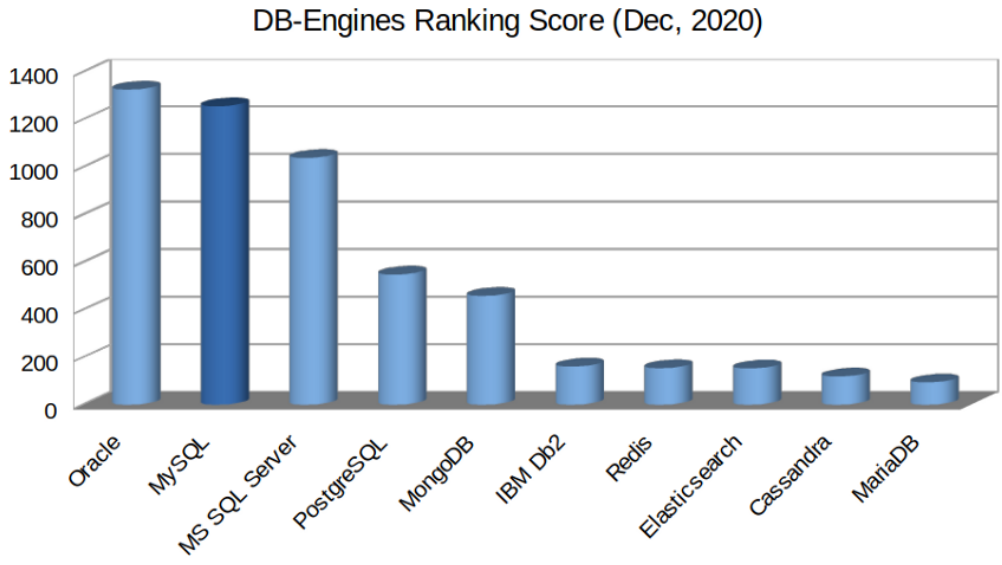
Oracle SQL - INTRODUCTION

Databases are the cornerstone of any Software Applications. You will need one or more databases to develop almost all kind of Software Applications: Web, Enterprise, Embedded Systems, Real-Time Systems, AI, ML, HPC, Blockchain, IoT, and many other applications.
With the rise of Microservices, Cloud, Distributed Applications, Global Scaling, Semi-Structured Data, Big Data, Fast Data, Low Latency Data: the traditional **SQL** databases are now joined by
various **NoSQL. NewSQL,** and **Cloud** databases.

There are a whopping **343** databases at present. Here I will list popular databases from them



Different databases in the market:

Oracle
MS SQL Server
Teradata
IBM DB2
Sybase
MySQL
PostgreSQL
Natezza

Oracle Database

When **Edgar F. Codd**'s published his revolutionary paper "**A Relational Model of Data for Large Shared Data Banks**" (1970) on the Relational Database Management System (RDBMS), it has completely changed the landscape of database Systems. The paper particularly inspired a young Software Engineer **Larry Ellison** (current CTO of Oracle Corporation). He later created the world's first commercially available RDBMS system **Oracle** in 1979. Since then, Oracle remained the leading commercial RDMBS System and dominated the Unix and Linux Systems. Over the last 41 years, Oracle has evolved with time and contributed to the RDBMS and the overall database Systems innovations.

Currently, Oracle is the number one commercially supported database and one of the widely used RDBMS overall. Its latest release (21.c) has added many innovative features that will make it an attractive option in the coming years.

5 Key Features

- Proprietory RDBMS.
- Offers ACID transactional guarantee. In terms of CAP, it offers immediate Consistency as a single Server.
- Advanced Multi-Model databases supporting Structured Data (SQL), Semi-Structured Data(JSON, XML), Spatial Data, and RDF Store. Offers multiple access pattern depending on the specific Data Model
- Offers Blockchain Tables.
- Supports both OLTP and OLAP workload.

When to Use Oracle

- If a company wants to have a Converged database or Master Database (One database for OLTP and OLAP).
- Traditional transactional workloads with structured (SQL) data, and when ACID transaction guarantee is a key criterion.
- Blockchain Table is required.
- For Data Warehousing.
- A multi-model database including Geospatial Data type is an essential requirement.

When not to Use Oracle

- If a company wants to save money on a database.
- Multi-Master ACID transaction is a must-have feature.
- Data is Semi-structured, i.e., JSON data with advanced query functions.
- Data is extremely relational (e.g., Social Media), i.e., Graph like data.

Oracle As a Service

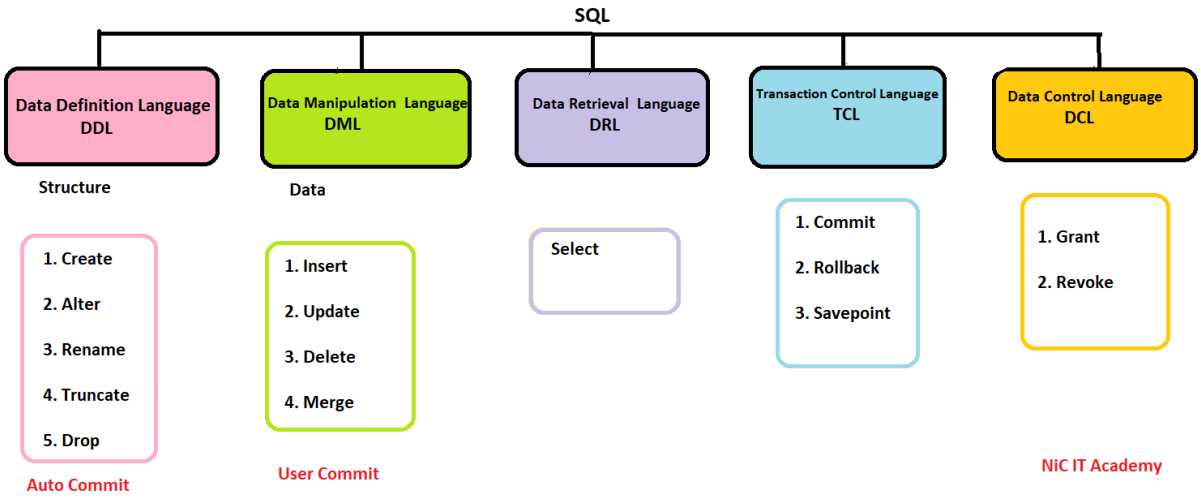
- Oracle Converged Database
- Amazon RDS for Oracle

In the past, almost all databases were relational. They used a set data structure, which allowed them to link information from different “tables”, using indexes. These data “buckets” could then be linked through a “relationship”. SQL (Structured Query Language) is the language used for this kind of databases. It provides commands to create, retrieve, update, and delete information stored in the tables.

NoSQL, then, stands for “No Structured Query Language”. It is a non-relational type of database. In this case, databases do not use any kind of relational enforcement. The architect of the database determines what relationships, if any, are necessary for their data, and creates them.

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;

NG 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.