READING MATERIAL 2

SELECT Statement

The SELECT statement is the most commonly used statement in SQL and is used to retrieve information already stored in the database. To retrieve data, you can either select all the column values or name specific columns in the SELECT clause to retrieve data.

Syntax of SELECT statement

```
SELECT  [distinct] <tablename.columname >
FROM  <tablename >
[where <condition>]
[group by <columnname(s)>]
[having <condition>]
[order by <expression>] ;
```

The main components of a SELECT statement are the SELECT clause and FROM clause. In this chapter, we will use basic syntax of SELECT statement with WHERE and ORDER BY clause while GROUP BY and HAVING clause will be discussed in subsequent chapters.

A SELECT clause contains the list of columns or expressions containing data you want to see. The FROM clause tells Oracle which database table is used to retrieve the information. Rest all the clauses are optional. The end of the statement is indicated by the semicolon "**;**" .

The simple form of a SELECT statement is:

```
SELECT (column_name1, column_name2,…… column_nameN) FROM tablename;
```

Ques: How do you list the EMPNO and ENAME from EMP table?

If you know the column names and the table name, writing the query is very simple. Execute the query by ending the query with a semicolon. In SQL*Plus, you may write the query as follows:

SQL> *SELECT empno, ename FROM emp;*

OUTPUT:

EMPNO	ENAME
7369	SMITH
7499	ALLEN
7521	WARD
……	……

14 rows selected.

Use of Wildcard(*) character

The asterisk (*) is used to select all columns in the table. This is very useful when we do not know the column names or when we are too lazy to type all column names.

SQL> *SELECT * FROM emp;*

OUTPUT:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7876	ADAMS	CLERK	7788	23-MAY-87	1100		20
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

14 rows selected

The statement uses a wildcard (*) character, which indicates to Oracle that you want to view data from every column in the table.

We can also provide a column alias. The column alias name is defined next to the column name. Consider the following query, it provides an alias name for the column ENAME. Here, the column heading will be displayed as EMPLOYEE_NAME.

SQL> *SELECT empno, ename employee_name FROM emp;*

OUTPUT:

EMPNO	EMPLOYEE_N
7369	SMITH
7499	ALLEN
7521	WARD
…..	…….

14 rows selected.

If you want a space in the column alias name, you must enclose it in double quotation marks. The case is preserved only when the alias name is enclosed in double quotes; otherwise, it will display it in the uppercase.

The following example demonstrates using an alias name for the ENAME column.

SQL> *SELECT empno, ename "Employee Name" FROM emp;*

OUTPUT:

It will display the column heading like:

EMPNO	Employee N

Elimination of Duplication with DISTINCT Clause

The DISTINCT keyword followed by the SELECT keyword ensures that the resulting rows are unique. Uniqueness is verified according to the columns specified in query.

Syntax:

SQL> *SELECT DISTINCT column_name(s) FROM tablename;*

If you need to find the unique departments and salaries from EMP table then, issue the following query:

SQL> *SELECT DISTINCT deptno, sal FROM emp;*

OUTPUT:

DEPTNO	SAL
10	1300
10	2450
10	5000
20	800
20	1100
20	975
20	3000
30	950
30	1250
30	1500
30	1600
30	2850

12 rows selected.

**In this case the combination of deptno and sal should be unique.

READING MATERIAL 3

The Oracle table 'DUAL'

DUAL is a dummy table available to all users in the database. It has one column and one row. Besides arithmetic calculations, it also supports data retrieval and its formatting can be retrieved with following query.

Example:

SQL> *SELECT * FROM dual;*

OUTPUT:

D
X

We observe that column D is varchar2 (1) and X is a character data.

There is no meaningful data actually in DUAL; it simply exists as a SQL construct. The DUAL table contains only one column called DUMMY and one row with a value, "X." Often a simple calculation needs to be done, for example 2*2. The only SQL verb to get the output is SELECT. However a SELECT must have a table name in its FROM clause, otherwise the SELECT fails. When an arithmetic exercise is to be performed such as 2*2 or 4/2 etc, there really no table being referenced; only numeric literals are being used.

To facilitate such calculations via a SELECT, Oracle provides a dummy table called DUAL, against which SELECT statements that are required to manipulate numeric literals can be fired and output is obtained.

Example:

SQL> *SELECT 2*2 FROM dual;*

OUTPUT:

2*2
4

The current date can be obtained from the table DUAL in the required format as shown below.

Sysdate is a pseudo column that contains the current date and time. It required no arguments when selected from the table DUAL and returns the current date.

Example:

SQL> *SELECT sysdate FROM dual;*

OUTPUT:

SYSDATE
23-JUL-03

LIMITING ROWS

While viewing the data from the table it is not necessary that we want to see all the data from the columns selected but we want that there should be some restriction on the selected data. We can restrict the rows returned from the query by using the WHERE clause. When there is where clause in the statement, it checks each row of the table to determine whether the condition is met or not. The WHERE clause contains the condition which consists of any arithmetic expression, column values or functions etc. Moreover, it is not necessary that the columns used in the condition should be present in the SELECT statement.

Note: It is important to provide character and date values in single quotes in WHERE condition and character values which we supply in single quotes are case sensitive.

We can use column names or expressions in the WHERE clause but not column alias names. How do you list the employee information for department 10? This example shows how to use a WHERE clause to limit the query only to the records belonging to department 10.

Example: Retrieve the empno, ename, sal of deptno 10

SQL> *SELECT empno,ename,sal FROM emp WHERE deptno = 10;*

OUTPUT:

EMPNO	ENAME	SAL
7782	CLARK	2450
7839	KING	5000
7934	MILLER	1300

Example: List the empno, ename, job, manager number of the employees who do the job of MANAGER and belong to the department 20.

SQL> *SELECT empno,ename,job,mgr FROM emp WHERE job = 'MANAGER' and deptno = 20;*

OUTPUT:

EMPNO	ENAME	JOB	MGR
7566	JONES	MANAGER	7839

Example: List the employee number of employees who earn greater than 2000.

SQL> *SELECT empno,sal FROM emp WHERE sal>2000;*

OUTPUT:

EMPNO	SAL
7566	2975
7698	2850
7782	2450
7788	3000
7839	5000
7902	3000

Example: List the name of the employees who do the job of clerk or salesman.

SQL> *SELECT ename, job FROM emp WHERE job = 'CLERK' or job = 'SALESMAN'.*

OUTPUT:

ENAME	JOB
SMITH	CLERK
ALLEN	SALESMAN
WARD	SALESMAN
MARTIN	SALESMAN
TURNER	SALESMAN
ADAMS	CLERK
JAMES	CLERK
MILLER	CLERK

Example: List all the department names in BOSTAN from the dept table.

SQL> *SELECT dname FROM dept WHERE loc = 'BOSTON';*

OUTPUT:

DNAME
OPERATIONS

SPECIAL OPERATORS

There are following special operators used in SELECT statement:

Special Operators	
LIKE	Pattern matching from a column
IN	To check a value within a set
BETWEEN	To check a value within a range

Example: List the empno, ename and salary of the employees, whose salary is between 1500 and 2000.

SQL> *SELECT empno, ename, sal FROM emp WHERE sal between 1500 and 2000;*

OUTPUT:

EMPNO	ENAME	SAL
7499	ALLEN	1600
7844	TURNER	1500

** While using the between operator it must be remembered that both the values will be included in the range.

Example: List employee number and name of the employees who belong to department 10, 20.

SQL> *SELECT empno, ename FROM emp WHERE deptno in (10, 20);*

OUTPUT:

EMPNO	ENAME
7369	SMITH
7566	JONES
7782	CLARK
7788	SCOTT
7839	KING
7876	ADAMS
7902	FORD
7934	MILLER

Example: List employee number of the employees, whose name is not 'FORD', 'JAMES 'or 'JONES';

SQL> *SELECT empno FROM emp WHERE ename not in ('FORD','JAMES','JONES');*

OUTPUT:

EMPNO
7369
7499
7521
7654
7698
7782
7788
7839
7844
7876
7934

Importance of parentheses

While writing queries care should be taken to use the parentheses carefully so as not to change the meaning of the query.

SQL> *SELECT ename, job, comm FROM emp WHERE comm IS NULL and job ='CLERK ' or job = 'SALESMAN';*

OUTPUT:

ENAME	JOB	COMM
ALLEN	SALESMAN	300
WARD	SALESMAN	500
MARTIN	SALESMAN	1400
TURNER	SALESMAN	0

In this example the query becomes list the name of the employees, job and commission of those employees who get no commission and do the job of clerk or those employees who do the job of salesman only. Similarly if we write the query as:

SQL> *SELECT ename, job, comm FROM emp WHERE comm IS NULL and (job =’CLERK ‘ or job = ‘SALESMAN’);*

OUTPUT:

ENAME	JOB	COMM
SMITH	CLERK	
ADAMS	CLERK	
JAMES	CLERK	
MILLER	CLERK	

In this example the query becomes list the name of the employees, job and commission of those employees who do the job of either clerk or salesman and get no commission. So only those names will be displayed who is clerk or salesman but both of them should not be getting any commission.

READING MATERIAL 4

Working with NULL Values

Sometimes, a query for some information produces no result. In database terms, no result or nothing is called NULL. Null does not mean 0 or blank. Since NULL means no value or nothing or unknown value, it cannot be compared using relational or logical operators. Therefore you cannot test NULL value with equal to (=) operator, as NULL value cannot be equal to or unequal to any value. This value can be inserted into column of any data type. Generally, null value is used when the actual value is not known. We use the special operator 'IS' with the keyword NULL to find NULL values in the table.

Example: List the employee number and name who do not get any commission.

SQL> *SELECT empno, ename FROM emp WHERE comm IS NULL;*

OUTPUT:

EMPNO	ENAME
7369	SMITH
7566	JONES
7698	BLAKE
7782	CLARK
7788	SCOTT
7839	KING
7876	ADAMS
7900	JAMES
7902	FORD
7934	MILLER

Example: List the employee name and job of employees who do not report to anybody.

SQL> *SELECT ename, job FROM emp WHERE mgr IS NULL;*

OUTPUT:

ENAME	JOB
KING	PRESIDENT

Example: List the employee number and salary of the employees whose salary is greater than 2000 and are getting no commission.

SQL> *SELECT empno, sal FROM emp WHERE sal >2000 and comm IS NULL;*

OUTPUT:

EMPNO	SAL
7566	2975
7698	2850
7782	2350
7788	3000
7839	5000
7902	3000

ORDER BY Clause

Generally the result of the query is in unsorted form. To get the sorted result we use the ORDER BY clause. This clause will allow you to get the result of the query sorted either in ascending or descending order. The default order is ascending.

The syntax is:

```
SQL>
SELECT <column name(s) >
FROM <table name(s) >
[WHERE < condition(s)>]
[ORDER BY <column(s), expression> ] [ asc| desc]
```

Example: List the employee name and salary and sort them in descending order of their salary.

SQL> SELECT ename, sal FROM emp ORDER BY sal desc;

OUTPUT:

ENAME	SAL
KING	5000
SCOTT	3000
FORD	3000
JONES	2975
BLAKE	2850
CLARK	2450
ALLEN	1600
…..	……

Example: List the employee name and hiredate of the employees who do the job of clerk. Sort them in ascending order of their hiredate.

SQL> *SELECT ename, hiredate FROM emp WHERE job = 'CLERK' ORDER BY hiredate asc;*

OUTPUT: ** 'asc' is optional.

ENAME	HIREDATE
SMITH	17-DEC-80
JAMES	03-DEC-81
MILLER	23-JAN-82
ADAMS	23-MAY-87

Example: List the employee name, job and salary details and order them by their salary in descending order.

SQL> *SELECT ename,job,sal FROM emp ORDER BY sal desc;*

OUTPUT:

ENAME	JOB	SAL
KING	PRESIDENT	5000
SCOTT	ANALYST	3000
FORD	ANALYST	3000
JONES	MANAGER	2975
BLAKE	MANAGER	2850
CLARK	MANAGER	2450
ALLEN	SALESMAN	1600
TURNER	SALESMAN	1500
MILLER	CLERK	1300
WARD	SALESMAN	1250
MARTIN	SALESMAN	1250
ADAMS	CLERK	1100
JAMES	CLERK	950
SMITH	CLERK	800

Example: List the name, salary and department number of the employees and order them by their salary in descending order.

SQL> *SELECT ename, sal, deptno FROM emp ORDER BY 2 desc;*

OUTPUT:

ENAME	SAL	DEPTNO
KING	5000	10
SCOTT	3000	20
FORD	3000	20
JONES	2975	20
BLAKE	2850	30
CLARK	2450	10
ALLEN	1600	30
TURNER	1500	30
MILLER	1300	10
WARD	1250	30
MARTIN	1250	30
ADAMS	1100	20
JAMES	950	30
SMITH	800	20

Order by clause can be used to sort multiple numbers of columns (the maximum is the columns of a given table). Moreover it is not necessary that the order by clause can be used only with the selected columns.

Example: List the employee name, salary and department number and sort the result by their department number and salary in descending order.

SQL> *SELECT ename, sal, deptno FROM emp ORDER BY deptno desc, sal desc;*

OUTPUT:

ENAME	SAL	DEPTNO
BLAKE	2850	30
ALLEN	1600	30
TURNER	1500	30
WARD	1250	30
MARTIN	1250	30
JAMES	950	30
SCOTT	3000	20
FORD	3000	20
JONES	2975	20
ADAMS	1100	20
SMITH	800	20
KING	5000	10
CLARK	2450	10
MILLER	1300	10

Here, the output is in the descending order of deptno and within same deptno it is in descending order of sal.

Example: List the employee number, names and hiredate and sort the result by the employee number and then by their hire dates.

SQL> *SELECT empno, ename, hiredate FROM emp ORDER BY empno, hiredate asc;*

OUTPUT:

EMPNO	ENAME	HIREDATE
7369	SMITH	17-DEC-80
7499	ALLEN	20-FEB-81
7521	WARD	22-FEB-81
7566	JONES	02-APR-81
…..	…..	…..

** NULL values come at the end of the output of the query in case we use the ORDER BY clause.

READING MATERIAL 5

Changing Column Headings with Column Aliases

As we discussed earlier, Oracle reprints the column name exactly as it was included in the select statement, including functions if there are any. Unfortunately, this method usually leaves you with a bad description of the column data, compounded by the fact that Oracle truncates the expression to fit a certain column length corresponding to the datatype of the column returned. Fortunately, Oracle provides a solution to this situation with the use of column aliases in the select statement. Any column can be given another name by you when the select statement is issued. This feature gives you the ability to fit more descriptive names into the space allotted by the column datatype definition.

SQL> SELECT ename "Employee name", Sal* 12 "Annual Salary" FROM emp;

OUTPUT:

Employee n	**Annual Salary**
SMITH	9600
ALLEN	19200
WARD	15000
JONES	35700
MARTIN	15000
…….	……..

14 rows selected.

**Notice that the column heading in the output is exactly the same as the column alias.

In order to specify an alias, simply name the alias after identifying the column to be selected, with or without an operation performed on it, separated by white space. Alternatively, you can issue the as keyword to denote the alias. By default, alias headings appear in the uppercase. If the alias contains spaces, special characters (such as # or $), or is case sensitive, enclose the alias in the double quotation marks(" ").

SQL> *SELECT ename as "employee name", sal salary FROM emp;*

OUTPUT:

Employee n	**SALARY**
SMITH	9600
ALLEN	19200
WARD	15000
JONES	35700
MARTIN	15000
…….	……..

14 rows selected

Column aliases are useful for adding meaningful headings to output from SQL queries. As shown, aliases can be specified in two ways: either by naming the alias after the column specification separated by white space, or with the use of the as keyword to mark the alias more clearly.

Column Concatenation

Columns can be combined together to produce more interesting or readable output. The method used to merge the output of two columns into one is called concatenation. The concatenation operator is two pipe characters put together, i.e. ||.

In this way we can increase the meaningfulness of the output produced. For good measure, the use of column aliases is recommended in order to make the name of the concatenated columns more meaningful.

Example: In order to display the output in English like sentences we may concatenate two or more columns of the table even with the constant strings as shown below:

SQL> *SELECT 'empno of '||ename ||' is '|| empno FROM emp;*

OUTPUT:

'EMPNOOF'||ENAME||'IS'||EMPNO
empno of SMITH is 7369
empno of ALLEN is 7499
empno of WARD is 7521
……..

14 rows selected

Pattern Matching

In order to select rows that match a particular character pattern we use the LIKE operator. This character matching operation is called as wildcard search. The following symbols are used for matching the pattern % (percentage). This symbol represents any sequence of zero or more characters . __ (underscore) this symbol is used for any single character search. The % and _ symbols can be used in any combination with literal characters.

Pattern matching symbols	**Description**
%(percentage)	represents any sequence of zero or more characters
_(underscore)	it is used for any single character search.

Example: List the employee names and empno whose names start with J.

SQL> SELECT ename, empno FROM emp WHERE ename like 'J%';

OUTPUT:

ENAME	EMPNO
JONES	7566
JAMES	7900

Example: List the employee names whose names are of six characters length.

SQL> *SELECT ename FROM emp WHERE ename like '_ _ _ _ _ _ ';*

OUTPUT:

ENAME
MARTIN
TURNER
MILLER

Example: List the employee names whose names end with ‘h’.

SQL> *SELECT ename FROM emp WHERE ename like ‘%H’;*

Output:

ENAME
SMITH

Example: List the employee names having D as the second character.

SQL> *SELECT ename FROM emp WHERE ename LIKE ‘_D%’;*

OUTPUT:

ENAME
ADAMS

Example: List the employee names having two A in their name.

SQL> *SELECT ename FROM emp WHERE ename LIKE ‘%A%A%’;*

OUTPUT:

ENAME
ADAMS

Accepting Values at Runtime

To create an interactive SQL command, define variables in the SQL commands. This allows the user to supply values at runtime, further enhancing the ability to reuse your scripts. SQL*Plus lets you define variables in your scripts. An ampersand (&) followed by a variable name prompts for and accepts values at runtime. For example, look at the following SELECT statement that queries the EMP table based on the department number supplied at runtime:

SQL> *SELECT empno, ename FROM emp WHERE deptno = &detpt;*

OUTPUT:

Enter value for dept: 10

old 3: WHERE deptno = &dept

new 3: WHERE deptno = 10

EMPNO	ENAME
7782	CLARK
7839	KING
7934	MILLER

3 rows selected.

** The old line with the variable and the new line with the substitution are displayed. You can turn off this display by using a SET command: SET VERIFY OFF.

READING MATERIAL 1

Introduction to tables

An Oracle database can contain multiple data structures. The table data structure stores the data in a relational database and it can be created at any time even while users are using the database. A table is composed of rows and columns. A table can represent a single entity (entity is an object whose information is stored in database) that you want to track within your system. Each entity or table has number of characteristics. The table must have a unique name through which it can be referred to after its creation .The characteristics of the table are called its attributes. These attributes can hold data. This type of a table could represent a list of the employees within your organization, or the orders placed for your company's products. A table has one attribute, which identifies each record uniquely; this attribute is called as Primary Key. Each value in the Primary Key attribute is unique and it cannot be NULL. Each record of the table is called as tuple.

Structure of table: Design tables before creating them

Usually, the application developer is responsible for designing the elements of an application, including the tables.

Consider the following guidelines when designing your tables:

- Use descriptive names for tables, columns, indexes, and clusters.
- Be consistent in abbreviations and in the use of singular and plural forms of table names and columns.
- Document the meaning of each table and its columns with the COMMENT command.
- Normalize each table.
- Select the appropriate data type for each column.
- Define columns that allow NULL values at the last, to conserve storage space.

Before creating a table, you should also determine whether to use integrity constraints. Integrity constraints can be defined on the columns of a table to enforce the business rules of your database automatically.

Table creation rules

There are certain standard rules for naming the tables and attributes/columns.

- Table names and column names must begin with a letter and can be 1-30 characters long.

- Names must contain only the characters A-Z, a-z, 0-9, _(underscore),$ and # .
- Names must not be an Oracle Server reserved word.
- Names must not duplicate the names of other objects owned by the same Oracle server user.
- Table name is not case sensitive.
- The table name must not be a SQL reserved word.

Create Table Statement

Tables are the fundamental storage objects of the database. Tables consist of rows and columns. A single table can have a maximum of 1000 columns.

Syntax

```
CREATE Table tablename
(
Column1 datatype [DEFAULT expr] [column constraint],
  column2 datatype[DEFAULT expr] [column constraint],…
        [table_constraint]}
);
```

Keywords used in the above syntax are:

Schema

It is the schema to contain the table. If it is omitted, Oracle creates the table in creator's own schema.

Table name

It is the name of the table to be created. Table name and Column name can be 1 to 30 characters long. First character must be alphabet, but name may include letters, numbers and underscores. The table name cannot be the name of another object owned by same user and cannot be a reserved word.

Column

Specifies the name of a column of the table. The number of columns in a table can range from 1 to 1000.

Datatype

It is the datatype of a column. Specifies the type and width of data to be stored in the column.

Default

It specifies a value to be assigned to the column if a subsequent INSERT statement omits a value for the column. The datatype of the expression must match the datatype of the column. A DEFAULT expression cannot contain references to other columns, the pseudo columns CURRVAL, NEXTVAL, LEVEL and ROWNUM or data constants that are not fully specified.

Example:

- Create a table student with the following fields:

Column Name	**Type**	**Size**	**Description**
Name	Varchar2	20	Name of the student
Class	Varchar2	15	Class of the student
Roll_no	Number	4	Roll number of the student
Address	Varchar2	30	Address of the student

```
SQL>Create table student
    (Name varchar2 (20),
    Class varchar2 (15),
    Roll_no number (4),
   Address varchar2 (30));
```

We can create tables using the SQL statement CREATE TABLE. When user SCOTT issues the following statement, it creates a table named STUDENT in its schema and stores it in the USERS tablespace.

Since creating a table is a DDL statement, an automatic commit (or save) takes place when this statement is executed.

ROLE OF CONSTRAINTS

Creating table from an existing Table

In addition to creating table as above, we can also create a new table from the existing table also .We apply the AS sub-query clause to both create the table and insert rows returned from the subquery. For example:

```
SQL> Create table emp1
     AS
     Select empno, ename, hiradate, sal from EMP where comm IS NULL;
```

This statement will create a new table emp1 which contains the rows returned by another table emp which satisfies the condition comm IS NULL.

Follow the following guidelines while creating a table from another table:

- If column specifications are given, the number of columns must equal the number of columns in the subquery select statement.
- If no column specifications are given, the columns of the new table are the same as the column names in the subquery.
- The select statement in the subquery will insert the records into the new table created.

Role of Constraints to achieve data integrity

To understand the role of constraints in database let us consider a case, when we appear for some interview, there are certain constraints for our qualification. Person who satisfies the given qualification constraints are eligible for interview, so these restrictions are called constraints which are to be enforced to select the correct candidate. Such limitations have to be enforced on the data to achieve the integrity (correctness).

The data, which does not, satisfies the conditions will not be stored, this ensures that the data stored in the database is valid and has gone through the integrity rules which prohibit the wrong data to be entered into the database.

Oracle allows us to apply constraints on single column or more than one column through the SQL syntax that will check data for integrity. While creating a table, constraints can be placed on the values that will be entered into the column(s). SQL will reject any value that violates the criteria that were defined.

Categories of Constraints

There are two categories of constraints, these are:

- Column constraints
- Table constraints

Column Constraints

When we define constraints along the definition of the column while creating or altering the table structure, they are called as column constraints. Column constraints are applied only to the individual columns. These constraints are applied to the current column. A column level constraint cannot be applied if the data constraint spans across multiple columns in a table.

For example: Empno of EMP table is a primary key, which is column level constraint.

Table Constraints

Table constraints are applied to more than one column of a table. These constraints are applied after defining all the table columns when creating or altering the structure of the table. A table level constraint can be applied if the data constraint spans across multiple columns in a table.

For example: in case of bank database the account number field of account holder table is not sufficient to uniquely identify the entities because two person can have joint account and hence repeating the value of account number once for each account holder,so in this case we have to apply primary key constraint on combination of account number field and name field to achieve the uniqueness. Now, in this case the primary key

constraint is applied on more than one column of the table, so this is the table level constraint.

Constraints can be added to the table at the time of creating a table with the create table command or later on with the help of ALTER TABLE command. All the details of constraints are stored in data dictionary. Each constraint is assigned a name by itself. Constraints are stored in the system tables by the Oracle engine .We can also give more user friendly names to the constraints so that they can be easily referenced, otherwise the name is automatically generated of the form SYS_Cn where n is unique number.

NOT NULL, UNIQUE and PRIMARY KEY CONSTRAINT

Types of Constraints

There are following types of constraints according to their nature.

NOT NULL constraint

This constraint prevents the column from accepting NULL values. In other words, in this case it is compulsory to supply some value to the constrained column during data entry. We cannot leave the column as empty. Columns without the NOT NULL constraint allow NULL values. This constraint can be applied only at column level. But one table can have more than one column level NOT NULL constraint on different columns.

Syntax:

```
Columnname datatype (size) [constraint constraintname] NOT NULL
```

Example:

- Create a table student with the fields name, roll_no, address and phone number.

```
SQL> CREATE table student
    (Name varchar2 (20) CONSTRAINT NN_NAME NOT NULL,
     Roll_no number (5) CONSTRAINT NN_ROLL NOT NULL,
     Address varchar2 (40),
     phone_no varchar2 (10));
```

Unique constraint

The unique key allows unique values to be entered into the column i.e. every value is a distinct value in that column. When we apply a unique constraint on a group of columns, then each value of the group should be unique, they must not be repeated. A table can have more than one unique key. This constraint can be applied both at the table level and the column level. The syntax is:

Column level syntax

```
Columnname datatype (size) [constraint constraintname] UNIQUE
```

Example:

- Create a table student with roll_no not null, unique name, unique address and unique phone number.

```
SQL> CREATE table student
  (Roll_no number (5) NOT NULL,
   name varchar2(20) constraint un_name unique,
  Address varchar2 (40) unique, phone_no varchar2 (10) unique);
```

Table level syntax of unique constraint

```
[constraint constraintname]  Unique (columnname [, columnname, …….])
```

Example

- Create a table student with roll_no not null and the combination of name, address and phone number must be unique.

```
SQL> CREATE table student
  (Roll_no number (5) NOT NULL,
   Name varchar2 (20),
  Address varchar2 (40),
   phone_no varchar2 (10),
   Constraint tb_un UNIQUE (name, address, phone_no));
```

Primary key constraint

A primary key is used to identify each row of the table uniquely. A primary key may be either consists of a single column or group of columns. It is a combination of both the unique key constraint as well as the not null constraint i.e. no column with the primary key constraint can be left blank and it must contain unique value in each row. We can declare the Primary key of a table with NOT NULL and UNIQUE constraint. SQL supports primary keys directly with the Primary Key constraint. When primary key is applied on a single field it is called as Simple Key and when applied on multiple columns it is called as Composite Primary Key. In a table there is only one primary key. The syntax of the Primary Key:

Column level syntax

```
Columnname datatype(size) [ constraint_name]  PRIMARY KEY
```

Example

- Create a table student with roll_no as the Primary Key, unique name, address cannot be NULL and phone number.

```
SQL> CREATE table student
   (Roll_no number (5) Primary key,
    Name varchar2 (20) Unique,
   Address varchar2 (40) Not Null,
    phone_no varchar2 (10));
```

Table level syntax of Primary Key constraint

```
PRIMARY KEY (columnname [, columnname , ……])
```

Example

- Create a table student with name and class as the composite primary key, address and phone_no are the other fields.

```
SQL> CREATE table student
   (Name varchar2 (20),
    Class varchar2 (15),
   Address varchar2 (40) Not Null,
    phone_no varchar2 (10)
    PRIMARY KEY (name, class));
```

CHECK CONSTRAINT

Check constraints allow Oracle to verify the validity of data being entered on a table against a set of constant values. These constants act as valid values. The Check constraint consists of the keyword CHECK followed by parenthesized conditions. Check constraints must be specified as a logical expression that evaluates either to TRUE or FALSE. If an SQL statement causes the condition to be false an error message will be displayed.

Syntax of the check constraint is:

Column level syntax

Columnname datatype (size) [constraint constraintname] CHECK (logical expression)

Example

- Create a table EMP1 with empid as primary key, phone number as unique attribute and the salary of all the employees must be greater than 5000, address and phone_no are the other fields.

```
SQL> CREATE TABLE emp1
(empid number (10),
 salary number (10, 3) CHECK (salary > 5000),
home_phone number (12),
CONSTRAINT pk_empid
PRIMARY KEY (empid),
CONSTRAINT uk_phone UNIQUE (home_phone));
```

If, for example, someone tries to create an employee row for the table defined earlier with a salary of Rs. 1000, Oracle will return an error message saying that the record data defined for the SALARY column has violated the check constraint for that column.

Example

- Create a table emp1 with empid as primary key, phone number as unique attribute and the department of the employees must be either general or accounts with name and class as the composite primary key, address and phone_no are the other fields.

```
SQL> CREATE TABLE emp1
(empid number (10),
 deptname varchar2 (20) CHECK deptname in ('general','accounts'),
home_phone number (12),
CONSTRAINT pk_empid
PRIMARY KEY (empid),
CONSTRAINT uk_phone UNIQUE (home_phone));
```

Another Example

```
CREATE TABLE emp(
code CHAR(4) PRIMARY KEY, name VARCHAR2(15) NOT NULL,
department CHAR(10) CHECK (department IN ('mkt','sales','Acct'))
age NUMBER CHECK (age between 18 and 55),
basic NUMBER);
```

Limitations of Check Constraints

Check constraints have a number of limitations, these are:

- A column level check constraint cannot refer to another column of any table.
- It cannot refer to special keywords that can have values in them, such as user, sysdate or rowid.

Violations of Check Constraint

The check constraint in the table definition earlier is valid, but the one in the following excerpt from a table definition is not:

```
CREATE TABLE address
(……...,
city VARCHAR2(80) check(city in (SELECT city FROM cities))
...);
```

-- It is invalid because here check constraint refers a column of different table through SELECT statement.

Table level syntax

Check (logical expression)

Example

```
SQL> CREATE TABLE emp1
(empid number (10),
 salary number (10, 3),
comm number (10, 3),
home_phone number (12),
CONSTRAINT pk_empid PRIMARY KEY (empid),
CONSTRAINT uk_phone UNIQUE (home_phone),
CHECK (salary > comm));
```

Example

```
SQL> CREATE TABLE emp1
(empid number (10),
 salary number (10, 3),Home_phone number (12),
CONSTRAINT pk_empid    PRIMARY KEY (empid),
CONSTRAINT uk_phone UNIQUE (home_phone),
CHECK (salary < 500000));
```

FOREIGN KEY CONSTRAINT

A foreign key is a kind of constraint which establishes a relationship among tables. A foreign key may be a single column or the combination of columns which derive their values based on the primary key or unique key values from another table. A foreign key constraint is also known as the referential integrity constraint, as its values correspond to the actual values of the primary key of other table.

A table in which there is a foreign key, it is called as the detail table or the child table and the table from which it refers the values of the primary key, it is called as the master table or the parent table. A foreign key must have the corresponding primary key or unique key value in the master table. Whenever you insert a value in the foreign key column, it checks the value from the primary key of the parent table. If the value is not present, it gives an error. The parent table can be referenced in the foreign key by using the References option.

Consider two tables (EMP, DEPT) given below.

Target Attribute

EMP

Empno	Ename	Job	Sal	Deptno
1	A	Clerk	4000	10
2	A	Clerk	4000	30
3	B	Mgr	8000	20
4	C	Peon	2000	40
5	D	Clerk	4000	10
6	E	Mgr	8000	50

DEPT

Deptno	Dname	Loc
10	A	Asr
20	B	Jal
30	C	Qadian
40	D	-

If we try to insert information of employee with deptno 50, then this is an invalid information, because there is no deptno 50 exists in the company(as shown in table DEPT).Then, this invalid information should be prevented from insertion, which would only be possible if deptno of EMP table refer the deptno of DEPT table. It means that only those values are permitted in deptno of EMP table, which appears in the deptno attribute of the DEPT table. Thus, we can say that

deptno of EMP table is the foreign key which refers the primary key deptno of DEPT table. Thus, we can insert the empno 6 with any deptno from 10,20,30 and 40.

Null may also be permitted in the deptno of EMP table. Here, deptno of DEPT table is Target attribute and DEPT is the target table.

Consider another example:

Student

Rno	Name	Class_Code
1	A	2
2	B	1
3	C	-

Class

Class_Code	Name
1	B.TECH
2	B.TECH
3	BBA

Here, college has three valid classes with class code 1,2 and 3. The class_code(foreign key) of student table refers class_code of class table.

The syntax of the foreign key at the column level is

```
Columnname datatype (size) references tablename [(columnname)]
```

Example

Create a table emp_detail in which the deptno field refers its value from the deptno field of the dept table.

```
SQL> CREATE table emp_detail
  (Empno number (4) primary key,
    Ename varchar2 (20) not null,
    Hiredate date not null,
    Deptno number (2) references dept (deptno) ,
    Salary number (6, 2));
```

So, in the emp_detail table deptno is a foreign key, which gets its values from the parent table i.e. the dept table.

The syntax of the foreign key at the table level is

```
Foreign key (columnname [,columnname……..]) references tablename (columnname [,columnname……. ])
```

Example

- Create a table emp_detail in which the deptno and dname together refers its value from deptno and dname field of dept table.

```
SQL> CREATE table emp_detail
      (Empno number (4) primary key,
       Ename varchar2 (20) not null,
       Hiredate date not null,
       Deptno number (2),
       Dname varchar2 (10),
       Salary number (6, 2)
       Foreign key (deptno,dname) references dept(deptno,dname)) ;
```

So in this table deptno and dname is the foreign key which references the corresponding fields of the dept table.

Important Note:

It is important to consider what will happen to child record if parent record is updated or deleted.

For example, let us consider the following database:

Emp (eno, ename, job, dno) Having eno as primary key and dno as forign key referencing dept (dno).

Dept (dno, dname) having dno as primary key.

Here, dept table is parent as shown below:

EMP					DEPT	
ENO	ENAME	JOB	DNO		DNO	DNAME
1	RAJ	CLERK	10		10	COMPUTER
2	RAM	PROF	20		20	CIVIL
3	RAHAT	PROF	10			
4	RISHAN	ASSOC PROF	20			

Now, it is important to consider what will happen, if user tries to delete record of deptno 20 from DEPT table. Then of course record of emp no 2 and 4 become invalid as they belong to this department.

There are four options to handle this situation.

- Prevent the dept being deleted until all its employees are re allotted to some other dept
- Automatic delete the employees who belong to that dept
- Set the dept column for the employees who belong to that dept to NULL
- Set the dept column of these employees to some default value which indicate they are currently not allotted to any dept

To achieve this, four options van be set in CREATE TABLE command

ON DEELTE RESTRICT

ON DELETE CASCADE

ON DELETE SET NULL

ON DELETE SET DEFAULT

The syntax to each of these cases has been given below:

```
SQL> CREATE TABLE DEPT(DNO NUMBER(2) PRIMARY KEY,
DNAME CHAR(20));
```

Case-I

```
SQL> CREATE TABLE EMP(ENO NUMBER(2) PRIMARY KEY,
ENAME CHAR(20), JOB CHAR(20)
DNO NUMBER(2) REFERENCES DEPT(DNO) ON DELETE RESTRICT;
```

It will prevent the dept being deleted until all its employees are re allotted to some other dept. This is the default case and will be applicable if did not give any option. It means the below command will also has the same effect.

```
SQL> CREATE TABLE EMP(ENO NUMBER(2) PRIMARY KEY,
ENAME CHAR(20), JOB CHAR(20)
DNO NUMBER(2) REFERENCES DEPT(DNO);
```

Case-II

```
SQL> CREATE TABLE EMP(ENO NUMBER(2) PRIMARY KEY,
ENAME CHAR(20), JOB CHAR(20)
DNO NUMBER(2) REFERENCES DEPT(DNO) ON DELETE CASCADE;
```

It will also delete the employees who belong to that dept.

Case-III

```
SQL> CREATE TABLE EMP(ENO NUMBER(2) PRIMARY KEY,
ENAME CHAR(20), JOB CHAR(20)
DNO NUMBER(2) REFERENCES DEPT(DNO) ON DELETE SET NULL;
```

It will set the dept column for the employees who belong to that dept to NULL.

Case-IV

```
SQL> CREATE TABLE EMP(ENO NUMBER(2) PRIMARY KEY,
ENAME CHAR(20), JOB CHAR(20)
DNO NUMBER(2) REFERENCES DEPT(DNO) ON DELETE SET DEFAULT;
```

It will set the dept column of these employees to some default value which indicate they are currently not allotted to any dept. This concept is not implemented in Oracle.

ALTERING TABLES

After you have created the tables, there may be time when you want to change the structure of the table. Either ou want to add a column or change the definition of the existing columns. We can do this with the help of ALTER TABLE statement. To alter a table, the table must be contained in your schema, or you must have either the ALTER object privilege for the table or the ALTER ANY TABLE system privilege. A table in an Oracle database can be altered for the following reasons:

- To add or drop one or more columns to or from the table.
- To add or modify an integrity constraint on a table.
- To modify an existing column's definition (datatype, length, default value, and NOT NULL integrity constraint).
- To enable or disable integrity constraints or triggers associated with the table.
- To drop integrity constraints associated with the table.

You can increase the length of an existing column. However, you cannot decrease it unless there are no rows in the table. Furthermore, if you are modifying a table to increase the length of a column of datatype CHAR, realize that this may be a time consuming operation and may require substantial additional storage, especially if the table contains many rows. This is because the CHAR value in each row must be blank-padded to satisfy the new column length.

Before altering a table, familiarize yourself with the consequences of doing so.

- **If a new column is added to a table, the column initially has NULL values.**
- **You can add a column with a NOT NULL constraint to a table only if the table does not contain any rows.**
- **If a view or PL/SQL program unit depends on a base table, the alteration of the base table may affect the dependent object.**

To change the definition of the table ALTER TABLE statement is used.

The syntax of the ALTER TABLE statement is

```
ALTER TABLE < table_name >
[ADD   < columnname > | <constraints >…….]
[MODIFY   <columnname>……]
[DROP <options   >];
```

To ADD new columns

If you want to add new columns to your table than use the ALTER TABLE statement with the ADD clause. You can add one or more than one columns at the same time. The database developer will need to understand how to implement changes on the database in an effective and simple manner. The developer can do one of two things when a request to add some columns to a table comes in. One is to add the columns, and the other is to re-create the entire table from scratch. Obviously, there is a great deal of value in knowing the right way to perform the first approach. Columns can be added and modified in the Oracle database with ease using the alter table statement and its many options for changing the number of columns in the database. The following code block is an example of using the alter table statement. If the developer or the DBA needs to add a column that will have a NOT NULL constraint on it, then several things need to happen. The column should first be created without the constraint, and then the column should have a value for all rows populated. After all column values are NOT NULL, the NOT NULL constraint can be applied to it. If the user tries to add a column with a NOT NULL constraint on it, the developer will encounter an error stating that the table must be empty.

Syntax

```
ALTER TABLE <table_name>
[ADD <column_name datatype (size) | <constraints> ,…….];
```

Here in the syntax:

table_name	Name of the table
Column_name	Name of the new column
datatype	Datatype of the column
size	Size of the column
constraints	Any type of constraint you want to put

The ADD option allows you to add PRIMARY KEY, CHECK, REFERENCES constraint to the table definition.

Example:

Adding columns

- Add the fields address and phone number in the emp table of width 50 and 10 charcater respectively.

```
SQL> ALTER table emp
  Add (address varchar2 (50),phone_no varchar2(10)) ;
```

Adding Primary Key

- Alter the dept table add the Primary Key constraint to deptno.

```
SQL> ALTER table dept add Primary Key (deptno);
```

- Alter the dept table add the Primary Key constraint with name PK_deptno to deptno.

```
SQL> ALTER table dept add constraint
  PK_deptno Primary Key (deptno);
```

Adding foreign key

- Alter the emp table, add foreign key constrain to deptno column so that refers to deptno of dept table.

```
SQL> ALTER TABLE emp add foreign key(deptno)REFERENCES dept(deptno);
```

OR

```
SQL> ALTER TABLE emp
    add constraint  FK_DEPTNO foreign key (deptno)
     REFERENCES dept(deptno) ;
```

TO MODIFY COLUMNS

Several conditions apply to modifying the datatypes of existing columns or to add columns to a table in the database. The general thumb rule is that increases are generally OK, while decreases are usually a little trickier. Some examples of increases that are generally acceptable are listed as follows:

- Increases to the size of a VARCHAR2 or CHAR column.
- Increases in size of a NUMBER column.
- Increasing the number of columns in the table.

The following list details the allowable operations that decrease various aspects of the database:

- Reducing the number of columns in a table (empty table only)
- Reducing the size of a NUMBER column (empty column for all rows only)
- Reducing the length of a VARCHAR2 or CHAR column (empty column for all rows only)
- Changing the datatype of a column (empty column for all rows only)

The Modify option changes the following of the existing column

- Datatype
- Column width
- Constraints i.e. DEFAULT, NOT NULL or NULL.

There are certain restrictions while making these modifications:

- The type and/or size of a column can be changed, if every row for the column is NULL.
- An existing column can be modified to NOT NULL only if it has a non-null value in every row.

The syntax of the MODIFY option is

```
Alter table <tablename >
[Modify < columnname >…….];
```

Examples

- Modify the job column of the emp table by increasing its width to varchar2 (40).

```
SQL> ALTER TABLE emp
MODIFY job varchar2 (40);
```

- Modify the job column of the emp table by increasing its width to varchar2 (35) and applying the constraint of NOT NULL.

```
SQL> ALTER TABLE emp
 MODIFY job varchar2 (35) NOT NULL;
```

The DROP option

This option removes the constraints from a table. When a constraint is dropped, any associated index with that constraint (if there is one) is also dropped.

The syntax is

```
Alter table < tablename >
 [DROP    <constraints > …….] ;
```

Example

- Remove the primary key constraint from the emp table.

```
SQL> ALTER TABLE emp
 DROP primary key;
```

- Remove the constraint on the deptno field of the dept table.

```
SQL> ALTER TABLE dept DROP constraint pk_deptno ;
```

Drop column option

Alter table command can also be used to drop the existing columns of the tables, but this option is available only in Oracle 8i or versions after that.

Syntax

```
ALTER TABLE <table_name> DROP COLUMN <column_name>;
```

Example

- Drop the column ename of emp table.

```
SQL> ALTER TABLE emp DROP COLUMN ename;
```

Removing Tables

In order to remove a table from the database, the drop table command must be executed.

The syntax is

```
DROP table <tablename >;
```

Example

```
SQL> DROP TABLE emp;
```

However, dropping tables may not always be that easy. Recall from the earlier lesson that when you disable constraints like primary keys that have foreign-key constraints in other tables depending on their existence, you may have some errors. The same thing happens when you try to drop the table that has a primary key referenced by enabled foreign keys in another table. If you try to drop a table that has other tables' foreign keys referring to it, the following error will ensue:

ORA-02266: unique/primary keys in table referenced by enabled foreign keys

When there are foreign-key constraints on other tables that reference the table to be dropped, then you can use on delete cascade option. The constraints in other tables that refer to the table being dropped are also dropped with cascade option. There are usually some associated objects that exist in a database along with the table. These objects may include the index that is created by the primary key or the unique constraint that is associated with columns in the table. If the table is dropped, Oracle automatically drops any index associated with the table as well.

```
SQL> DROP TABLE dept
CASCADE CONSTRAINTS;
```

Alternately, you can disable or drop the foreign key in the other table first and then issue the drop table statement without the cascade constraints option. However, with this method you run the risk that many other tables having foreign keys that relate back to the primary key in the table you want to drop will each error out, one at a time, until you disable or drop every foreign-key constraint referring to the table. If there are several, your drop table activity may get extremely frustrating.

Available tables as Data Dictionary

User tables are tables created by the user such as student. There is another collection of tables, which are owned by the SYS user in the Oracle database known as data dictionary. These tables are maintained and created by the Oracle Server and contains information about the database. Information stored in the data dictionary include name of the Oracle Server users, privileges granted to users, database objects names, table constraints and other information. You can query the following data dictionary tables to view various database objects owned by you .The data dictionary tables frequently used are:

- USER_TABLES
- USER_OBJECTS
- USER _CATALOG

Flash Back

The table data structure stores the data and it is composed of rows and columns. A table can also represent a relation between two entities. First step before creating tables is to design them so that it gives a meaningful picture. One must follow rules to create a table as the name of the should not be a SQL reserve word , not more than 30 characters,

must contain only the characters A-Z, a-z, 0-9, _, $, # . the name of the table is case insensitive.

The table can be created with the create statement and the table can be created from the existing table also which may or may not contain data. We can impose number of constraints on the table as a whole as well as on the columns also. these constraints provides data integrity and are divided into two categories:

- Column constraints(applied on one column)
- Table constraints(applied on more than one column)

The different types of constraints are;

- Not null constraint
- Unique constraint
- Primary key constraint
- Default constraint
- Foreign key constraint
- Check constraint

The user can see the names of the constraints in the user_constraints table. Finally, if we want to change the structure of the table we have the ALTER table command. With the alter statement we can add column or constraints, modify the datatypes or to add more columns, and drop the column or table itself.

WEEK 3: READING MATERIAL 1

JOINING OF TABLES

Joining of tables for multiple table Queries

In Relational Database Management Systems, data stored in different tables is related. We can use the power of SQL to relate the information and query data. A SELECT statement has a mandatory SELECT clause and FROM clause. The SELECT clause can have a list of columns, expressions, functions, and so on. The FROM clause tells you which table(s) to look in for the required information. So far, we have seen only one table in the FROM clause, now we will learn how to retrieve data from more than one table. In order to query data from more than one table, we need to identify a common column that relates the two tables. In the WHERE clause, we define the relationship between the tables listed in the FROM clause using comparison operators.

When data from more than one table in the database is required, a join condition is used. Rows in one table can be joined to rows in another table according to common values existing in corresponding columns, that is, usually primary and foreign key columns. A simple join condition in the WHERE clause. Oracle performs a join whenever multiple tables appear in the query's FROM clause. The query's SELECT clause can have the columns or expressions from any or all of these tables.

Syntax of Join

```
SELECT  table1.column, table2.column FROM table1, table2
WHERE table1.column1=table2.column2;
```

In the syntax:

table1.column, table2.column	denotes the table and column from which data is retrieved
table1.column1= table2.column2	is the condition that joins (or relates) the tables together

Points to Note:

- Write the join condition in the WHERE clause.
- Prefix the column name with the table name when the same column name appears in more than one table.

  Here, all the examples are based on the EMP and DEPT tables, whose data shown below:

SQL>*SELECT * FROM dept;*

OUTPUT:

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

SQL>*SELECT * FROM emp;*

OUTPUT:

EMPNO	ENAME	SALARY	COMM	DEPTNO
7566	JONES	2975		20
7654	MARTIN	1250	1400	30
7698	K_BLAKE	2850		30
7788	SCOTT	3000		20
7839	A_EDWARD	5000	50000	10
7844	TURNER	1500	0	30
7845	FORD	3000		20

How would we list the department name location for each employee, along with his or her salary? The department name and location are in the DEPT table; the employee name and salary are in the EMP table. So, to list the information together in one query, we need to do a join. The DEPTNO column is common to both tables; use this column to relate the rows.

To get the required information following query is used:

SQL>*SELECT dname, loc, ename, sal FROM dept, emp*

WHERE dept.deptno = emp.deptno;

OUTPUT:

DNAME	LOC	ENAME	SALARY
RESEARCH	DALLAS	JONES	2975
SALES	CHICAGO	MARTIN	1250
SALES	CHICAGO	K_BLAKE	2850
RESEARCH	DALLAS	SCOTT	3000
ACCOUNTING	NEW YORK	A_EDWARD	5000
SALES	CHICAGO	TURNER	1500
RESEARCH	DALLAS	FORD	3000

7 rows selected.

Here data is selected from two tables: EMP and DEPT. The department number (DEPTNO) is the column on which join is established. Notice that in the WHERE clause, the column names are qualified by the table name; this is required to avoid ambiguity, because the column names are the same in both tables. If the column names are different in each table, you need not qualify the column names. Just as we can provide column alias names, we can alias table names, also. Aliases improve the readability of the code, and they can be short names that are easy to type and use as references. The table alias name is given next to the table name.

The following example uses alias names d and e for DEPT and EMP tables and uses them to qualify the column names:

SQL> *SELECT d.name, d.loc, e.ename, e.sal*

FROM dept d, emp e

WHERE d.deptno = e.deptno

ORDER BY d.dname;

Note: Once table alias names are defined; you cannot use the table name to qualify a column. You should use the alias name to qualify the column.

To execute a join of three or more tables, Oracle takes these steps:

- Oracle joins two of the based tables on the join conditions, comparing their columns

- Oracle joins the result to another table, based on join conditions.
- Oracle continues this process until all tables are joined into the result.

The Join query also contain in the WHERE clause to restrict rows based on column in one table. Here's example:

SQL> *SELECT d.dname, d.loc, e.ename, e.sal*

FROM dept d, emp e

WHERE e.deptno = d.deptno and comm IS NOT NULL

OUTPUT:

DNAME	**LOC**	**ENAME**	**SAL**
SALES	CHICAGO	ALLEN	1600
SALES	CHICAGO	WARD	1250
SALES	CHICAGO	MARTIN	1250
SALES	CHICAGO	TURNER	1500

WEEK 3: READING MATERIAL 2

Types of Joins

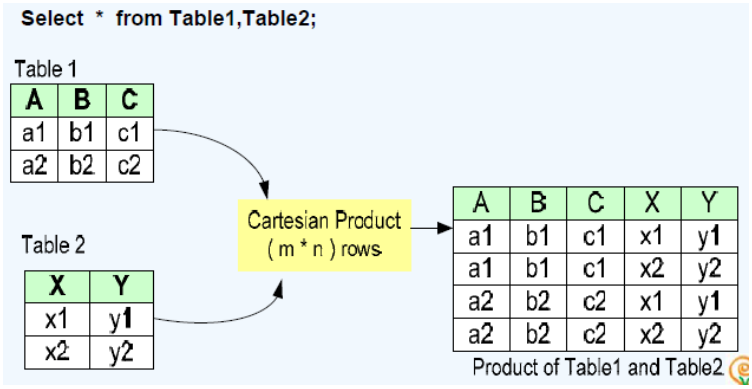
Following are the type of join.

- Cartesian Product
- Inner join
  - Equi join
  - Non-equi join
- Outer join
  - Left-outer join
  - Right-outer join
- Self join

Cartesian product

A Cartesian join occurs when data is selected from two or more tables and there is no common relation specified in the WHERE clause. If we do not specify a join condition for the tables listed in the FROM clause, Oracle joins each row from the first table to every row in the second table. In Cartesian join each row from the first table is combined with all rows from the second table.

- The working of Cartesian product is illustrated below with example database.



If the first table has three rows and the second table has four rows, the result will have 12 rows. Suppose we add another table with two rows without specifying a join condition; the result will have 24 rows. We should avoid Cartesian joins; for the most part, they happen when there are many tables in the FROM clause and developers forget to include the join condition.

Example:

SQL>*Select ename,dname FROM emp, dept;*

The above query displays employee name and department name from EMP and DEPT tables. Because no WHERE clause has been specified , all rows(14 rows) from EMP table are joined with all (4 rows) in the DEPT table, there by generating 56 rows in the output as shown below:

EMP (14 rows)

EMPNO	ENAME	DEPTNO
7839	KING	10
7698	BLAKE	30
…….	…….	…..
7934	MILLER	10

DEPT (4 rows)

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

ENAME	DNAME
KING	ACCOUNTING
BLAKE	ACCOUNTING
……	…….
KING	RESEARCH
BLAKE	RESEARCH
……	……..

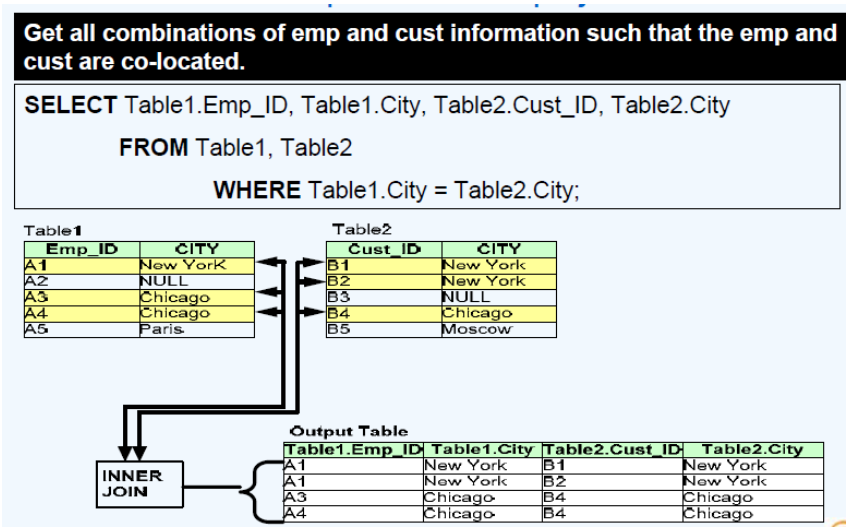
Cartesian product 14*4=56 rows selected

Inner Join

An inner join between two (or more) tables is the Cartesian product that satisfies the join condition in the WHERE clause. It is of two types, depending upon the WHERE consition.

Equality Joins

If the WHERE clause in query of relating two tables used an equality operator (=), it is an equality join, also known as an inner join or an equijoin. As illustrated below.



The query discussed earlier to list the department name location for each employee, along with his or her salary is an example of equality join.

To get the required information following query is used:

SQL>*SELECT dname, loc, ename, sal FROM dept, emp*

WHERE dept.deptno = emp.deptno;

Non-Equality Join

If any other operator instead of equality operator (=) is used to join the tables in the query, it is non-equality join.

We have already seen examples of equality joins; let's consider an example of non-equality join.

Example:

EMP

EMPNO	ENAME	SAL
7839	KING	5000
7698	BLAKE	2850
7782	CLARK	2450
7566	JONES	2975
7654	MARTIN	1250
7499	ALLEN	1600
7844	TURNER	1500
7900	JAMES	950

Salary in the EMP table is between low salary and high salary in the SALGRADE table

SAL GRADE

GRADE	LOSAL	HISAL
1	700	1200
2	1201	1400
3	1401	2000
4	2001	3000
5	3001	9999

The relationship between the EMP table and the SALGRADE table is a non-equijoin, meaning that no column in the EMP table corresponds directly to a column in the SALGRADE table. The relationship between the two tables is that the SAL column in the EMP table is between the LOSAL and HISAL column of the SALGRADE table. The relationship is obtained using an operator other than equal (=).

SQL> *SELECT e.ename, e.sal, s.grade from emp e, salgrade s*

WHERE e.sal BETWEEN s.losal and S.hisal;

OUTPUT:

ENAME	SAL	GRADE
JAMES	950	1
SMITH	800	1
ADAMS	1100	1
…….	…….	……

14 rows selected.

The above example creates a non-equijoin to evaluate an employee's grade. The salary must be between any pair of the low and high salary ranges.

WEEK 3: READING MATERIAL 3

Outer Joins

It retrieve all rows that match the WHERE clause and also those that have a NULL value in the column used for join. The concept and need of outer join has been explained below.

Sometimes, we might want to see the data from one table, even if there is no corresponding row in the joining table. Oracle provides the outer join mechanism for this.

Such rows can be forcefully selected by using the outer join symbol (+). The corresponding columns for that row will have Nulls. For example, to write a query that performs an outer join of tables A and B and returns all rows from A, apply the outer-join operator (+) to all columns of B in the join condition. For all rows in A that have no matching rows in B, the query returns NULL values for the columns in B.

Example:

In the emp table, no record of the employee belongs to the department 40. Therefore, in case of equi join, the row of department 40 from the dept table will not be displayed. In order to include that row in the output Outer join is used.

- Display the list of employees working in each department. Display the department information even if no employee belongs to that department.

SQL>*SELECT empno, ename, emp.deptno, dname, loc FROM emp, dept*

WHERE emp.deptno (+) = dept.deptno;

OUTPUT:

EMPNO.	ENAME	EMP.DEPTNO	DNAME
7369	SMITH	20	RESEARCH
7499	ALLEN	30	SALES
7521	WARD	30	SALES
7566	JONES	20	RESEARCH
7654	MARTIN	30	SALES
7698	BLAKE	30	SALES
7782	CLARK	10	ACCOUNTING
7788	SCOTT	20	RESEARCH
7839	KING	10	ACCOUNTING

7844	TURNER	30	SALES
7876	ADAMS	20	RESEARCH
7900	JAMES	30	SALES
7902	FORD	20	RESEARCH
7934	MILLER	10	ACCOUNTING
7945	ALLEN	20	ACCOUNTING
7526	MARTIN	20	RESEARCH
7985	SCOTT	30	SALES
		40	OPERATIONS

If the symbol (+) is placed on the other side of the equation then all the employee details with no corresponding department name and location, will be displayed with NULL values in DNAME and LOC column.

Outer Join is of two types, i.e., left outer join and right outer join.

Left/Right-Outer joins

Left outer joins include all records from the first (left) of two tables, A = B (+), while right outer joins include all records from the second (right) of two tables, A (+) = B.

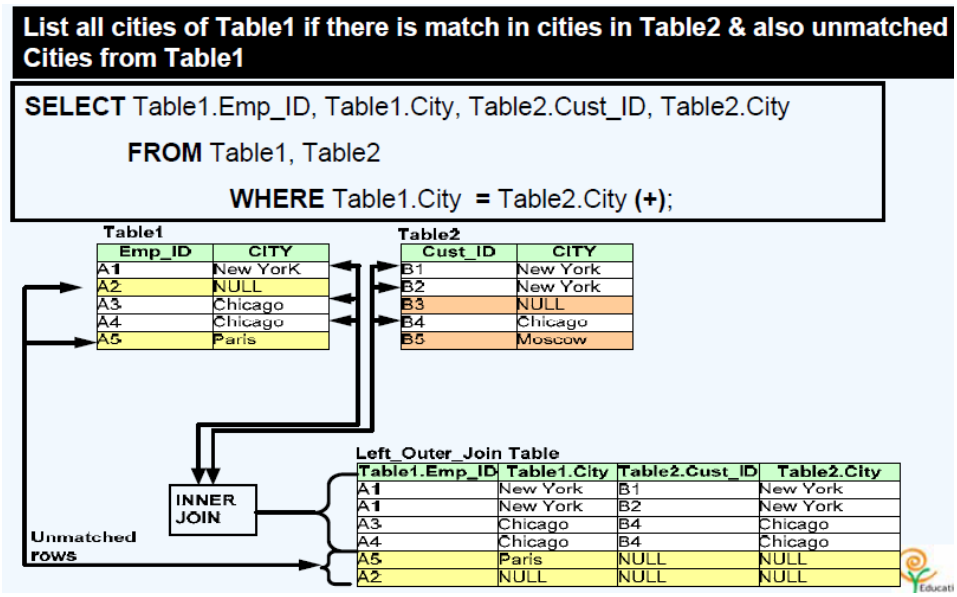
Example:

Can you answer, the earlier example of outer join is left outer join or right outer join?

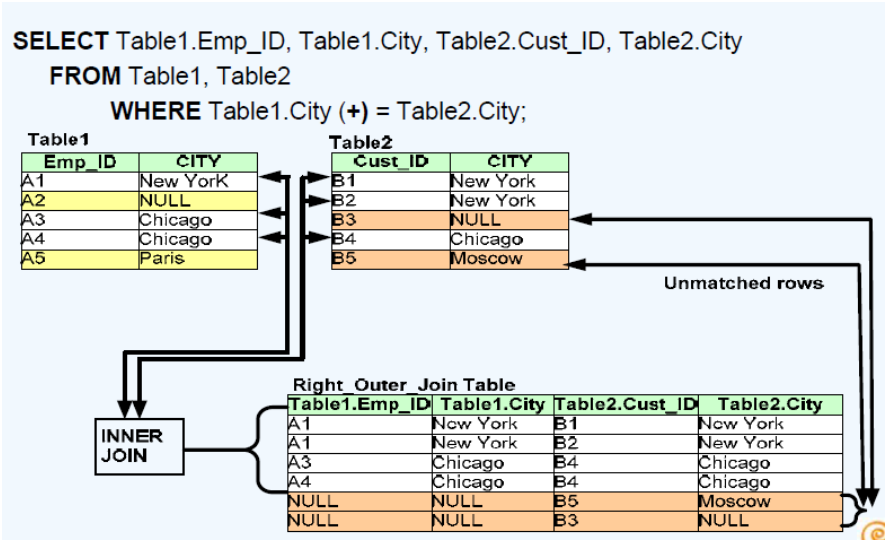
SQL>*SELECT empno, ename, emp.deptno, dname, loc FROM emp, dept*
WHERE emp.deptno (+) = dept.deptno;

Yes, it is right outer join, as full right table appears in the output by having NULL corresponding to missing values of EMP table. Here, right table appears fully and + appears on left side, so it is right outer join. In simple words, in right outer join plus appears on left side, while in left outer join plus appears on right side.

A case of left outer join has been illustrated below.



A case of right outer join on same database has been illustrated below.



Example of Left Outer Join

• List all customer details and loan details if they have availed loans.

SQL>*Select Customer_details.Cust_id,Cust_Last_name,Loan_no,Amount_in_dollars*

from Customer_details,Customer_loan

where Customer_details.Cust_id = Customer_loan.Cust_id (+);

Rules to Place (+) operator:

- The outer join symbol (+) cannot be on both sides.
- We cannot “outer join” the same table to more than one other table in a single SELECT statement.
- A condition involving an outer join may not use the IN operator or be linked to another condition by the OR operator.

WEEK 3: READING MATERIAL 4

Self-Join

To join a table to itself means that each row of the table is combined with itself and with every other row of the table. The self-join can be viewed as a join of two copies of the same table. The table is not actually copied, but SQL performs the command as though it were.

The syntax of the command for joining a table to itself is almost the same as that for joining two different tables. To distinguish the column names from one another, aliases for the actual the table name are used, since both the tables have the same name. Table name aliases are defined in the FROM clause of the query. To define the alias, one space is left after the table name and the alias.

Example:

EMP TABLE

EMPNO	ENAME	MGR
7839	KING	
7566	JONES	7839
7876	ADAMS	7788
7934	MILLER	7782
	……..	……

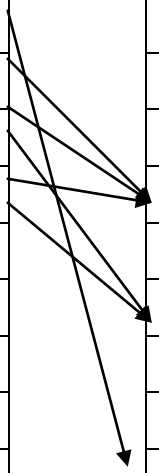
Consider the emp table shown above. Primary key of the emp table is empno. Details of each employee's manager is just another row in the EMP table whose EMPNO is stored in MGR column of some other row. So every employee except manager has a Manager. Therefore MGR is a foreign key that references empno. To list out the names of the manager with the employee record one will have to join EMP itself.

```
SQL>SELECT WORKER. Ename "Ename", MANAGER.ename "Manager"
    FROM emp WORKER, emp MANAGER
    WHERE WORKER.mgr=MANAGER.empno;
```

Where WORKER and MANAGER are two aliases for the EMP table and acts as a virtual tables.

WORKER

EMPNO	ENAME	MGR
7369	SMITH	7902
7499	ALLEN	7698
7521	WARD	7698
7566	JONES	7839
7654	MARTIN	7698
7698	BLAKE	7839
7782	CLARK	7839
7788	SCOTT	7566
7839	KING	
7844	TURNER	7698
7876	ADAMS	7788
7900	JAMES	7698
7902	FORD	7566
7934	MILLER	7782



MANAGER

EMPNO	ENAME	MGR
7369	SMITH	7902
7499	ALLEN	7698
7521	WARD	7698
7566	JONES	7839
7654	MARTIN	7698
7698	BLAKE	7839
7782	CLARK	7839
7788	SCOTT	7566
7839	KING	
7844	TURNER	7698
7876	ADAMS	7788
7900	JAMES	7698
7902	FORD	7566
7934	MILLER	7782

OUTPUT:

Ename	Manager
SMITH	FORD
ALLEN	BLAKE
WARD	BLAKE
JONES	KING
MARTIN	BLAKE
BLAKE	KING
CLARK	KING
SCOTT	JONES
TURNER	BLAKE
ADAMS	SCOTT

JAMES	BLAKE
FORD	JONES
MILLER	CLARK

13 rows selected.

- List all employees who joined the company before their manager.

SQL>*Select e.ename, e.hiredate, m.ename manager, m.hiredate FROM emp e, emp m*

WHERE e.mgr= m.empno

and e.hiredate<m.hiredate;

OUTPUT:

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

6 rows selected.

Note: If we wish to include those employees name who has no corresponding manager also in above list. Then it becomes the case of self join and outer join. In this scenario, we wish to list all the employees whether it has manager or not. So, worker table, i.e., left table appears full and (+) will appear on manager side so it becomes the case of right outer join. And corresponding query has been shown below.

SQL>*SELECT WORKER. Ename "Ename", MANAGER.ename "Manager"*

FROM emp WORKER, emp MANAGER

WHERE WORKER.mgr=MANAGER.empno(+);

OUTPUT:

Ename	Manager
SMITH	FORD
ALLEN	BLAKE
WARD	BLAKE
JONES	KING
MARTIN	BLAKE
BLAKE	KING
CLARK	KING
SCOTT	JONES
TURNER	BLAKE
ADAMS	SCOTT
JAMES	BLAKE
FORD	JONES
MILLER	CLARK
KING	

14 rows selected.

It shows that KING appears in the list and he has no corresponding manager or in simple words he is at top of hierarchy.

- List all employees who joined the company before their manager.

SQL>*Select e.ename, e.hiredate, m.ename manager, m.hiredate FROM emp e, emp m*
WHERE e.mgr= m.empno
and e.hiredate<m.hiredate;

GROUPING DATA WITH GROUP BY

GROUP BY clause is used to group or categorize the data. In other words it divide rows in a table into smaller groups. We can then use the group functions to return summary information for each group.

If no GROUP BY clause is specified, then the default grouping is the entire result set. When the query executes and the data is fetched, it is grouped based on the GROUP BY clause and the group function is applied.

Syntax of GROUP BY

```
SELECT column,group_function(column) FROM table
[WHERE condition]
[GROUP BY   group_by_expression]
[ORDER BY column];
```

Here, group_by_expression specifies columns whose values determine the basis for grouping rows.

For example, If we have to find the total salary of each department manually, first we group the records on the basis of department number and then we apply the sum function on salary of each group to obtain the required result.

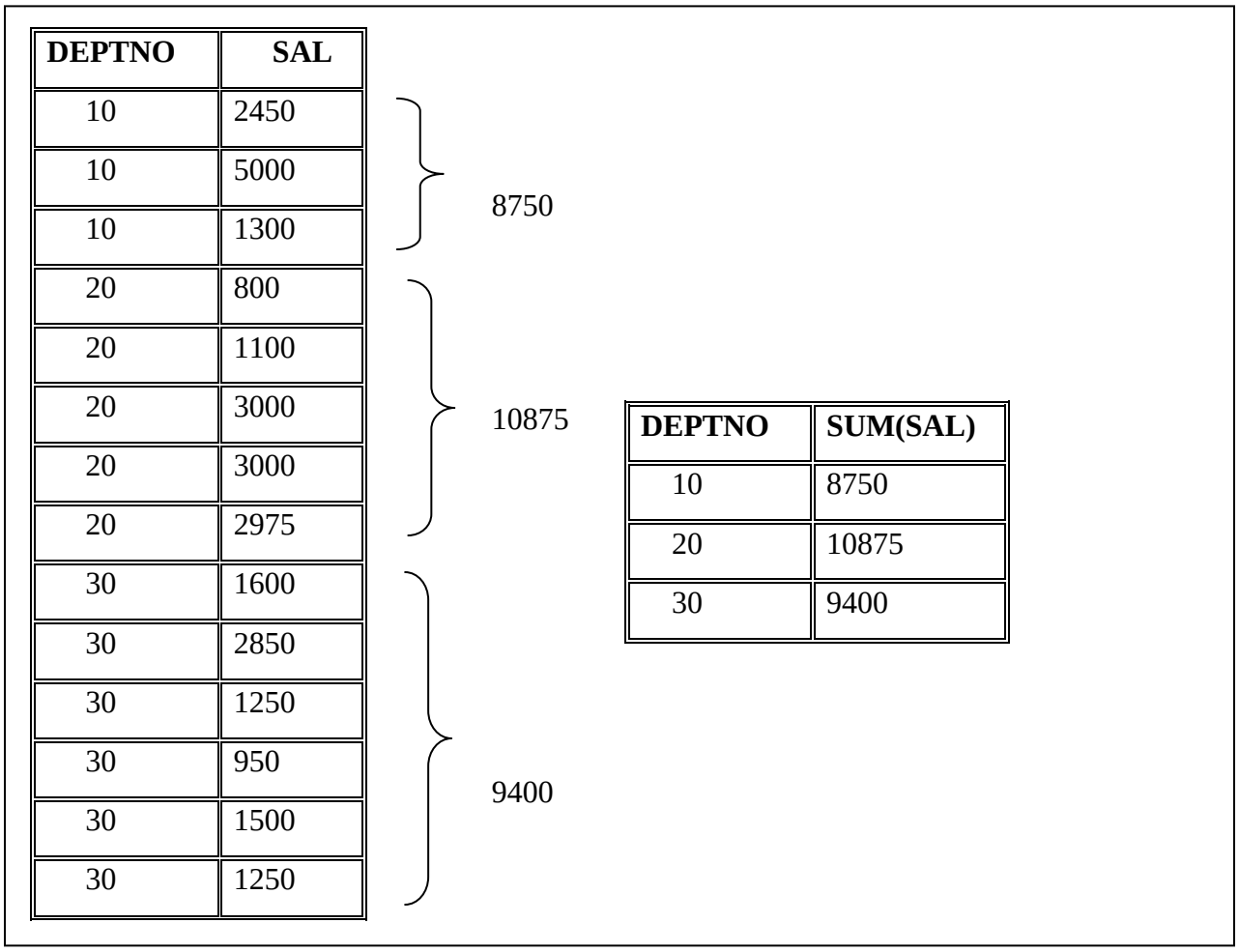
Similarly in SQL we apply the GROUP BY clause on deptno and then calculate the total salary for each group by Sum(sal) function as shown below:

SQL>*SELECT deptno, Sum(sal) FROM emp GROUP BY deptno;*

OUTPUT:

DEPTNO	SUM(SAL)
10	2916.6667
20	2175
30	1566.6667

The below figure shows the grouping and execution of query:



DEPTNO	SAL
10	2450
10	5000
10	1300
20	800
20	1100
20	3000
20	3000
20	2975
30	1600
30	2850
30	1250
30	950
30	1500
30	1250

DEPTNO	SUM(SAL)
10	8750
20	10875
30	9400

Here is how this SELECT statement, containing a GROUP BY clause, is evaluated:

- The SELECT clause specifies the columns to be retrieved i.e Department number column in the EMP table, the sum of all the salaries in the group you specified in the GROUP By clause
- The FROM clause specifies the tables that the database must access i.e EMP table.
- The WHERE clause specifies the rows to be retrieved. Since there is no WHERE clause, by default all rows are retrieved.

The GROUP BY clause specifies how the rows should be grouped. Department number groups the rows, so the AVG function that is being applied to the salary column will calculate the average salary for each department.

- List the average salary of each job in the emp table.

SQL>*SELECT JOB,AVG(SAL) FROM EMP GROUP BY JOB;*

OUTPUT:

JOB	AVG(SAL)
ANALYST	3166.5
CLERK	962.5
MANAGER	2758.3333
PRESIDENT	5000
SALESMAN	1400

- List the maximum salary for each dept.

SQL>*SELECT DEPTNO,MAX(SAL) FROM EMP GROUP BY DEPTNO;*

OUTPUT:

DEPTNO	MAX(SAL)
10	5000
20	3333
30	2850

WEEK 4: READING MATERIAL 3

Functions in SQL*PLUS

Functions are programs that take zero or more arguments and return a single value. Oracle has built a number of functions into SQL, and these functions can be called from SQL or PL/SQL statements (detail in next chapters).

Need of Functions

Functions can be used for the following purposes:

- To Perform Data calculations.
- To make modification of data.
- To manipulate data for desired output.
- To converting data values from one type to another.

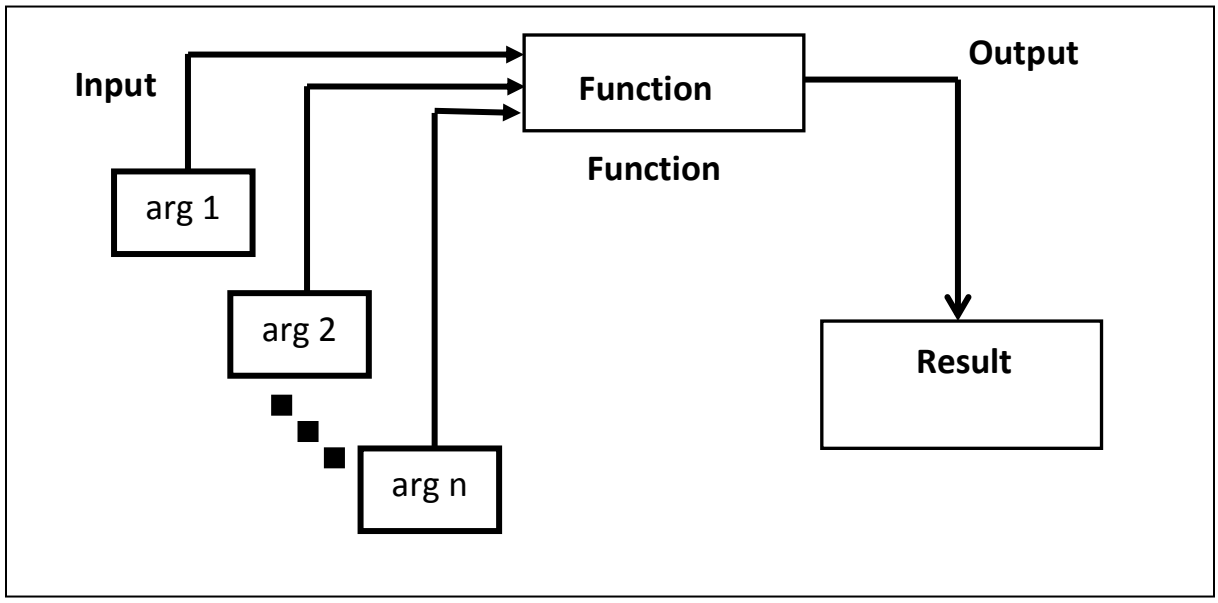


Figure 5.1: Basic functionality of function

Classes of Functions

There are two significant classes of functions as shown in figure 5.2.

- Single-row functions
- Group functions (also known as aggregate functions)

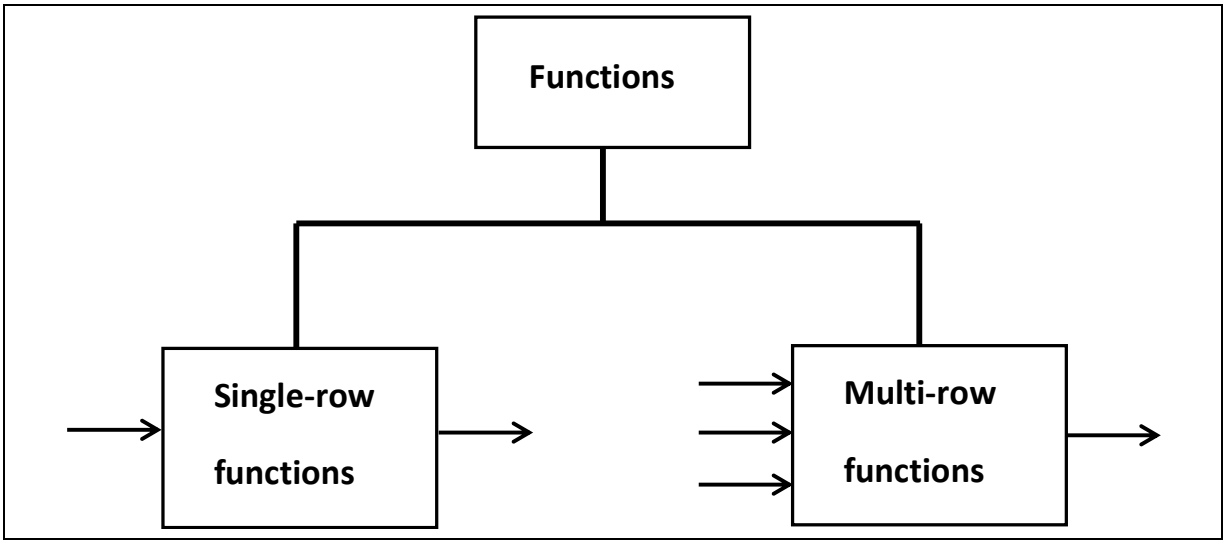


Figure 5.2: Classes of functions

Single-row functions know how many arguments they will have to process before data is fetched from the tables. Group functions don't know how many arguments they will have to process until all the data is extracted and grouped into categories.

Single Row Functions

Single Row Functions act on each row returned by the query. It result one result per row**.**

Classification of Single Row Functions

Single Row Functions can be classified into the following categories as shown in figure 5.3.

(i) Character

(ii) Number

(iii) Date

(iv) Conversion

(v) General

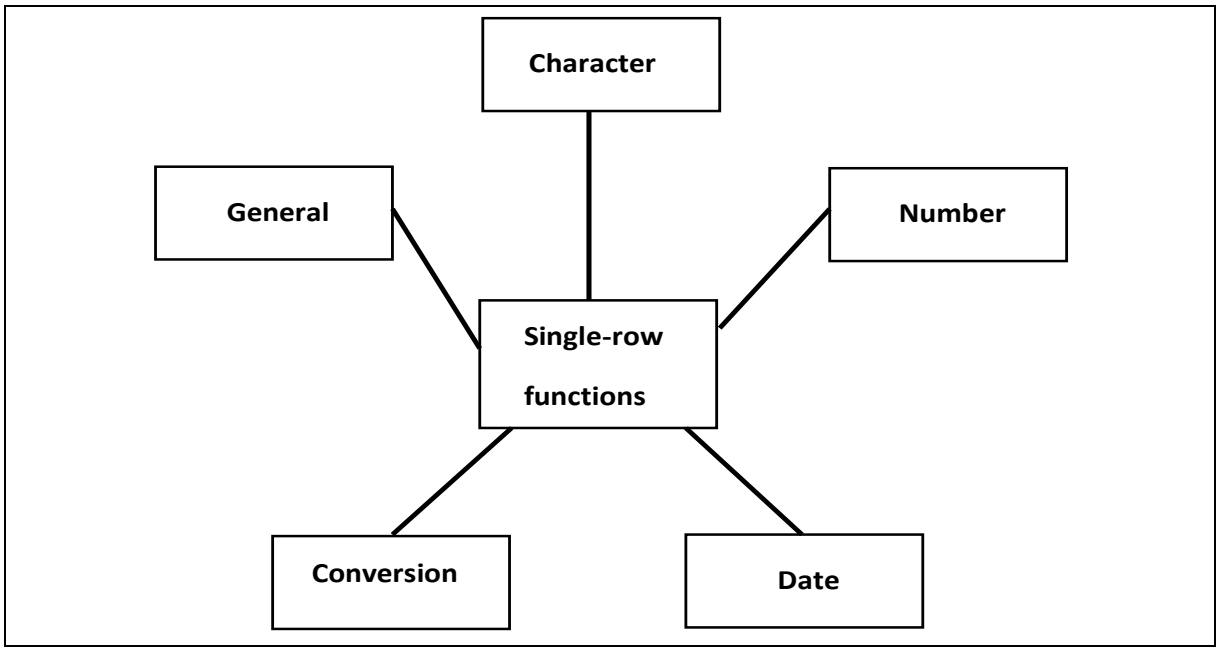


Figure 5.3: Classification of single row functions

Single-Row Character Functions

Single-row character functions operate on character data. Most have one or more character arguments, and most return character values. The classification of single row character is shown in figure 5.4.

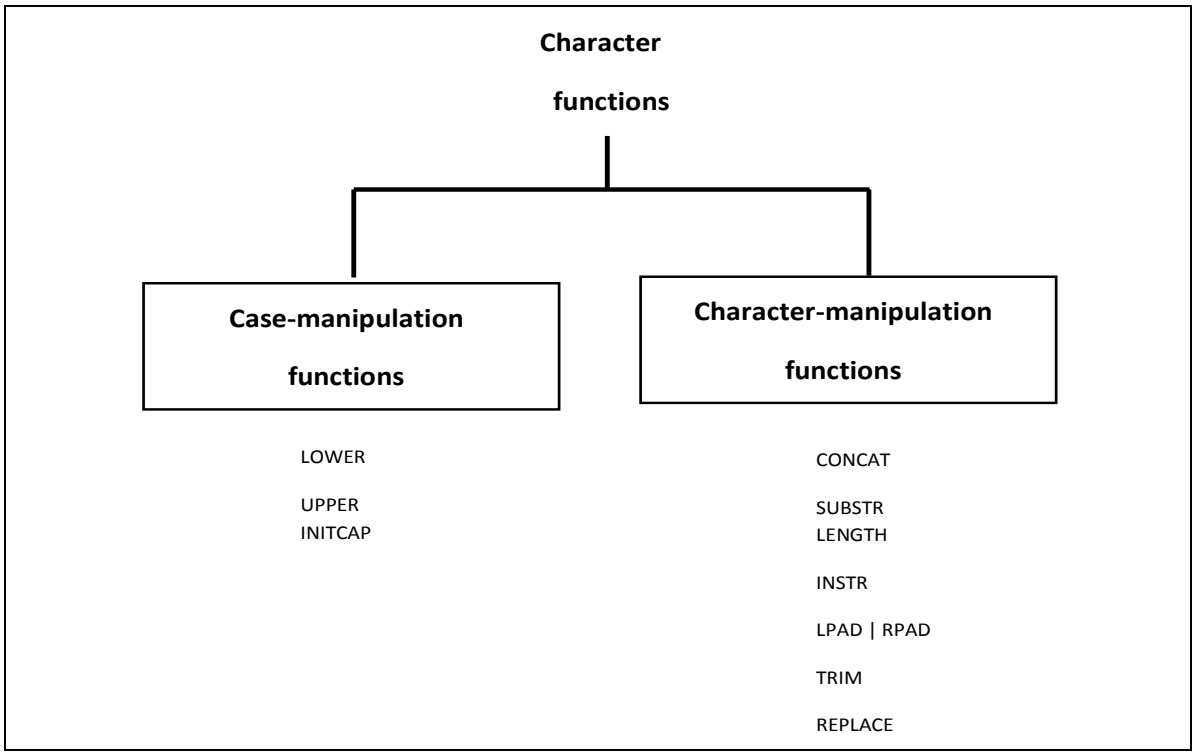


Figure 5.4: Classification of character fonctions

CONCAT(<*c1*>, <*c2*>)

Where *c1* and *c2* are character strings. This function returns *c2* appended to *c1*. If *c1* is NULL, then *c2* is returned. If *c2* is NULL, then *c1* is returned. If both *c1* and *c2* are NULL, then NULL is returned. CONCAT returns the same results as using the concatenation operator: *c1*||*c2*.

SQL>*SELECT CONCAT ('dav ','college') CollgeName FROM dual ;*

OUTPUT:

COLLEGENAME

---

dav college

ASCII(<*c1*>)

Where *c1* is a character string. This function returns the ASCII decimal equivalent of the first character in *c1*. See also CHR() for the inverse operation.

SQL>*SELECT ASCII ('A') Big_A, ASCII('z') Little_Z FROM dual;*

OUTPUT:

BIG_A	LITTLE_Z
65	122

CHR(x)

This function gives the result as a character corresponding to the value x in the character set.

SQL>*SELECT CHR(97) first , chr(99) second from dual ;*

OUTPUT:

FIRST	SECOND
a	c

INITCAP (<*c1*>)

Where *c1* is a character string. This function returns *c1* with the first character of each word in uppercase and all others in lowercase.

SQL>*SELECT INITCAP ('amit', 'raju', 'rajesh') name FROM dual;*

OUTPUT:

NAME

Amit, Raju, Rajesh

LENGTH (<*c*>)

Where *c* is a character string. This function returns the numeric length in characters of *c*. If *c* is NULL, a NULL is returned.

SQL> *SELECT LENGTH ('dav college') name FROM dual;*

OUTPUT:

NAME

11

LOWER (<*c*>)

Where *c* is a character string. This function returns the character string *c* with all characters in lowercase. It frequently appears in WHERE clauses. See also UPPER.

SQL>*select lower(dname) from dept ;*

OUTPUT:

lower(dname)
accounting
Research
Sales
operations

LPAD(<*c1*>, <*i*> [,<*c2*>])

Where *c1* and *c2* are character strings and *i* is an integer. This function returns the character string *c1* expanded in length to *i* characters using *c2* to fill in space as needed on the left-hand side of *c1*. If *c1* is over *i* characters, it is truncated to *i* characters. *c2* defaults to a single space. See also RPAD.

Example:

SQL> *select lpad(dname,12,' '),lpad(dname,12,'*') from dept ;*

OUTPUT:

lpad(dname,12,' ')	lpad(dname,12,'* ')
ACCOUNTING	**ACCOUNTING
RESEARCH	****RESEARCH
SALES	*******SALES
OPERATIONS	**OPERATIONS

LTRIM (<*c1*>, <*c2*>)

Where *c1* and *c2* are character strings. This function returns *c1* without any leading characters that appear in *c2*. If no *c2* characters are leading characters in *c1*, then *c1* is returned unchanged. *c2* defaults to a single space. See also RTRIM.

SQL> *SELECT LTRIM ('dav','d') FROM dual;*

OUTPUT:

```
LT
--
av
```

RPAD (<*c1*>, <*i*>[, <*c2*>])

Where *c1* and *c2* are character strings and *i* is an integer. This function returns the character string *c1* expanded in length to *i* characters using *c2* to fill in space as needed on the right-hand side of *c1*. If *c1* is over *i* characters, it is truncated to *i* characters. *c2* defaults to a single space. See also LPAD.

SQL> *SELECT RPAD (table_name, 38, '.'), num_rows FROM user_tables;*

OUTPUT:

RPAD(TABLE_NAME,38, '.')	NUM_ROWS
TEMP_ERRORS.....	9
CUSTOMERS.....	367,296

RTRIM(<*c1*>,<*c2*>)

Where *c1* and *c2* are character strings. This function returns *c1* without any trailing characters that appear in *c1*. If no *c2* characters are trailing characters in *c1*, then *c1* is returned unchanged. *c2* defaults to a single space. See also LTRIM.

SQL>*SELECT RTRIM ('Mississippi', 'ip') FROM dual;*

OUTPUT:

RTRIM('
Mississ

REPLACE (<*c1*>, <*c2*>[, <*c3*>])

Where *c1, c2* and *c3* are all characters string. This function returns *c1* with all occurrences of *c2* replaced with *c3*. *c3* defaults to NULL. If *c3* is NULL, all occurrences of *c2* are removed. If *c2* is NULL, then *c1* is returned unchanged. If *c1* is NULL, then NULL is returned.

SQL>*SELECT REPLACE ('uptown', 'up', 'down') FROM dual;*

OUTPUT:

REPLACE (
downtown

SUBSTR (<*c1*>, <*i*>[,<*j*>])

Where *c1*is a character string and both *i* and *j* are integers. This function returns the portion of *c1* that is *j* characters long, beginning at position *i*. If *j* is negative, the position is counted backwards (that is, right to left). This function returns NULL if *i* is 0 or negative. *j* defaults to 1.

SQL>*SELECT SUBSTR ('Message', 1,4) from dual;*

OUTPUT:

SUBS

Mess

TRANSLATE(<*c1*>, <*c2*>, <*c3*>)

Where *c1, c2,* and *c3* are all character string. This function returns *c1* with all occurrences of character in *c2* replaced with the position ally corresponding character in *c3*. A NULL is returned if any of *c1, c2,* or *c3* is NULL. If *c3* has fewer characters than *c2*, then the unmatched characters in *c2* are removed from *c1*. If *c2* has fewer characters than *c3*, then the unmatched characters in *c3* are ignored.

SQL>*SELECT TRANSLATE ('fumble', 'uf', 'aR') test FROM dual;*

OUTPUT:

TEST

Ramble

TRIM (string[,char(s)])

It removes all the blank spaces from the left as well as right side of the string if no char is specified. If we give an char, then it removes the leading and trailing occurrences of that character from the string. This function is new to 8i.

SQL>*SELECT TRIM(' space padded ') trimmed FROM dual;*

OUTPUT:

TRIMMED

---

space padded

UPPER (<*c*>)

Where *c* is a character string. This function returns the character string *c* with all characters in upper case. UPPER frequently appears in WHERE clauses.

Example:

SQL>*select upper(dname) from dept ;*

OUTPUT:

UPPER(DNAME)
ACCOUNTING
RESEARCH
SALES
OPERATIONS

Character Function Summary

Function	**Description**
ASCII	Returns the ASCII decimal equivalent of a character
CHR	Returns the character given the decimal equivalent
CONCAT	Concatenates two string; same as the operator ||
INITCAP	Returns the string with the first letter of each word in uppercase
LENGTH	Returns the length of a string in characters
LOWER	Converts string to all lowercase
LPAD	Left-fills a string to a set length using a specified character
LTRIM	Strips leading characters from a string
RPAD	Right-fills a string to a set length using a specified character
RTRIM	Strips trailing characters from a string
REPLACE	Performs substring search and replace
SUBSTR	Returns a section of the specified string, specified by numeric character positions
TRANSLATE	Performs character search and replace
TRIM	Strings leading, trailing, or both leading and trailing characters from a string
UPPER	Converts string to all uppercase

SINGLE-ROW NUMERIC FUNCTIONS

Single-row numeric functions operate on numeric data and perform some kind of mathematical or arithmetic manipulation. All have numeric arguments and returns numeric values.

ABS(<*n*>)

Where *n* is a number. This function returns the absolute of n.

SQL>*SELECT ABS(-52) negative, ABS(52) positive FROM dual;*

OUTPUT:

NEGATIVE	POSITIVE
52	52

CEIL(<*n*>)

Where *n* is a number. This function returns the smallest integer that is greater than or equal to *n*. CEIL rounds up to a whole number. See also FLOOR.

SQL>*SELECT CEIL (9.8), CEIL(-32.85), CEIL(0) FROM dual;*

OUTPUT:

CEIL(9.8)	CEIL(-32.85)	CEIL (0)
10	-32	0

COS(<n>**)**

It returns trigonometric cosine of the number n.

SQL> *SELECT COS(45) FROM DUAL;*

OUTPUT:

COS(45)
.52532199

EXP(<*n*>)

Where *n* is a number. This function returns *e* (the base of natural logarithms) raised to the n^{th} power.

SQL>*SELECT EXP(1) "e" FROM dual*;

OUTPUT:

e
2.71828183

FLOOR(<*n*>)

Where *n* is a number. This function returns the largest integer that is less than or equal to *n*. FLOOR round down to a whole number. See also CEIL.

SQL>*SELECT FLOOR(9.8), FLOOR(-32.85), FLOOR(137) FROM dual;*

OUTPUT:

FLOOR(9.8)	FLOOR(-32.85)	FLOOR(137)
9	-33`	137

LOG(<*n1*>, <*n2*>)

Where *n1* and *n2* are numbers. This function returns the logarithm base *n1* of *n2*.

SQL>*SELECT LOG(8,64), LOG(3,27), LOG(2,1024) FROM dual;*

OUTPUT:

LOG(8,64)	LOG(3,27)	LOG(2,1024)
2	3	10

MOD(<*n1*>, <*n2*>)

Where *n1* and *n2* are numbers. This function returns *n1* modulo *n2* or the remainder of *n1* divided by *n2*. If *n1* is negative, the result is negative. The sign of *n2* has no effect on the result. This behavior differs from the mathematical definition of the modulus operation.

SQL>*SELECT MOD(14,5), MOD(8,2.5), MOD(-64,7) FROM dual;*

OUTPUT:

MOD(14,5)	MOD(8,2.5)	MOD(-64,7)
4	.5	-1

POWER(<*n1*>, <*n2*>)

Where *n1* and *n2* are numbers. This function returns *n1* to the $n2^{th}$ power.

SQL>*SELECT POWER(2, 10), POWER(3,3), POWER(5,3) FROM dual;*

OUTPUT:

POWER(2,10)	POWER(3,3)	POWER(5,3)
1024	27	125

ROUND(<*n1*>, <*n2*>)

Where *n1* and *n2* are numbers. This function returns *n1* rounded to *n2* digits of precision to the right of the decimal. If *n2* is negative, *n1* is rounded to left of the decimal. This function is similar to TRUNC( ).

SQL>*SELECT ROUND(12345,-2), ROUND(12345.54321,2) from dual;*

OUTPUT:

ROUND(12345,-2)	ROUND(12345.54321,2)
12300	12345.54

SIGN(<*n*>)

Where *n* is a number. This function returns -1 if *n* is negative, 1 if *n* is positive, and 0 if *n* is 0.

SQL>*SELECT SIGN(-2.3), SIGN(0), SIGN(47) FROM dual;*

OUTPUT:

SIGN(-2.3)	SIGN(0)	SIGN(47)
-1	0	1

SQRT (<*n*>)

Where *n* is a number. This function returns the square root of *n*.

SQL>*SELECT SQRT(64), SQRT(49), SQRT(5) FROM dual;*

OUTPUT:

SQRT(64)	SQRT(49)	SQRT(5)
8	7	2.23606798

TRUNC (<*n*>)

Where *n1* is a number and *n2* is an integer. This function returns *n1* truncated to *n2* digits of precision to the right of the decimal. If *n2* is negative, *n1* is truncated to left of the decimal. See also ROUND.

SQL>*SELECT TRUNC(123.456,2) pos, TRUNC(123.456,-1) neg FROM dual;*

OUTPUT:

POS	NEG
123.45	120

Comparison of ROUND and TRUC Functions

The comparison of Round and Trunc functions has been further illustrated below:

Example	OUTPUT OF ROUND	TRUNC	Output of TRUNC
ROUND(6876.678, -1)	6880	TRUNC(6876.678, -1)	6870
ROUND(6876.678, -2)	6900	TRUNC(6876.678, -2)	6800
ROUND(6876.678, -3)	7000	TRUNC(6876.678, -3)	6000
ROUND(6876.678, -4)	10000	TRUNC(6876.678, -4)	0
ROUND(6876.678, -5)	1	TRUNC(6876.678, -5)	0
ROUND(6876.678, -6)	0	TRUNC(6876.678, -6)	0

Numeric Function Summary

Function	Description
ABS	Returns the absolute value
CEIL	Returns the next higher integer
COS	Returns the cosine
EXP	Returns the base of natural logarithms raised to a power
FLOOR	Returns the next smaller integer
LN	Returns the natural logarithm
LOG	Returns the logarithm
MOD	Returns modulo (remainder) of a division operation
POWER	Returns a number raised to an arbitrary power
ROUND	Rounds a number
SIGN	Returns an indicator of sign: negative, positive, or zero
SIN	Returns the sine
SQRT	Returns the square root of a number
TRUNC	Truncates a number

Single-Row Date Functions

Single-row date functions operate on date data type.

ADD_MONTHS(<*d*>, <*i*>)

Where *d* is a date and *i* is an integer. This function returns the data *d* plus *i* months. If *i* is a decimal number, the database will implicitly convert it to an integer by truncating the decimal portion (for example, 3.9 becomes 3).

SQL> *SELECT SYSDATE, ADD_MONTHS(SYSDATE,3)plus_3,* ADD_MONTHS(SYSDATE,-2) minus_2 FROM DUAL;

OUTPUT:

SYSDATE	PLUS_3	MINUS_2
01-JAN-98	01-APR-98	01-NOV-97

LAST_DAY(<*d*>)

Where *d* is a date. This function returns the last day of the month for the date *d*.

SQL>*SELECT SYSDATE, LAST_DAY(SYSDATE)+1FROM dual;*

OUTPUT:

SYSDATE	LAST_DAY(SY
23-NOV-1999	01-DEC-1999

MONTHS_BETWEEN(<*d1*>, <*d2*>)

Where *d1* and *d2* are both dates. This function returns the number of months that *d2* is later then *d1*. A whole number is returned if *d1* and *d2* are the same day of the month or if both dates are the last day of a month.

SQL>*SELECT MONTHS_BETWEEN ('19-Dec-1999', '19-Mar-2000') FROM dual;*

OUTPUT:

MONTHS_BETWEEN('19-DEC-1999', '19-MAR-2000')
3

NEXT_DAY(<*d*>, <*dow*>)

Where *d* is a date and *dow* is a text string containing the full or abbreviated day of the week in the session's language. This function returns the next *dow* following *d.* The time portion of the return date is the same as the time portion of *d.*

SQL> *SELECT NEXT_DAY('01-Jan-2004','Monday') "1st Monday" FROM dual;*

OUTPUT:

1st Monda
05-JAN-04

ROUND(<*d*>[, <*fmt*>])

Where *d* is a date and *fmt* is a character string containing a date-format string.

SQL> *SELECT SYSDATE, ROUND(SYSDATE, 'MM') FROM dual;*

OUTPUT:

SYSDATE	ROUND(SYS
16-JAN-98	01-FEB-98

SYSDATE

This function takes no arguments and returns the current date and time to the second level.

SQL>*SELECT SYSDATE FROM dual;*

OUTPUT:

SYSDATE
24-Nov-1999 09:26:01

Arithmetic with Dates

Important points regarding arithmetic operations on dates are given below:

- Add or subtract a number to or from a date for a resultant date value.
- Subtract two dates to find the number of days between those dates.
- Add hours to a date by dividing the number of hours by 24.

Examples:

SQL> *SELECT SYSDATE+2 FROM DUAL;*

OUTPUT:

SYSDATE+2
09-JUN-16

It shows the date after two days.

SQL> *SELECT ROLLNUMBER,SYSDATE-DATEOFBIRTH FROM STUDENT;*

OUTPUT:

ROLLNUMBER	SYSDATE - DATEOFBIRTH
1	9674.917997685185185185185185185185185 19
2	9520.917997685185185185185185185185185 19
3	9288.917997685185185185185185185185185 19
4	8761.917997685185185185185185185185185 19
5	9454.917997685185185185185185185185185 19

It provides the age in number of days, it is important to note that it provides number of days up to points depending upon the time of day.

SQL> *SELECT ROLLNUMBER,DATEOFBIRTH, SYSDATE,(SYSDATE-DATEOFBIRTH)/365 FROM STUDENT;*

OUTPUT:

ROLLNUMBER	DATEOFBIRTH	SYSDATE	(SYSDATE - DATEOF BIRTH)/365
1	12-DEC-89	07-JUN-16	26.506630580923389142567224759005580 9234
2	15-MAY-90	07-JUN-16	26.084712772704211060375443937087772 7042
3	02-JAN-91	07-JUN-16	25.449096334348046676813800101471334 3481
4	12-JUN-92	07-JUN-16	24.005260717909690512430238457635717 9097
5	20-JUL-90	07-JUN-16	25.903890854895991882293252156265854 896

It provides the age in years.

Date Function Summary

Function	Description
ADD_MONTHS	Adds a number of months to a date
LAST_DAY	Returns the last day of a month
MONTHS_BETWEEN	Returns the number of months between two dates
NEXT_DAY	Returns the next day of a week following a given date
ROUND	Rounds a date/time
SYSDATE	Returns the current date/time

Single-Row Conversion Functions

Single-row conversion functions operate on multiple data types. In Oracle, we have To_NUMBER, TO_DATE and TO_CHAR as inbuilt functions for explicit data types conversion as shown in figure 5.4.

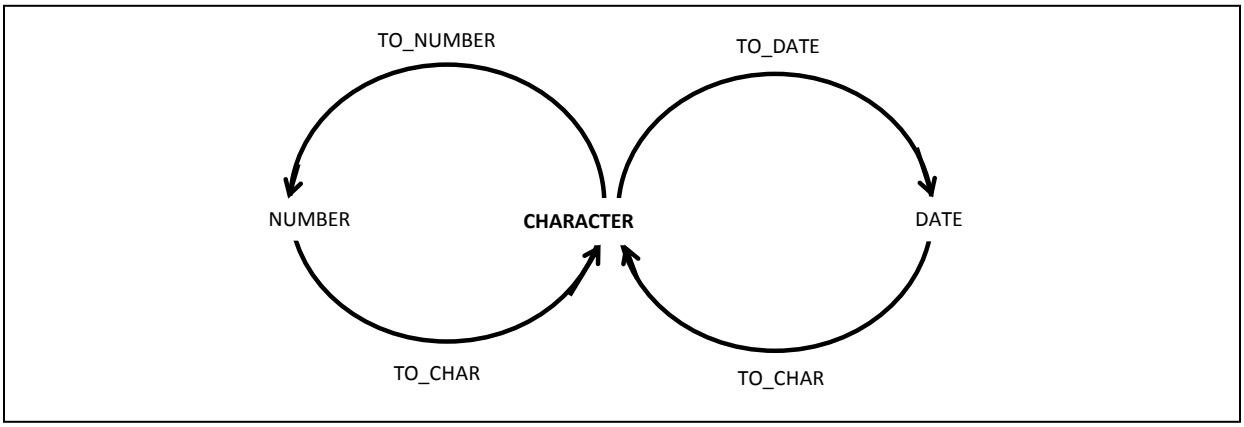


Figure 24.4: Explicit Data Type Conversion

To_char(number|date,fmt)

This function converts a number or date value to a varchar2 character string with format model fmt . It facilitates the retrieval of data in a format different from the default format (DD-MON-YY) when using for dates. With the help of this function part of the date i.e the date, month or year can also be extracted. While using this function to convert dates following guidelines must be followed:

- the format model must be enclosed in single quotation marks and is case sensitive
- the format model can include any valid date format element be sure to separate the date value from the format model by the comma
- the names of days and months in the output are automatically padded with blanks
- you can resize the display width of the resulting character field with the SQL * Plus column command.

SQL>*Select sysdate, to_char(sysdate ,'DAY') from dual;*

OUTPUT:

SYSDATE	TO_CHAR(S
07-JULY-03	MONDAY

Example: To display current time in three different columns in form of hour, minutes and second.

SQL> *SELECT TO_CHAR(SYSDATE,'HH') HOUR, TO_CHAR(SYSDATE,'MI') MIN,TO_CHAR(SYSDATE,'SS') SEC FROM DUAL;*

OUTPUT:

HO	MI	SE
--	--	--
03	01	16

Example: To display current date and time.-

SQL>*SELECT TO_CHAR(SYSDATE, 'DD-MON-YYYY HH24:MI:SS') CURRENT_DATE_TIME FROM DUAL;*

OUTPUT:

CURRENT_DATE_TIME
07-JUN-16 22:08:33

Example: To display current day.

SQL> *SELECT TO_CHAR(SYSDATE, 'DAY') CURRENT_DAY FROM DUAL;*

OUTPUT:

CURRENT_DAY
TUESDAY

Following table shows all the available options for date format.

Elements of the Date Format Model

YYYY	Full year in numbers
YEAR	Year spelled out
MM	Two-digit value for month
MONTH	Full name for month
MON	Three-letter abbreviation of the month
DY	Three-letter abbreviation of the day of the week
DAY	Full name of the day of the week
DD	Numeric day of the month

SQL>*Select to_char(sal ,'$99,999') salary from emp where ename ='SCOTT' ;*

OUTPUT:

SALARY
$3,000

To_date(char[,'fmt'])

This function converts a character value into a date value where *char* stand for the value to be inserted in the date column and *fmt* is the date format in which *char* is specified.

SQL>*Select to_date('07-july-03','RM') from dual ;*

OUTPUT:

to_date('07-july-03'),'RM'
7

To_number(text|date)

This function which converts text or date information into a number.

SQL>*SELECT TO_NUMBER('49583') FROM DUAL;*

OUTPUT:

TO_NUMBER('49583')
49583

YY and RR Date Format

There are two dates format, i.e, RR and YY format. These formats play vital roles when we specify only digits of year. In case of YY format when we specify two digits of year, the other digits (20) are automatically assigned to the current century, i.e, 99 will be considered ir stored as 2099.

RR converts two-digit years into four-digit years by rounding. It means, 50-99 are stored as 1950-1999, and dates ending in 00-49 are stored as 2000-2049. RRRR accepts a four-digit

input (although not required), and converts two-digit dates as RR does. YYYY accepts 4-digit inputs but doesn't do any date converting

Examples:

```
SQL> SELECT ENAME FROM EMP WHERE HIREDATE = TODATE('01/05/81','DD/MM/YY');
```

```
SQL>SELECT ENAME FROM EMP WHERE HIREDATE = TODATE('01/05/81','DD/MM/RR');
```

Here, in first query it will search records for hiredate 2081 while in second query it will search for hiredate 1981. Obviously, first query will not return any records while second query will work.

The further illustration of YY and RR date format has been given below:

RR Date Format

Current Year	Specified Date	RR Format	YY Format
1976	25-FEB-85	1985	1995
1976	27-OCT-16	2016	1916
2016	27-OCT-16	2016	2016
2016	27-OCT-95	1995	2095

		If the specified two-digit year is:	
		0 - 49	50 - 99
If two digits of the current year are:	0 - 49	The return date is in the current century	The return date is in the century before the current one
	50 - 99	The return date is in the century after the current one	The return date is in the current century

MULTI ROW OR GROUP FUNCTIONS IN SQL

Group functions, sometimes-called aggregate functions, return a value based on a number of inputs. The exact number of input is not determined until the query is executed and all rows are fetched. This differs from single-row functions, in which the number of inputs is known at parse time before the query is executed. Because of this difference, group functions have slightly different requirements and behavior from single-row functions. Group functions do not process NULL values and do not return a NULL value.

Aggregate Functions

The aggregate functions produce a single value for an entire group or table.

Aggregate Functions	Description
COUNT	Determine the number of rows or non NULL column values
SUM	Determines the sum of all selected columns
MAX	Determines the largest of all selected values of a column
MIN	Determines the smallest of all selected values of a column
AVG	Determines the average of all selected values of a column

NOTE: In all the above functions, NULLs are ignored.

Aggregate functions are used to produce summarized results. They operate on sets of rows. They return results based on groups of rows. By default all rows in a table are treated as one group. The GROUP BY clause of the SELECT statement is used to divide rows into smaller groups (discussed in next section).

COUNT

Count function determines the number of rows or non-NULL column values.

The syntax is

```
COUNT (*|[Distinct]| ALL|column name)
```

If * is passed, then the total number of rows is returned.

Examples:

- List the number of employees working with the company:

SQL>*Select COUNT(*) FROM emp;*

OUTPUT:

```
COUNT(*)
--------
      14
```

- List the number of jobs available in the emp table.

SQL>*SELECT COUNT (DISTINCT JOB) FROM emp;*

OUTPUT:

```
COUNT(DISTINCTJOB)
------------------
                 5
```

With the COUNT function, a column name can also be specified. In this case, the NULL values will be ignored.

SUM

The sum function returns the sum of values for the selected list of columns.

The syntax is

```
SUM([DISTINCT|ALL] column name)
```

Example

- List the total salaries payable to employees.

SQL>*SELECT SUM(sal) FROM emp;*

OUTPUT:

```
SUM(SAL)
--------
   29025
```

MAX

Max function returns the maximum value of the selected list of item. Note that DISTINCT and ALL have no effect, since the maximum value would be same in either case.

The syntax is:

```
MAX(column name)
```

- List the maximum salary of employee working as a salesman.

SQL>*SELECT MAX(sal) FROM emp WHERE job='SALESMAN';*

OUTPUT:

```
MAX(SAL)
---------
1600
```

MIN

This function returns the minimum value of the selected list of items.

The syntax is

```
MIN(Column name)
```

Example

- List the minimum salary from emp table.

SQL>*SELECT MIN(sal) FROM emp;*

OUTPUT:

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

AVG

This function returns the average of column values.

The syntax is:

```
AVG(DISTINCT|ALL] Column name)
```

Example

- List the average salary and number of employees working in the department 20.

SQL>*SELECT AVG(sal), COUNT(*) FROM emp WHERE deptno=20;*

OUTPUT:

AVG(SAL)	COUNT(*)
2175	5