- ----------
      7782 CLARK      MANAGER         7839 09-JUN-81       2450                    10
      7839 KING       PRESIDENT            17-NOV-81       5000
      7934 MILLER     CLERK           7782 23-JAN-82       1300
                                                        ----------           **********
                                                              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 adds 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> sho 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 correspoinding 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 your 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 shows 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 priting 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 headsetp !
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 then by hand. You can put any command in login.sql that you can use in SQLPLUS, including SQLPLUS commands and SQL statements. All ot 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);
```