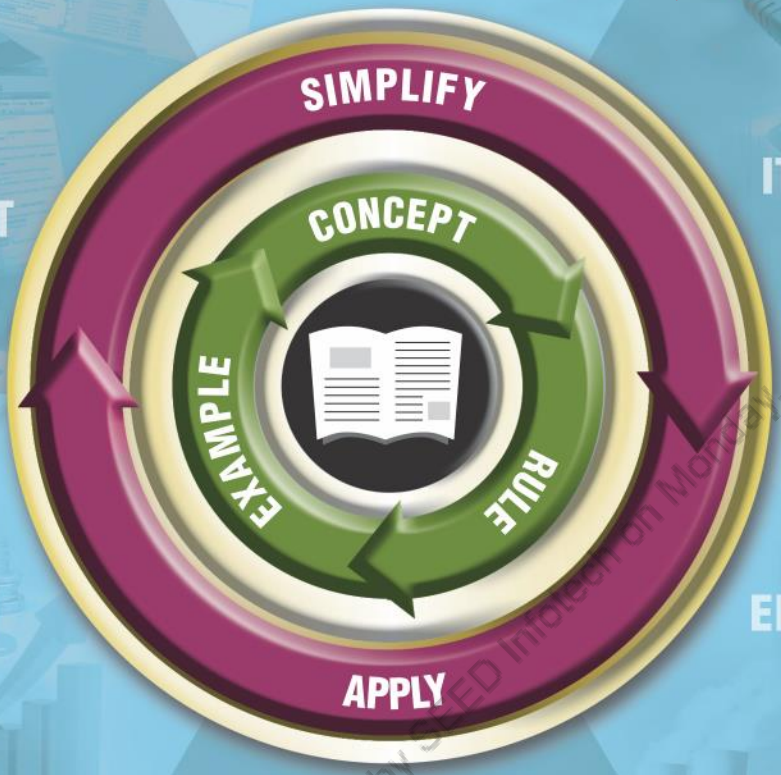
SOFTWARE TESTING

SOFTWARE DEVELOPMENT

IT-IMS



BFSI

ENGINEERING SERVICES

SOFT SKILLS

N·S·D·C National Skill Development Corporation
Transforming the skill landscape
Training Partner

seed
Official Curriculum

Learning SQL

Official Curriculum of SEED Infotech Ltd.

seed®
beyond the obvious

Published by :

Avani
publications

Author : Mr. Milind Choudhari

This edition has been printed and published in house by Avani Publications.

Print Edition : December 2013

Issue No./ Date : 01 / May 12, 2012 **Revision No. & Date** : 00 / May 12, 2012

Avani Publications

15A-1/2/4, Anandmayee Apartment, Off Karve Road, Pune 411004

Contents

Chapter 1

Introduction to SQL*PLUS

Objectives

At the end of this chapter, student will be able to

- Describe the SQL*PLUS environment.
- Explain and use the SQL*PLUS environment commands.

Introduction

Structure Query Language (SQL) is the language used by all the Relational Database Management Systems (RDBMS) to enable the user to access the data. Using SQL one can insert, retrieve, delete or modify the data. SQL is a non-procedural language and PL/SQL is Oracle's procedural language extension to SQL.

When we install ORACLE Server, it also installs certain tools and utilities. SQL*PLUS is one of such utilities, which is part of ORACLE. SQL*PLUS environment is used in various stages of application development. This chapter first explains what SQL, PL/SQL and SQL*PLUS is and then covers how to use the SQL*PLUS environment.

What is SQL, PL/SQL and SQL*PLUS?

While working with ORACLE, we come across certain buzz words like SQL, PL/SQL and SQL*PLUS. Let us understand what they mean.

What is SQL?

SQL is "Structured Query Language". SQL, pronounced as 'EsQueEl' or 'SeeQuel', is a powerful query language that is used with relational database management systems. It was first designed for use with IBM's database management system. In 1986, The

American National Standards Institute (ANSI) made SQL the standard for all RDBMS.

SQL is the language using which we can perform the operations like create, retrieve, add, modify, delete and control access to the data in ORACLE database. SQL is a non-procedural language. It does not support programming language features like variables, programming constructs.

What is PL/SQL?

PL/SQL is ORACLE proprietary procedural language extension to the SQL. SQL has certain limitations like one can not declare variables and use programming constructs for controlling flow of the program.

Using PL/SQL we can

- declare variables.
- use control structures to control the flow of a program.
- use modular programming features to decompose a program into modules in the form of subprograms. Subprograms can be procedures, functions and packages.

PL/SQL is a language that is supported by various ORACLE products like ORACLE RDBMS, ORACLE Forms and Reports etc. Due to this the application developer now has to learn a single language.

What is SQL*PLUS?

SQL*PLUS is not a language. It is a powerful support product, which comes with ORACLE. It is an environment in which we can run SQL commands and PL/SQL blocks. We can use this environment to write and test the code. This code can then be used from SQL*PLUS itself or other ORACLE tools like Forms, Reports and others.

SQL*PLUS environment is used in various stages of the application development. ORACLE also comes with an additional utility call SQL Worksheet for using SQL and PL/SQL.

We can use SQL commands, PL/SQL blocks and SQL*PLUS

commands in SQL*PLUS environment. We will discuss more about SQL*PLUS in the following section.

Overview of SQL*PLUS

SQL*PLUS is a program that is used in conjunction with SQL, the standard database language and PL/SQL, the procedural extension to SQL. The SQL*PLUS environment enables us to manipulate SQL Commands, PL/SQL blocks and many additional tasks as well.

In SQL*PLUS environment we can

- List the structure or definition of the database schema objects like table, views.
- Enter, edit, store, retrieve and run SQL commands and PL/SQL blocks
- Format and perform calculations on data retrieved from the database.
- Store and print the query results.

The commands supported by SQL*PLUS environment can be divided into three categories.

- SQL commands
- PL/SQL blocks
- SQL*PLUS commands

SQL Commands

SQL commands are standard SQL commands used for working with the data in the database. SQL is made of three sub-languages. These sub-languages are Data Definition Language (DDL), Data Manipulation Language (DML) and Data Control Language (DCL)

The Data Definition Language consists of a set of commands used to create the database objects such as tables, views, and indexes. The Data Manipulation Language is used for query, insertion, updating and deletion of data stored in the database. The Data Control Language is used for controlling access to the data.

PL/SQL Commands

PL/SQL blocks are pieces of code in the form of SQL and PL/SQL statements. The PL/SQL code is written in a block structured fashion. PL/SQL blocks are also used for working with data in the database.

SQL*PLUS commands

SQL*PLUS commands are SQL*PLUS environment commands for doing additional tasks along with database operations.

These are set of commands using which we can format query results, set environment options, edit and store SQL commands and PL/SQL blocks.

Working with SQL*PLUS

In this section we describe how to start the SQL*PLUS Environment and use SQL*PLUS commands.

Starting SQL*PLUS Environment

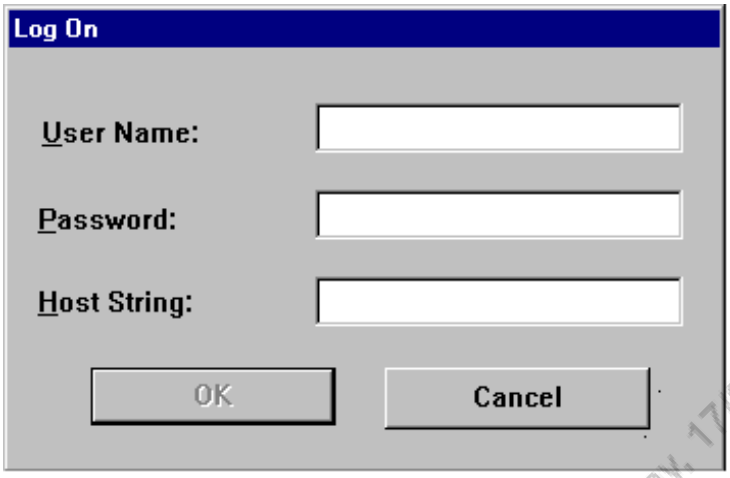
SQL*PLUS is installed on both Server and Client code. The basic SQL*PLUS environment works like a command line interpreter. Typically on Clients, it runs on some GUI environment and at Server it may be running on GUI based or characters based environment.

Invoking SQL*PLUS in a GUI based environment

- Turn on the computer
- Start WINDOWS
- Double Click the SQL*PLUS icon on the desktop

  Or
- Click on the start button and from the menu hierarchy, select SQL PLUS command.

- SQL*PLUS displays a login dialog box.



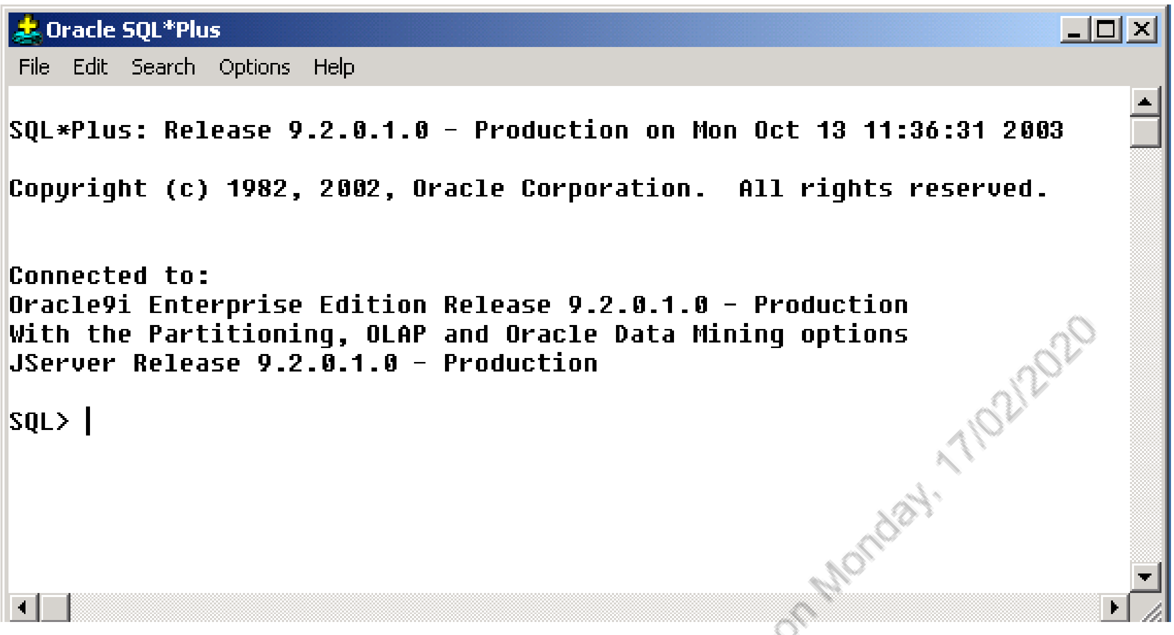
  - Type a valid user name, password and host string (if any) information.
  - SQL*PLUS environment has been invoked.
    - This starts SQL*PLUS session and shows us a prompt

SQL>

  - Now, we can use commands to perform our task.
  - To exit the environment, type command

SQL> Exit

and press enter.



When using SQL*PLUS under WINDOWS, it combines the basic command line interpreter environment with graphical environment of WINDOWS. SQL*PLUS under WINDOWS has menu based features along with the command line environment.

With menu based features, we can

- Save and open SQL scripts.
- Copy and paste information to and from the WIN95/NT clipboard and the SQL*PLUS.
- Invoke and use the WINDOWS user friendly text editor like notepad rather than using command line for editing the SQL commands.
- Perform searching / replacing and other functions.

Commands and Conventions

The SQL*PLUS environment recognizes various types of commands like SQL commands, PL/SQL blocks and SQL*PLUS environment command.

In order to distinguish between these commands we have separate command termination character for each category

- An SQL command always ends with a semicolon - ‘ **;** ‘
- A PL/SQL block ends with a (period) - ‘**.** ‘
- An SQL*PLUS environment command does not require any termination character.

SQL*PLUS Environment Commands

The following section describes the SQL*PLUS environment editing commands like Append, Change and other editing commands.

Editing Commands

The SQL*PLUS environment is a command line interpreter environment. This means, when we type a command and terminate by appropriate terminator, it is executed immediately. The current SQL command or a PL/SQL block, which we write in SQL*PLUS environment is stored in its buffer and can be edited. Following commands describe how to use the editing command like append text, change text, delete line and others.

1. APPEND

Syntax : A [PPEND] TEXT

Description: The Append command appends the specified TEXT to the current line

(Note : Here the square brackets indicate the optional part of the command. So append command can be typed as ‘APPEND’ or ‘A’. Commands are not case sensitive.)

Example:

Suppose, we have typed the following command

```
SQL>select * from emp where job ='CLERK';
```

To modify this command by appending additional criteria, we can use the

Append command as follows

```
SQL> APPEND  and salary < 1500;
```

To list the modified command, use the List command (explained later)

SQL> LIST

1* select * from emp where job ='CLERK' and sal < 1500

2. Change

Syntax 1 : C[hange] /Old text / New text

Description: The Change command changes the old text to the new text in the current line

Example:

Suppose, we type the following command

```
SQL> select * from emp
2 where dept_id=10;
```

We will get an error, because dept_id column is not existing in emp table.

```
where dept_id=10
   *
ERROR at line 2:
ORA-00904: invalid column name
```

We can change dept_id to dept using the Change command.

```
SQL>C/dept_id/deptno;
2*   where deptno=10
```

To list the modified command, use the List command.

```
SQL>List
1 select * from emp
2 where deptno =10
```

Syntax2 : C[hange] / TEXT

Description: Removes the text from the current line.

Example:

Suppose our current command is

```
1* Select * from emp, dept
```

To remove ‘, dept’ from this command use the Change command in the following way

```
SQL>c/ , dept
```

To list the modified command, use the List command.

```
SQL>List
1* Select * from emp
```

3. Del

Syntax : Del

Description: Deletes the current line from the SQL buffer.

Example:

Suppose, the existing command is

```
1    Select * from emp
2 *  where deptno = 10
```

To remove the second line from this command use Del command

```
SQL>Del
```

To list the modified command, use the List command.

```
SQL>L
1* Select * from emp
```

4. Clear Buffer

Syntax : Clear Buffer

Description: Clears the contents of the SQL buffer.

Example:

Suppose, the existing command is

```
1    Select * from emp
2 *  where deptno = 10
SQL>Clear buffer
```

We will get the message

Buffer cleared

5. Input

Syntax 1 : I[nput]

Description: Adds one or more lines to the SQL buffer.

Example:

Suppose, the existing command is

```
SQL> select * from emp;
```

To add a line to the existing command, use the Input command

```
SQL> input
2  where deptno = 10;
```

To list the modified command, use the List command.

```
SQL> L
1  select * from emp
2* where deptno = 10
```

Syntax 2 : I[nput] TEXT

Description: Adds a line consisting of text to current line of the SQL buffer

Example:

Suppose, the existing command is

```
  1  select * from emp
    2* where deptno = 10
```

To add a text to the current line, use the Input command

```
SQL> Input and job = 'CLERK'
```

To list the modified command, use the List command.

```
SQL> L
    1  select * from emp
    2  where deptno = 10
    3* and job = 'CLERK'
```

6. List

Syntax1 : L[ist]

Description: List command lists all the lines in a SQL buffer.

Example:

To list the contents of the SQL buffer, use the List command.

```
SQL>L
1* select * from emp
```

Syntax2 : L[ist] n

Description: List the line numbers n from SQL buffer.

Example:

```
SQL> L 2
2* where deptno = 30
```

Syntax3 : L[ist] *

Description: Lists the current line of the SQL buffer.

Example:

```
SQL>L *
  1 select * from emp
2* where deptno = 30
```

Syntax4 : L[list] Last

Description: Lists the last line of the SQL buffer.

Example:

```
SQL> L last
2     where deptno= 30
```

Syntax5 : L[ist] m n

Description: Lists a range of lines between line number m to n in a SQL buffer.

Example:

```
SQL> L 1 2
```

```
1 select * from emp
2*where deptno = 30
```

Other Commands

This section explains the commands like describe, edit, exit and other SQL*PLUS commands.

Describe

Syntax : Desc[ribe] table name

Description: Describe command lists the structure of a table.

Example:

```
SQL> Desc emp;
```

Name	Null?	Type
Empno	not null	NUMBER(4)
Ename		VARCHAR2(25)
Job		VARCHAR2(25)
Mgr		NUMBER(4)
Hiredate		DATE
Sal		NUMBER(7,2)
comm.		NUMBER(7,2)
Deptno		NUMBER(2)

Edit

Syntax : Ed[it] [File name]

Description: Opens the operating system text editor and fetches

the contents of the file specified using filename. If filename is not specified contents of the SQL buffer are displayed.

Example:

```
SQL> ED emp_record.sql
```

Exit

Syntax : Exit

Description: Exits the SQL*PLUS environment and closes the current session.

Example:

```
SQL> Exit
```

This will close down the SQL*PLUS session.

GET

Syntax : GET Filename

Description: Loads the specified operating system file into a SQL buffer.

Example:

To load the contents of a emp_record.sql file to the SQL buffer, we can use the Get command.

```
SQL>GET emp_reocrd.sql
```

HOST

Syntax : HO[st] Command

Description: Executes the specified host operating system command.

Example:

```
SQL>Host
```

takes us to the Operating System's command prompt

```
C:\>
```

We can issue any host command now

```
C:\>exit
```

Type exit to return to SQL prompt from operating system's command prompt.

START

Syntax : STA[RT] file name

Description: Loads the contents of the specified file into the SQL buffer and executes it.

Example:

```
SQL>Start emp_record.sql
```

This will load the contents of the emp_record.sql file and will execute the command loaded.

Save

Syntax : SAV[E] File name [REPLACE/APPEND]

Description: Saves the contents of the SQL buffer to the specified operating system file.

Example:

```
SQL> Save emp_record.sql
```

This will save the contents of the SQL buffer to the file emp_record.sql by giving us the message

Created file emp_record.sql

If that file already exists then for replacing the content of a file with current one, use

Syntax:

SAV[E] File name REPLACE

Example:

```
SQL> Save emp_record.sql REPLACE
```

If you want to append the content at the end of existing file then use

Syntax:

SAV[E] File name APPEND

Example:

```
SQL> Save emp_record.sql APPEND;
```

Run

Syntax : RUN

Description: Executes the command in SQL buffer.

Example:

Use the Get command to load the contents of a file in the SQL buffer.

```
  SQL>GET emp_record.sql;
```

We can run the command from the SQL buffer using Run command.

```
SQL>RUN
```

/ (Slash)

Syntax : /

Description: Runs the command from SQL buffer.

Example:

```
SQL> GET emp_record.sql
```

We can run the command from the SQL buffer using Run command or by using / (Slash)

```
SQL> /
```

@ ('at the rate' Sign)

Syntax : @

Description: Loads the contents of the operating system file into the SQL buffer, lists the command and executes the command.

Example:

```
SQL> @emp_record.sql;
```

Summary

- SQL*PLUS is a powerful support product that comes with ORACLE.
- SQL*PLUS is installed on both, the Client and the Server.
- The SQL*PLUS environment command categories are SQL commands, PL/SQL commands and SQL*PLUS environment commands.
- A semicolon terminates SQL commands.
- PL/SQL Blocks are terminated by a period.
- SQL*PLUS environment commands don't require any command terminator.

Chapter 2

ORACLE Data Types

Objectives

At the end of this chapter, student will be able to :

- List ORACLE in-built Data Types.
- Explain the use of ORACLE Data Types.
- Identify data conversion functions.

Introduction

In a database, the data is stored in the form of values in tables. The data or the information could be numbers, text or character strings, date and time. For handling various types of information efficiently and conveniently, data types are used. In this chapter we discuss various data types supported in ORACLE.

While handling various types of data, we need to convert the data of one type into another or vice versa. The implicit and explicit data conversion is also discussed in this chapter.

Data Types

In a database, the data is stored in the form of values in tables. The data is organised in a table in terms of rows and columns. A table is a set of rows and a row consists of a set of columns (also called as attributes).

In a database table, various types of information is stored. The data or the information could be in the form of numbers, text or character strings, date and time. For example, in our sample table '**emp**', employee number and salary information are numeric values; employee name and job are character strings.

For handling various types of information efficiently and conveniently, the data is stored in a database using appropriate data type. When we create a table in an ORACLE database we

must specify the column names with their data type, size and constraints (if required).

The column's data type describes the type of the information, the valid data range, specific format and constraints. The column's data type describes the basic type of data that is acceptable in that column. For example, empno column in our sample table 'emp' uses the data type NUMBER. The column specification is

Empno NUMBER(4)

This means Empno column data type is Number and it can accept a four-digit number from 0 to 9999.

Various data types supported in ORACLE are as follows

- Character Data Types
- Number Data Types
- Binary Data Types
- Date Data Type
- LOB Data Type
- ROW ID

Character Data types

We can store and manipulate the words, documents or text using the character data types. The character data types are used to store character or alphanumeric data in the table. Character columns can store all alphanumeric values, but NUMBER columns can only store numeric values. ORACLE supports both single-byte and multi-byte (National Language) character sets.

The Character data types in ORACLE are

- CHAR
- VARCHAR2
- VARCHAR
- NCHAR
- NVARCHAR2
- LONG

CHAR Data Type

Syntax : CHAR(n [BYTE | CHAR]) where n = 1 to 2000

Description :

CHAR data types specify the fixed length character string. When we create the table with CHAR data types, we can specify the length in byte, which is default. If we do not specify the length, ORACLE uses the default length. The default length of CHAR data type is 1 byte. If we insert the values greater than the length of the column, ORACLE returns an error. If we insert the data shorter than the length of the column, it is padded with trailing blanks.

Example : The employee name column can be defined as follows.

Name CHAR(10)

When a CHAR qualifier is used, for example CHAR(10 CHAR), you supply the column length in characters. Its size can range from 1 byte to 4 bytes, depending on the database character set.

VARCHAR2 Data Type

Syntax : VARCHAR2 (n [BYTE | CHAR]) where n = 1 to 4000

Description :

The VARCHAR2 data type is used to store the variable-length character strings having maximum length n. The maximum size is 4000 bytes. If we insert the data shorter than the length of the column then it takes its actual size and it is not padded with trailing blanks.

Example: The declaration of employee job column in the 'emp' table is as follows

Job VARCHAR2 (10)

VARCHAR Data Type

Syntax : VARCHAR (n) where n = 1 to 4000

Description :

Currently VARCHAR data type is same as VARCHAR2 data type. ORACLE Corporation recommends using VARCHAR2 rather than

VARCHAR. In future version, VARCHAR might be used for implementing some other data type.

Example: The employee address column can be defined as follows

Address VARCHAR (25)

NCHAR Data Type

Syntax : CHAR (n) where n = 1 to 2000

Description :

NCHAR data types specify the fixed length character string data, with trailing blanks. The maximum size is 2000 bytes. The length refers to the number of characters. This data type is used with National Language Character Set.

NVARCHAR2 Data Type

Syntax : NVARCHAR2 (n) where n = 1 to 4000

Description :

The NVARCHAR2 data type is used to store the variable-length character strings having maximum length n. The maximum size is 4000 bytes. The length refers to the number of characters. This data type is used with National Language Character Set.

LONG Data Type

Syntax : LONG

Description :

This data type is used to store variable length text or character string data. This data type is used to store large text data. The maximum length is 2 GB.

This data type is supported for backward compatibility. The new applications should use LOB data types.

We can have only one long column per table and a long column has following restrictions on its use.

- Only one column per table can be defined as LONG
- A LONG column can't be indexed
- A LONG column can't be passed as argument to a procedure or function
- A function can't be used to return a LONG column
- A LONG column can't be used in where, order by, group by and other clauses

Example: The column containing comments about an employee can be defined as follows.

```
Emp_comments LONG
```


Number data types

We can use number data types to store any type of number. We can store the zero, positive and negative fixed and floating point numbers.

NUMBER Data Type

Syntax1 : NUMBER (p, s)

Where p = 1 to 38 and s = - 84 to 127.

Description :

We can use this data type to store variable length numeric data with up to 38 digits of precision having magnitude between 1.0 * 10^{-130} to 9.9….9 * 10^{125}. In the syntax, p is the precision i.e. number of digits and s is the scale i.e., number of digits to the right of decimal point.

Example: The salary column in the 'emp' table is defined as follows..

Sal NUMBER (7,2) ;

Syntax2 : NUMBER (p)

WHERE p = 1 to 38

Description :

We can use this data type to fixed-point number with a precision p and scale 0.

Example: The employee number column in the 'emp' table is defined as follows..

empno NUMBER (4)

Syntax3 : NUMBER

Description :

We can use this data type to store floating-point number with a precision of 38.

If the scale is negative, the actual data is rounded to the specified number of places to the left of the decimal point.

Binary Data Types

The binary data types are used to store the binary data. The values are inserted in columns with this data type using hexadecimal notation.

The binary data types supported in ORACLE are

- RAW
- LONG RAW

RAW Data Type

Syntax : RAW (n) where n = 1 to 2000

Description :

We can use this data type to store a variable length binary data. The values are inserted in columns with this data type using hexadecimal notation.

LONG RAW Data Type

Syntax : LONG RAW

Description :

We can use this data type to store variable length binary data of large size. It can store binary data up to 2 GB size. The values are inserted in columns with this data type using hexadecimal notation.

Date Data Type

Syntax : DATE

Description :

This Date data type stores century, year, month, day, date, hours, minutes and seconds. Valid dates range from January 1, 4712 BC to December 31, 9999 AD. It requires 7 bytes for storing date and time information. The default date format is 'DD-MON-YY'. If date is specified without time component, the default time is 12:00:00 a.m. (midnight).

Example: The hiredate column in the 'emp' table is defined as follows..

```
hiredate DATE
```

TimeStamp datatype

Syntax : TIMESTAMP [(F_S_Size)]

Description :

The TIMESTAMP data type is almost identical to DATE and differs in only one way, that it can represent fractional seconds. While declaring TIMESTAMP data type precision for fractional seconds can be specified in the range of 0 to 9, default is 6.

Example : The hiredate column in the 'emp' table can be defined as follows..

```
hiredate TIMESTAMP(5)
```

TimeStamp With Time Zone datatype

Syntax : TIMESTAMP [(F_S_Size)] WITH TIME ZONE

Description :

The TIMESTAMP WITH TIME ZONE data type is an extension of

TIMESTAMP where you can specify time zone also. This functionality is required because a specific date and time value is ambiguous unless its time zone is known. In timestamp literal, the time zone can be represented as a positive or negative offset from Coordinated Universal Time (UTC).

TimeStamp With Local Time Zone datatype

Syntax : TIMESTAMP [(F_S_Size)] WITH LOCAL TIME ZONE

Description :

The TIMESTAMP WITH TIME ZONE data type is an extension of TIMESTAMP with the addition of assumed time zone. When date time value is inserted into TIMESTAMP WITH LOCAL TIME ZONE column, the time is converted into the database's time zone.

LOB Data Types

The LOB (Large Object) data types are used to store large data. Various LOB data types are

- BLOB
- CLOB
- NCLOB
- BFILE

BLOB Data Type

Description:

This data type is used to store large binary data. The maximum size is 4GB.

CLOB Data Type

Description :

This data type is used to store large character data. The maximum size is 4GB.

NCLOB Data Type

Description :

This data type is used to store large character data. The maximum size is 4GB. This data type is used with National Language Character set.

BFILE Data Type

Description :

This data type is used with large binary data stored outside the database in an external file. The maximum size is 4GB.

ROWID Data Type

This data type is used with binary data, representing the row address. Each row stored in an ORACLE database is identified using its ROWID. ROWID is assigned automatically when a row is inserted in a table and removed automatically when a row is deleted from a table.

ORACLE includes the extended ROWID, which is the 10-byte in the length. A restricted ROWID (6 bytes) is also supported in ORACLE 8 for backward compatibility. Each row in the database has a unique address. One can see the row address by querying the pseudo-column ROWID. For each row in the database, the ROWID pseudo-column returns a row's address. The address is representing in the hexadecimal string.

The ROWID values are represented as:

```
Object.block.row.file
```

where:

- **Object -** The data object number. This identifies the segment the row is in.
- **block -** Data block of the data file containing the row. The length of the block depends on operating system.
- **row -** The row in the data block. The first row in the block always starts with number 0.
- **file -** The database file containing the row. The first data file always has the number 1.

The ROWID is used to locate the

- Data block in the data file
- Row in the data block (always the first row is 0)
- Data file (always the first file is 1)

ROWID values can be used for the following purpose:

- They are the fastest means of accessing a single row.
- They can show us how a table's rows are stored.
- They are unique identifiers for rows in a table.

The ROWID value does not change during the lifetime of its row. You can not update this value.

Using Data Types - Examples

Few table definitions which make use of some of the data types we have discussed are as follows..

Example 1 : The table definition for 'dept' table.

```
SQL> CREATE TABLE DEPT (
    DEPTNO  NUMBER(2) NOT NULL,
    DNAME VARCHAR2(15),
    LOC VARCHAR2(15),
    CONSTRAINT DEPT_PRIMARY_KEY PRIMARY KEY
(DEPTNO));
```


Note : The create table command is a Data Definition Language command and is discussed in the chapter 'Data Definition Language'. The example here just shows the use of data types in the column definitions.

Example 1 : The table definition for 'emp' table.

```
SQL> CREATE TABLE EMP (
    EMPNO NUMBER(4) NOT NULL,
    ENAME VARCHAR2(10),
    JOB VARCHAR2(9),
    MGR NUMBER(4) CONSTRAINT EMP_SELF_KEY
```

```
    REFERENCES  EMP(EMPNO),
    HIREDATE  DATE,
    SAL NUMBER(7,2),
    COMM  NUMBER(7,2),
    DEPTNO  NUMBER(2) NOT NULL,
    CONSTRAINT  EMP_FOREIGN_KEY FOREIGN KEY
(DEPTNO)
    REFERENCES DEPT(DEPTNO),
    CONSTRAINT EMP_PRIMARY_KEY PRIMARY KEY
(EMPNO));
```

Data Conversion

While handling various types of data, one needs to convert the data of one type to another or vice versa. The implicit and explicit data conversions are discussed in this section.

Generally an expression cannot contain values of different data types. For example, an expression cannot multiply 5 by 10 and then add 'PETER' to it. ORACLE supports both implicit and explicit conversion of values from one data type to another.

Implicit Data Conversion

ORACLE automatically internally converts a value from one data type to another when such a conversion occurs. For example, the literal string '10' has data type CHAR; ORACLE internally converts it to the NUMBER data type if it appears in a numeric expression.

In the SELECT statement, conditions can also contain values of different data types. In such cases, ORACLE converts values from one data type to another. In the following query, ORACLE implicitly converts '7936' to 7936

```
SQL> SELECT ename
    FROM emp
    WHERE empno = '7936';
```

In the following query, ORACLE implicitly converts '12-MAR-1993' to a DATE value using the default date format 'DD-MON-YYYY'

```
SQL> SELECT ename
    FROM emp
```

```
    WHERE hiredate = '12-MAR-1993';
```

Explicit Data Conversion

One can also explicitly specify data type conversions using SQL conversion functions. The following table shows common SQL functions that explicitly convert a value from one data type to another.

To From	CHAR	NUMBER	DATE	RAW	ROWID
CHAR	--	TO_NUMBER	TO_DATE	HEXTORAW	CHARTOROWID
NUMBER	TO_CHAR	--	TO_DATE	--	--
DATE	TO_CHAR	TO_CHAR	--	--	--
RAW	RAWTOHEX	--	--	--	--
ROWID	ROWIDTOCHAR	--	--	--	--

Example :

In the following query, we use the TO_NUMBER function to convert a string '7936' to a number 7936.

```
SQL> SELECT ename
    FROM emp
    WHERE empno = TO_NUMBER('7936');
```

Summary

- In a database table, various types of information is stored. The data or the information could be in the form of numbers, text or character strings, date and time data.
- For handling various types of information efficiently and conveniently, the data is stored in a database using appropriate data type.
- The data type of a column describes the type of the information, the valid data range, specific format and constraints.
- Various data types supported in ORACLE are Character Data Types, Number Data Types, Binary Data Types, Date Data Type, LOB Data Type and ROWID
- ORACLE supports both implicit and explicit conversion of values from one data type to another.
- ORACLE automatically internally converts a value from one data type to another data type when such a conversion occurs.
- One can also explicitly specify data type conversions using SQL conversion functions.

Chapter 3

Data Retrieval Using SQL

Objectives

At the end of this chapter, student will be able to:

- Identify the components of SQL.
- Explain and perform the data retrieval using Structured Query Language commands.
- Describe the use of various clauses in the SELECT statement.
- List and use various column functions in SQL.

Introduction

SQL (Structured Query Language) is a language used to access data from the database. The SQL is used for doing various database operations like creating database objects, manipulating data in the database and controlling access to the data.

SQL is made of three sub-languages. These sub-languages are the Data Definition Language (DDL), the Data Manipulation Language (DML) and Data Control Language (DCL).

The Data Manipulation Language (DML) is used for querying, inserting, updating and deleting the data stored in the database. This includes commands like SELECT, INSERT, UPDATE. In this chapter data retrieval commands and functions supported by SQL are discussed.

Introduction to SQL

SQL, the Structured Query Language is a language used to access data from the database. It is pronounced as ‘SeeQuel’ or also as ‘EsQueEl’. In 1986, the ANSI made SQL as the standard for all RDBMS.

SQL was developed for the prototype RDBMS, ‘System R’, by IBM

in mid 1970s. ORACLE Incorporation was first to introduce the commercially available implementation of SQL. It is a powerful query language and all the application development tools that ORACLE provides are SQL based.

SQL is a non-procedural language since it only specifies the task to be done and not how it is to be done. SQL processes a set of records rather than a record at a time.

As far as the database operations are concerned, SQL is efficient and convenient to use. But there are certain limitations of SQL. The limitations of the SQL include the declaration and use of variables and constants, support programming constructs to control the flow of execution.

SQL Features

Some of the important features of SQL are

- It's an interactive query language. It allows the user to retrieve the data and display it.
- It is a database programming language that allows programmers to access the data in the database.
- It is a database administration language that defines the structure of the database and controls access to the data in the database.
- It is a client/server language that allows communication between applications on client machines and the database server

The Components of SQL

SQL is made of three sub-languages. These sub-languages are as follows:

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Control Language (DCL)

Data Definition Language

The Data Definition Language consists of a set of commands used to create the database objects such as tables, views and indexes. This includes commands like CREATE, ALTER and DROP.

Data Manipulation Language

The Data Manipulation Language is used for query, insertion, updating and deletion of data stored in the database. This includes commands like SELECT, INSERT, UPDATE and MERGE.

Data Control Language

The Data Control Language is used for controlling access to the data. This includes commands like GRANT, REVOKE.

In this chapter we start our discussion on SQL with the Data Manipulation Language.

Sample Tables

The sample tables that are used in this documentation are ‘Emp’ and ‘Dept’ tables, which come with ORACLE. Most of the examples in this documentation refer to these tables. The contents of these tables are as follows:

The Dept tables store the information about departments. The description and contents of the ‘Dept’ table are as follows:

‘Dept’ table – Description

Column Name	Description
DEPTNO	Department Number
DNAME	Department Name
LOC	Department Location

‘Dept’ table – Contents

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

The ‘emp’ table stores the details of employees. The description and contents of the ‘emp’ table are as follows.

‘Emp’ Table –Description

Column Name	Description
EMPNO	Employee Number
ENAME	Employee Name
JOB	Designation
MGR	Respective Manager’s EMPNO
HIREDATE	Date of Joining
SAL	Basic Salary
COMM	Commission
DEPTNO	Department Number

‘Emp’ Table – Contents

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	JONE	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	15000		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

Data Retrieval using SQL

The Data Manipulation Language (DML) is used for querying, inserting, updating and deleting the data stored in the database. This includes commands like SELECT, INSERT, UPDATE and MERGE. The DML can further be categorised as

- Commands used to retrieve the data
- Commands used to modify the data

In this chapter, the rest of our discussion will cover the first set of commands i.e. commands used to retrieve the data.

The SELECT Statement

The SELECT Statement can be used to retrieve the data from a database table. The SELECT Statement is the most important SQL statement. To select data from the table, the table must be in our schema or we must have SELECT privilege on that table. The privileges decide the type of access to the data. The SELECT privilege on a table enables the user to retrieve the information.

The SELECT Statement has many options. The simplest form of the SELECT Statement is

```
SELECT <column-list>
FROM <table-name>
[WHERE <condition>]
[GROUP BY <column-name(s)]
[HAVING <condition>]
[ORDER BY <expression>];
```


Where,

- **Column-list -** Specifies the columns to be selected. One or more columns may be selected. If we are selecting more than one column, commas must be used to separate the column names.
- **Table-name -** Specifies the table from which columns are to be selected.
- **Condition -** Specifies the condition to be met by the row for its selection. There can be more than one condition also.

The Select statement can be used to display

- Some or all the columns.
- Some or all of the rows.
- Calculated values from the table.
- Statistical information, like averages or sums of column values.

For example, to list all the details (all columns) of all the

employees the following query can be used.

Query

```
SQL> SELECT * FROM emp;
```

The SELECT statement in the above query uses the ' * ' symbol instead of column name list, to display the values from all the rows in all the columns. The columns are displayed in the order in which they are defined in the table definition.

For selecting partial details (few columns), the following query can be used

Query

```
SQL > SELECT ename, empno, mgr, deptno
      FROM emp;
```

The SELECT statement in the above query displays the values from all the rows for the listed columns in the order specified.

Points to Remember

- The SELECT and FROM clauses are required for any SQL query.
- Columns are displayed left to right in the order specified in the SELECT column list.
- In the SELECT column list, items must be separated by commas.
- The Select statement returns rows in arbitrary manner.
- Users may query only tables they have created or tables to which they have access.
- We can suppress the repeating values using DISTINCT clause with column name.
- To see the tables belonging to the current user account, query the ORACLE table TAB as

Query

```
SQL> SELECT * FROM TAB;
```

Using alias for column names

When displaying the result of a query, SQL*PLUS normally uses the selected column's name as the column heading. These column names may however be short and cryptic. The column names can be changed for better understanding of the query result by using an alias or substitute name.

For Example, the following query on dept table changes the column name dname to department in the result.

Query

```
SQL> SELECT deptno, dname DEPARTMENT
   FROM dept;
```

Result

DEPTNO	DEPARTMENT
10	ACCOUNTING
20	RESEARCH
30	SALES
40	OPERATIONS

Suppressing the repeating values

Consider the following query, which lists the jobs.

Query

```
SQL> SELECT job FROM emp;
```

Result

JOB
CLERK
SALESMAN
SALESMAN
MANAGER

SALESMAN
MANAGER
MANAGER
SALESMAN
CLERK
CLERK
ANALYST
CLERK
PRESIDENT
ANALYST

There are total 14 rows available in emp table. If we notice, the query returns all the jobs in emp table. Here some of the jobs have been displayed more than once or they are duplicate. To eliminate the duplicate rows, we can use the DISTINCT clause in the SELECT query as follows:

Query

```
SQL> SELECT DISTINCT job
     FROM emp;
```

Result

JOB
ANALYST
CLERK
MANAGER
PRESIDENT
SALESMAN

The WHERE Clause

Using SELECT statement, we can select specific rows from a table that satisfy singular or multiple search conditions. To achieve this

objective, we can use the WHERE clause.

For example, if we want to know the details of employees in department number 20 in the table emp, then we include the WHERE clause in the SELECT statement.

With WHERE clause, the SELECT statement retrieves those rows that meet the search conditions. The WHERE clause (if used) follows the FROM clause.

The general syntax of the statement is as follows:

```
SELECT <column-list>
FROM <table-name>
[ WHERE <condition> ];
```

For example, to retrieve the employee number, employee name and job information for all the employees in the department 20, we can use the following query.

Query

```
SQL> SELECT empno, ename, job
    FROM emp
    WHERE deptno = 20;
```

Result

EMPNO	ENAME	JOB
7369	SMITH	CLERK
7566	JONES	MANAGER
7788	SCOTT	ANALYST
7876	ADAMS	CLERK
7902	FORD	ANALYST

The above query compares the DEPTNO value of each row of the table with the value 20 and then displays the rows that satisfy the condition.

In a WHERE clause if character or date constant is used, it has to be enclosed with in single quotes as shown in the example below.

Example

List the name, manager number and department number information of all the managers from emp table.

Query

```
SQL> SELECT ename, mgr, deptno
     FROM emp
     WHERE job = 'MANAGER';
```

Result

ENAME	MGR	DEPTNO
JONES	7839	20
BLAKE	7839	30
CLARK	7839	10

Points to Remember

- Columns used in the WHERE clause must be part of the table specified in the FROM clause.
- Columns used in the WHERE clause do not have to be in the SELECT column list.

Operators

The SELECT statement may have various clauses like WHERE, HAVING and others that specify condition(s). The operators used to specify conditions are categorised as follows

- Relational Operators
- Logical Operators
- Special Operators

Relational Operators

- Relational Operators are

=	Equal to
>	Greater than

<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>, !=, ^=	Not equal to

The relational operators are group of sub-expressions or comparison operators, which is evaluated to a single value TRUE, FALSE or UNKNOWN.

For example, to display a list of employees whose employee number is greater than their manager's employee number, use the following query :

Query

```
SQL> SELECT empno, ename, mgr
   FROM emp
   WHERE empno>mgr;
```

Result

EMPNO	ENAME	MGR
7788	SCOTT	7566
7844	TURNER	7698
7876	ADAMS	7788
7900	JAMES	7698
7902	FORD	7566
7934	MILLER	7782

Logical Operators

The conditions are logical expressions, which return either TRUE or FALSE for each row, based on which the row is included or excluded in the result of the query. These are simple logical expressions. One can have compound logical expressions using logical operators (AND or OR operators). These operators are

AND	Logical AND
OR	Logical OR

NOT	Logical NOT

The use of these operators is shown in following examples.

Example 1

Find out the employee number, name, joining date and department number of all the managers in the organization that joined the organization on or after 1st January 1980.

This can be done using the following query.

Query

```
SQL> SELECT empno, ename, hiredate, deptno
    FROM emp
    WHERE   job = 'MANAGER'
    AND HIREDATE >= '01-JAN-80';
```

Result

EMPNO	ENAME	HIREDATE	DEPTNO
7566	JONES	02-APR-81	20
7698	BLAKE	01-MAY-81	30
7782	CLARK	09-JUN-81	10

Example 2

The following query lists all the employees whose job is SALESMAN and salary is greater than 1500.

Query

```
SQL> SELECT empno, ename, job, sal
    FROM emp
    WHERE job = 'SALESMAN' AND sal > 1500;
```

Result

EMPNO	ENAME	JOB	SAL
7499	ALLEN	SALESMAN	1600

Special Operators

The Special operators are used to specify special conditions or to ease the compound expressions.

For example, the BETWEEN operator is used to select the values that are within the range of values and NOT BETWEEN operator is used to select the values that are outside the range of values.

The following are the Special Operators:

Operator	Use
BETWEEN	Checking value within a range
NOT BETWEEN	Checking value outside a range
IN	Checking value in a set
IS NULL	Checking for Null value
LIKE	Matching pattern from a column

BETWEEN … AND … Operator

This operator is used to select the values that are within or outside a range of values.

The use of BETWEEN ... AND ... operator to include a range is shown in the following syntax.

```
SELECT column-list
FROM <table-name>
WHERE column BETWEEN min_value AND max_value;
```

For example, to display the list of all the employees whose salary is in the range 1200 to 2000, use the following query

Query

```
SQL> SELECT ename,sal
   FROM emp
   WHERE sal BETWEEN 1200 AND 2000;
```

Result

ENAME	SAL
ALLEN	1600

WARD	1250
MARTIN	1250
TURNER	1500
MILLER	1300

To exclude the range we can use BETWEEN ... AND ... Operator with NOT using the following syntax.

```
 SELECT column-list
 FROM <table-name>
 WHERE column NOT BETWEEN min_value AND
max_value;
```

Here the NOT BETWEEN returns the rows outside the range.

For example, if we want to know the names, jobs and joining dates of employees who did not join the organization between 1st January 1981 and 1st January 1982, then enter the query as

Query

```
SQL> SELECT ename, job, hiredate
   FROM emp
   WHERE hiredate NOT BETWEEN '01-JAN-81'
   AND '01-JAN-82';
```


Result

ENAME	JOB	HIREDATE
SMITH	CLERK	17-DEC-80
SCOTT	ANALYST	19-APR-87
ADAMS	CLERK	23-MAY-87
MILLER	CLERK	23-JAN-82

IN Operator

This operator can be used to select rows that match one of the values included in the list.

For example, if we want to find out the employees who are ANALYST, SALESMAN, then use the following query

Query

```
SQL> SELECT ename
   FROM emp
   WHERE job IN ('ANALYST','SALESMAN');
```

Result

ENAME

ALLEN

WARD

MARTIN

SCOTT

TURNER

FORD

We can use the NOT IN operator to select only those rows that have a value not in the list. This is shown in the following query.

Query

```
SQL> SELECT ename
   FROM emp
   WHERE job NOT IN ('ANALYST','SALESMAN');
```

Result

ENAME

SMITH

JONES

BLAKE

CLARK

KING

ADAMS

JAMES

MILLER

IS NULL Operator

This operator is used to compare the value in the column with NULL and return the row accordingly.

For example, To find out the employees whose commission is null, use the following query.

Query

```
SQL> SELECT ename, sal, comm
    FROM emp
    WHERE comm IS NULL;
```

Result

ENAME	SAL	COMM
SMITH	800	
JONES	2975	
BLAKE	2850	
CLARK	2450	
SCOTT	3000	
KING	5000	
ADAMS	1100	
JAMES	1950	
FORD	3000	
MILLER	1300	

LIKE Operator

Like operator is used for pattern searching with character columns. The Pattern consists of characters to be matched and the 'wildcard' characters as listed below.

Wildcard Character	Matches
_ (Underscore)	Any single character
%	Any sequence of zero or more characters

The general syntax is

```
SELECT column-list
FROM table-name
WHERE column LIKE 'pattern';
```

Or

```
WHERE column NOT LIKE 'pattern';
```

The use of LIKE operator is shown in the examples below.

Example 1

List the employee names that end in 'S'.

Query

```
SQL> SELECT ename
     FROM emp
     WHERE ename LIKE '%S';
```

Result

ENAME
JONES
ADAMS
JAMES

Example 2

List the employee names that begin with 'S'.

Query

```
SQL> SELECT ename
   FROM emp
   WHERE ename LIKE 'S%';
```

Result

ENAME
SMITH
SCOTT

Example 3

List the employee names with the third character as ‘R’.

Query

```
SQL> SELECT ename
   FROM emp
   WHERE ename LIKE '_ _R%';
```

Result

ENAME
WARD
MARTIN
TURNER
FORD

Example 4

List the employee names having exactly five characters.

Query

```
SQL> SELECT ename
   FROM emp
   WHERE ename LIKE '_ _ _ _ _';
```

Result

ENAME
SMITH
ALLEN
JONES
BLAKE
CLARK
SCOTT
ADAMS
JAMES

The ORDER BY Clause

The SELECT statement returns rows in an arbitrary manner. For getting the resultant rows in a specified order, the ORDER BY clause is used in SQL statements.

The general syntax is

```
SELECT column-list
FROM table-name
[WHERE condition]
ORDER BY column or expression [ASC/DESC];
```

The use of ORDER BY clause is shown in following example.

Example

List the employee name and the salary for all the employees in the department 30 in the ascending order of their salary.

This can be done using the following query.

Query

```
SQL>  SELECT ename, sal
    FROM emp
    WHERE deptno = 30
    ORDER BY sal;
```

Result

ENAME	SAL
JAMES	950
WARD	1250
MARTIN	1250
TURNER	1500
ALLEN	1600
BLAKE	2850

The default order is ascending, but using the keywords ASC/DESC can explicitly specify the ordering. We can specify the order on more than one column as well. For example, the following query specifies ordering on two columns.

Query

```
SQL> SELECT ename, sal, job
   FROM emp
   WHERE deptno = 30
   ORDER BY sal, ename DESC;
```

Result

ENAME	SAL	JOB
JAMES	950	CLERK
WARD	1250	SALESMAN
MARTIN	1250	SALESMAN
TURNER	1500	SALESMAN
ALLEN	1600	SALESMAN
BLAKE	2850	MANAGER

The ORDER BY clause may include integers that represent the relative position of a column in the SELECT list. This is shown in the following query.

Query

```
SQL> SELECT ename, sal, job
   FROM emp
   WHERE deptno = 30
   ORDER BY 3,1 DESC;
```

Result

ENAME	SAL	JOB
JAMES	950	CLERK
BLAKE	2850	MANAGER
WARD	1250	SALESMAN
TURNER	1500	SALESMAN
MARTIN	1250	SALESMAN
ALLEN	1600	SALESMAN

Points to Remember

- The ORDER BY clause must be the last clause.
- The ORDER BY clause is the only way to retrieve rows in a specific order.
- The Columns that are not in the SELECT list may still appear in the ORDER BY clause.

The GROUP BY Clause

Many times it is required to group the rows in a table based on certain criteria. The GROUP BY clause can be used for this. The grouping criterion is specified in the form of an expression. ORACLE groups rows based on this expression and then returns a single row of the summary information for each group.

For example,

If we want to know the average salary of the employees in each department, then use the following query

Query

```
SQL> SELECT deptno, AVG(sal)
   FROM emp
   GROUP BY deptno;
```

Result

DEPTNO	AVG(SAL)
10	2916.6667
20	2175
30	1566.6667

A query that includes the GROUP BY clause is called a grouped query, because it groups the data from its source tables and produces a single summary row for each row group. The columns named in the GROUP BY clause are called the grouping columns of the query.

If the SQL command does not contain WHERE clause then place GROUP BY clause after FROM clause and if the SQL command contain the WHERE clause then place the GROUP BY clause after the WHERE clause.

For example, to find out the average annual salary of the non-managerial staff in each department, use the following query.

Query

```
SQL> SELECT deptno, 12*AVG(sal)
   FROM emp
   WHERE job NOT IN ('MANAGER')
   GROUP BY deptno;
```

Result

DEPTNO	12*AVG(SAL)
10	37800
20	23700
30	15720

The HAVING Clause

Just as we can select specific rows with WHERE clause, we can select specific groups with a HAVING clause. The HAVING clause works very much like WHERE clause, except that its logic is only related to the results of group functions.

We may include both WHERE as well as HAVING clauses in a query. The way SQL*PLUS processes the query is as follows

- It applies the WHERE clause to select rows
- It forms the groups and calculates group functions
- It applies HAVING clause to select group

The use of HAVING clause to determine which departments have more than two people holding a particular job is shown in the following query.

Query

```
SQL> SELECT deptno, job, count(*)
   FROM emp
   GROUP BY deptno, job
   HAVING count(*) > 2;
```

Result

DEPTNO	JOB	COUNT(*)
30	SALESMAN	4

We can use both the WHERE clause and the HAVING clause in a SELECT statement. The following example shows the combination of WHERE and HAVING clauses.

Example

List the departments with more than two salesmen.

Query

```
SQL> SELECT deptno
   FROM emp
   WHERE job = 'SALESMAN'
   GROUP BY deptno
   HAVING count(*) > 2;
```

Note

You can use aggregate /Group functions in having clause.

Result

DEPTNO
30

Column Functions

The Column functions are a set of functions that can be used with values from database tables. The column functions can be categorised as

- Aggregate / Group Function
- Arithmetic Functions
- Character Functions
- Date Functions

Aggregate / Group Function

These functions act on a group or set of rows and return one row of summary per set of rows.

Function	Description
SUM	Computes the total value of the group
AVG	Computes the average value of the group
MIN	Computes the minimum value of the group
MAX	Computes the maximum value of the group
STDDEV	Computes the standard deviation of the group
VARIANCE	Computes the variance of the group
COUNT	Counts the number of non-NULL values for the specified group
COUNT(*)	Counts the number of rows including those having NULL values for the given condition

Examples on using column functions :

SUM Function

Example 1

List the total salary of all the employees.

Query

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

Result

SUM(SAL)
29025

Here, GROUP BY clause is not specified, the whole table is assumed as one group

Example 2

List the total salary of all employees department wise.

Query

```
SQL> SELECT deptno, SUM(sal)
   FROM emp
   GROUP BY deptno;
```

Result

DEPTNO	SUM(SAL)
10	8750
20	10875
30	9400

MIN and MAX Functions

Example

List the minimum and maximum salary paid in the company.

Query

```
SQL> SELECT MIN(sal), MAX(sal)
   FROM emp;
```

Result

MIN(SAL)	MAX(SAL)
800	5000

COUNT and COUNT(*) Functions

Example 1

The following query determines the number of employees who

have job titles i.e. where the job is not NULL.

Query

```
SQL> SELECT count(job)
    FROM emp;
```

Result

COUNT(JOB)
14

Example 2

The following query returns the number of different job titles available in the company.

Query

```
SQL> SELECT count(DISTINCT job)
   FROM emp;
```

Result

COUNT(DISTINCTJOB)
6

Example 3

The following query returns the number of employees in each department.

Query

```
SQL>  SELECT deptno, count(*)
    FROM emp
    GROUP BY deptno;
```

Result

DEPTNO	COUNT(*)
10	3
20	5
30	6

Arithmetic Functions

We can perform certain calculations using SQL functions. Some of the arithmetic functions supported by SQL are listed here.

Function	Usage	Result
ABS	abs (n)	Absolute value of n
CEIL	ceil (n)	Returns smallest integer >= n
COS	cos (n)	Returns the cosine of n
COSH	cosh (n)	Returns the hyperbolic cosine of n
EXP	exp (n)	Returns e raised to the nth power; e = 2.712183
FLOOR	floor (n)	Returns the largest integer <= n
GREATEST	greatest(n1, n2,…)	Returns n whichever is larger
LEAST	least(n1,n2,…)	Returns n whichever is smaller
LN	ln (n)	Returns the natural logarithm of n, where n is > 0
LOG	log (m, n)	Returns the logarithm, base m, of n.
MOD	mod (m, n)	Remainder of m divided by n. If n is 0, 0 is returned.
POWER	power (m,n)	Return the value of m raised to the n the power.
ROUND	round (m,n)	Returns m rounded to n places right of the decimal point.
SIGN	sign (n)	If n > 0, function returns 1. If n = 0, it returns 0 and if n < 0 then it returns -1.
SIN	sin (n)	Returns the sine of n
SINH	sinh (n)	Returns the hyperbolic sine of n
SQRT	sqrt (n)	Returns square root of n
TAN	tan (n)	Returns the tangent of n
TANH	tanh (n)	Returns the hyperbolic tangent of n
TRUNC	trunc (m,n)	Returns m truncated to n decimal places

Let us see examples on some of the functions.

Examples

ABS Function

abs (25) = 25; abs (-25) = 25;

CEIL Function

ceil (7) = 7, ceil (3.14) = 4, ceil (-3.14) = - 3

FLOOR Function

floor (7) = 7, floor (3.14) = 3, floor (-3.14) = - 4

MOD Function

mod (9, 2) = 1, mod (25,5) = 0, mod (23,0) = 0, mod (36,78) = 36

POWER Function

power (4,2) = 16, power (-3,3) = -27

SQRT Function

sqrt (100) = 10, sqrt (-25) = NULL

Square root of a negative number is an imaginary number. ORACLE doesn't support imaginary numbers hence NULL is returned.

ROUND and TRUNC Functions

These two are related functions. ROUND rounds numbers to a given number of digits of precision; TRUNC truncates or chops off digits of precision from a number.

Amount	ROUND (amount , 2)	TRUNC (amount, 2)
130.435	130.44	130.43
108.697	108.7	108.69
256.173	256.17	256.17

Character Functions

The character function is used with character or string columns. The following table shows the string functions available in SQL.

Function Name	Use
CONCAT or || (concatenation operator)	Concatenates two strings together. The symbol '||' is called a broken vertical bar. This can be used while concatenating more than two strings.
INITCAP	INITial CAPital. Changes the first letter of a word or series of words into uppercase
INSTR	Finds the location of a character IN a STRing.
LENGTH	Finds the LENGTH of a string
LOWER	Converts every letter in a string to LOWER case
LPAD	Left PAD. Makes a string of a certain length by adding a certain set of characters to the left
LTRIM	Left TRIM. Trims all the occurrences of any one of a set of characters off of the left side of a string.
REPLACE	Replace allows for substitution of one string for another in a given character string.
RPAD	Right PAD. Makes a string of a certain length by adding a certain set of characters to the right.
RTRIM	Right TRIM. Trims all the occurrences of any one of a set of characters, off of the right side of a string.
SOUNDX	Finds words that SOUND alike
TRANSLATE	Translate allows for substitution of characters from one string by the corresponding characters in another string.
UPPER	Converts every letter in a string into UPPER case
SUBSTR	Finds a part of the given string

Let us see some examples based on few of these functions.

CONCAT Function or Concatenation operator (||)

The concat function tells ORACLE to concatenate, or stick together, two strings. The string can be either column names or literal. Concatenation operator (||) can be used when more than two strings needs to be concatenated. The use is shown in the following query.

Query

```
SQL> SELECT ename || '-' || job "Employee Job"
   FROM emp;
```

Result

Employee Job

SMITH-CLERK

ALLEN-SALESMAN

WARD-SALESMAN

JONES-MANAGER

MARTIN-SALESMAN

BLAKE-MANAGER

CLARK-MANAGER

SCOTT-ANALYST

KING-PRESIDENT

TURNER-SALESMAN

ADAMS-CLERK

JAMES-CLERK

FORD-ANALYST

MILLER-CLERK

UPPER, LOWER & INITCAP Functions

The use of these functions is shown in the following query.

Query

```
SQL> SELECT INITCAP(ename) NAME, LOWER(job)
   FROM emp
   WHERE UPPER(ename) = 'SCOTT' ;
```

Result

NAME	LOWER(JOB)
Scott	analyst

RPAD and LPAD Functions

The general format of these functions is as follows..

```
Rpad ( string, length [, 'set'])
Lpad ( string, length [, 'set'])
```

Where,

- **String -** is the name of the character column from the database or literal string;
- **length** - is the total number of characters long that the result should be, and set is the set of characters that do the padding. The set must be enclosed in the apostrophes (single quotes).

The use of these functions is shown in the following query.

Query

```
SQL> SELECT RPAD (ename,8,'*')
   FROM emp;
```

Result

RPAD(ENA

SMITH***

ALLEN***

WARD****

JONES***

MARTIN**

BLAKE***

CLARK***

SCOTT***

KING****

TURNER**

ADAMS***

JAMES***

FORD****

MILLER**

RTRIM and LTRIM Functions

This function trims off unwanted characters from right and left edges of string respectively. The general format is

```
RTRIM(string [,'set'])
LTRIM(string [, 'set'])
```

Where,

- **String -** is the name of the column from the database or a literal string and set is the collection of characters that we want to trim off. The use of these functions is shown in the following query.

Query

```
SQL> SELECT dname, RTRIM(dname, 'NG' ) shortname
   FROM dept;
```

Result

DNAME	SHORTNAME
ACCOUNTING	ACCOUNTI
RESEARCH	RESEARCH
SALES	SALES
OPERATIONS	OPERATIONS

LENGTH Function

The LENGTH('ACCOUNTS') will return 8.

INSTR Function

The general syntax is

```
INSTR ( STRING , SET [, START [ , OCCURANCE]])
```

This function searches in the string for a certain set of characters.

The result of using INSTR function is shown below.

```
INSTR('NEWYORK','W') will return 3
INSTR('CHICAGO','C',2) will return 4
```

SUBSTR Function

The general syntax is

```
SUBSTR (STRING, m [, n])
```

This function returns part of the given string starting at m^{th} position and next n characters.

The result of using SUBSTR function is shown below.

```
SUBSTR('NEWYORK',4,3) will return 'YOR'
SUBSTR('CHICAGO',1,4) will return 'CHIC'
```

SOUNDEX

This function is used almost exclusively in a WHERE clause. It has the unusual ability to find words that sounds like another word, virtually regardless of how either is spelt. This is useful when we are not certain how the word is spelt.

The use of SOUNDEX is shown in the following query.

Query

```
SQL> SELECT ename
   FROM emp
     WHERE soundex(ename) = soundex('SCOT');
```

Result

ENAME

SCOTT

Date Functions

DATE is an ORACLE data type and it recognizes columns that are of the DATE data type. We can use the date functions specifically

for doing date arithmetic. The List of functions ORACLE uses to operate on the DATE columns is as follows

Function	Usage	Result
ADD_MONTHS	add_months(d, n)	Adds n months to date d. n can be negative
GREATEST	greatest(d1,d2...)	Picks latest date from the list of dates
LEAST	least(d1,d2...)	Picks earliest date from the list of dates
LAST_DAY	last_day (d)	Returns the date of the last day of the month containing d. Useful in determining number of days left in a given month.
MONTHS_BETWEEN	months_between(d,e)	Returns number of months between d and e.
NEW_TIME	new_time(d, a, b)	Returns date and time zone in b, when date and time zone in time zone a are d.
NEXT_DAY	next_day(d, char)	Returns date of first day of the week named by char that is later than d. Char must be a valid day of the week.
ROUND	round (d [, format])	Returns d rounded as specified by the rounding unit format, which defaults to nearest day.
SYSDATE	sysdate	Returns the current date & time.
NEW_TIME	NEW_TIME(d, a,b)	Returns date and time for time zone b, when d (date & time) is given for time zone a. a & b are three letter abbreviation fro the time zones. Abbreviations for time zones

		are as follows:
	AST/ADT BST/BDT CST/CDT EST/EDT GMT HST/HDT MST/MDT NST PST/PDT YST/YDT	Atlantic standard/daylight time Bering standard/daylight time Central standard/daylight time Eastern standard/daylight time Greenwich mean time Alaska-Hawaii standard/daylight time Mountain standard/daylight time Newfoundland standard time Pacific standard/daylight time Yukon standard/daylight time
TO_CHAR	TO_CHAR(d,'format')	Reformats date according to 'format'.
TO_DATE	TO_DATE(s,'format')	Converts s in a given 'format' into an Oracle date.

Some of the examples based on date functions are listed here

ADD_MONTHS Function

Add_months (To_date('23-0ct-86'), 12) will return date 23-Oct-87.

Add_months (To_date('23-0ct-86'), -2) will return date 23-Aug-86.

LAST_DAY Function

Last_day(To_date('20-Mar-88')) will return 31-MAR-88.

MONTHS_BETWEEN Function

Months_between (To_date('27-Mar-88'), To_date('5-May-88')) will return -1.29 .

Months_between (To_date('27-Aug-88'), To_date('5-May-88')) will return 3.71 .

GREATEST and LEAST Functions

Unlike many other Oracle functions and logical operators, the GREATEST and LEAST functions will not evaluate literal strings that are in date format as dates. In order for LEAST and GREATEST to work properly, the function TO_DATE must be applied to the literal strings:

Example

```
SQL> SELECT LEAST(TO_DATE('16-JAN-98'),
TO_DATE('22-FEB-98')) FROM DUAL;
```

Dual table

DUAL is a small but useful Oracle table created for testing functions or doing quick calculations. The actual column in DUAL is irrelevant. This means that you can experiment with date formatting and arithmetic using the DUAL table and the date functions in order to understand how they work.

Date formats

Following date formats are used with both TO_CHAR and TO_DATE :

Abbreviation	Description	Example (Output)
MM	Number of month	10
RM	Roman numeral month	XI
MON	Three-letter abbreviation of Month	FEB
MONTH	Month fully spelled out	FEBRUARY
DDD	Number of days in year, since Jan 1	245
DD	Number of days in month	23
D	Number of days upto to given date in respective week. (Sunday is 1st day of week and Saturday is 7th day of week.)	4
DY	Three-letter abbreviation of day.	FRI
DAY	Day fully spelled out.	FRIDAY

YYYY	Full four digit year.	1999
SYYYY	Signed year.	1000 B.C. = -1000
YYY	Last three digits of year.	999
YY	Last two digits of year.	99
Y	Last one digit of year.	9
YEAR	Year spelled out	NINETEEN NINETY-NINE
Q	Number of quarter	3
WW	Number of weeks in year since Jan 1.	34
W	Number of weeks in month	3
J	“Julian” – days since December 31, 4713 B.C.	2451508
HH or HH12	Hours of day, always 1-12	9
HH24	Hours of day, 24-hour clock	22
MI	Minutes of hour	46
SS	Seconds of minute	33
SSSS	Seconds since midnight, always 0-86399	33000
/ , : .	Punctuation to be incorporated in display for TO_CHAR or ignored in format for TO_DATE	10/12/2002 10:30:35
A.M. or P.M.	Displays A.M. or P.M., depending time of day	A.M. or P.M.
AM or PM	Same as above, but without periods.	AM or PM
B.C. or A.D.	Displays B.C. or A.D., depending upon date	B.C. or A.D.
BC or AD	Same as above, but without periods.	BC or AD

Following examples show use of some of above abbreviations:

```
SQL> SELECT TO_CHAR(SYSDATE, 'YYYY' ) FROM DUAL;
SQL> SELECT TO_CHAR(SYSDATE, 'Q' ) FROM DUAL;
SQL> SELECT TO_CHAR(TO_DATE('12-JAN-1999'), 'W'
) FROM DUAL;
SQL> SELECT TO_CHAR(TO_DATE('12-JAN-1999'),
'B.C.' ) FROM DUAL;
```

Following date formats work only with TO_CHAR. They do not work with TO_DATE.

Abbreviation	**Description**	**Example (Output)**
'string'	String is incorporated in display for TO_CHAR	This is JANUARY
Fm	Prefix to Month or Day: fmMONTH or fmday. Suppresses padding of Month or Day (defined earlier) in format. Without fm, all months are displayed at same width. Similarly true for days. With fm padding is eliminated.Months and days are only as long as their count of characters.	
TH	Suffix to a number: ddTH or DDTH. Capitalisation comes from the case of the number – DD – not from the case of the TH. Works with any number in a date: YYYY, DD, MM, HH, MI, SS, and so on.	24th or 24TH
SP	Suffix to a number that forces number to be spelled out: DDSP, DdSP, or ddSP produces THREE,Three, or three.	

	Capitalisation comes from case of number – DD – not from the case of SP. Works with any number in a date: YYYY, DD, MM, HH, MI, SS, and so on.	
SPTH	Suffix combination of TH and SP that forces number to be both spelled out and given an ordinal suffix: Ddspth produces Third. Capitalisation comes from case of number – DD – not from the case of SP. Works with any number in a date: YYYY, DD, MM, HH, MI, SS, and so on.	
THSP	Same as SPTH	

Following examples show use of some of above abbreviations:

```
SQL> SELECT TO_CHAR(TO_DATE('12-JAN-1999'),
'"This is" MONTH ' )
    FROM DUAL;
SQL> SELECT TO_CHAR(TO_DATE('12-JAN-1999'),
'"This is" fmMONTH ' )
    FROM DUAL;
SQL> SELECT TO_CHAR(TO_DATE('01-JAN-01'),
'MONTH, DDth YYYY HH:MI:SS')
    FROM DUAL;
SQL> SELECT TO_CHAR(TO_DATE('01-JAN-01'),
'MONTH, DDSPth YYYY HH:MI:SS')
    FROM DUAL;
```


CURRENT_DATE Functions

Returns current date in session's time zone, having DATE data type.

CURRENT_TIMESTAMP Function

Returns current date in session's time zone, having TIMESTAMP WITH TIME ZONE data type with specified precision.

```
SQL> SELECT CURRENT_TIMESTAMP(4) FROM dual;
```

LOCALTIMESTAMP Function

Returns current date in session's time zone, having TIMESTAMP data type with specified precision.

```
SQL> SELECT LOCALTIMESTAMP(4) FROM dual;
```

SYSTIMESTAMP Function

Returns system date including fractional seconds and time zone of the database, having TIMESTAMP WITH TIME ZONE data type with specified precision.

```
SQL> SELECT SYSTIMESTAMP(4) FROM dual;
```

EXTRACT Function

This datetime function extracts and returns the value of a specified datetime field from a datetime value expression.

The general syntax is as follows:

```
EXTRACT ( [YEAR] [MONTH] [DAY] [HOUR] [MINUTE]
[SECOND]
        FROM datetime_value )
```

Example :

```
SQL> SELECT ename, EXTRACT(MONTH FROM hiredate)
hire_month FROM emp;
```

General Functions

Handling NULL values

As we know, a data field without any value in it is said to contain NULL value. There are many situations where we are required to deal with the NULL value columns in queries or calculations.

We have already seen that in a query we use IS NULL or IS NOT NULL to select or deselect the column having NULL

values.

Applying simple arithmetic, the column having NULL value will always return a NULL value, i.e. any value + NULL is NULL, any value * NULL is NULL.

But in some situations we require to have a non-NULL value for the purpose of evaluating an expression containing a NULL value column.

For example, in calculating the total compensation for an employee as

(pay + commission)

We need to convert the NULL value in the commission column to zero before actually adding them together.

ORACLE provides the NVL function to convert the NULL value to any appropriate value we want.

The syntax for NVL function is :

```
NVL (col, value )
```

The NVL function takes two arguments. First is the name of a column in which the NULL values appear and second is the value we want to substitute for a NULL. The NVL function when encounters a non-null value, it will accept that value. When it finds a NULL it will use, instead, the value in the second argument.

Thus, the Compensation is calculated as

PAY + NVL (COMM, 0)

So, where the COMM field is NULL, the above expression returns PAY + 0 i.e. Pay.

The use of an NVL function is shown in the following query

Query

```
SQL> SELECT ename, sal, comm, sal+comm,
sal+NVL(comm,0)
    FROM emp;
```

NVL2 Function

The syntax for NVL2 function is :

```
NVL2 (col, value1, value2 )
```

The NVL2 function takes three arguments. First is the name of a column in which the NULL values appear, second is the value we want to substitute when column value is present (i.e. NOT NULL), and third is the value which is to be substituted for NULL value of column.

Query

```
SQL> SELECT comm, NVL2(comm, 'Applicable', 'Not
applicable') FROM emp;
```


NULLIF Function

The syntax for NULLIF function is :

```
NULLIF (expr1, expr2)
```

This function checks expr1 and expr2 for equality and returns NULL if equal, else returns value of expr1.

COALESCE Function

The syntax for COALESCE function is :

```
COALESCE (expr1, expr2, expr3, ….)
```

This function returns the first non-NULL value from the list of values.

DECODE Function

The general syntax is

```
Decode ( value, if1, then1, if2, then2 … else )
```

This function tests the value (which can be a column name, or any function used on the column or columns) for every row. If it equals if1, then then1 is returned by the function. If it equals if2, then2 is returned. We can construct as many if / then constructs as required. If all those fail then the value followed by else is returned.

The use of a DECODE is shown in the following query

Query

```
SQL> SELECT ename, decode(job, 'CLERK', 5,'
ANALYST', 4,2)
   FROM emp;
```

Result

ENAME	DECODE(JOB,'CLERK',5,'ANALYST',4,2)
SMITH	5
ALLEN	2
WARD	2
JONES	2
MARTIN	2
BLAKE	2
CLARK	2
SCOTT	4
KING	2
TURNER	2
ADAMS	5
JAMES	5
FORD	4
MILLER	5

CASE expression

CASE expression supports if-then-else logic within SQL statement without having to invoke procedure.

The syntax for CASE is :

```
CASE column | Expression
WHEN value1 THEN R_value1
WHEN value2 THEN R_value2
:
ELSE default
END
```

Example in DECODE function above can be re-written by using CASE as follows:

```
SQL> SELECT ename,
   CASE job
    WHEN 'CLERK' THEN 5
    WHEN 'ANALYST' THEN 4
    ELSE 2
    END
   FROM emp;
```


Translate Function

The translate function is used for a character by character substitution.

The general syntax is

```
TRANSLATE(col, char, subschar)
```

The use of TRANSLATE function is shown in the following example.

Example

translate('ORACLE', 'O', 'C') returns 'Cracle';

Expressions

In SQL queries, arithmetic computations can be done on numeric columns. An alias can be given to a column and / or an expression in a query. The use of alias makes the column heading (in the output of a query) more meaningful. If defined, alias names are displayed in place of column names. Alias names are given to the right of a column name enclosed within quotes.

For example,

Query

```
SQL> SELECT deptno, 12*AVG(sal) "SAL_AVG"
     FROM emp
     WHERE job NOT IN ('MANAGER')
```

```
    GROUP BY deptno;
```

Result

DEPTNO	SAL_AVG
10	37800
20	23700
30	15720

Summary

- SQL, the Structured Query Language is a language used to access data from the database. In 1986, the ANSI made SQL as the standard for all RDBMS. SQL is a non-procedural language since it only specifies the task to be done and not how it is to be done (unlike 3GL). It processes a set of records rather than a record at a time.
- SQL is made up of three sub-languages, namely Data Definition Language (DDL), Data Manipulation Language (DML) and Data Control Language (DCL). The Data Manipulation Language is used for query, insertion, updating and deletion of data stored in the database. This includes commands like SELECT, INSERT, UPDATE.
- The SELECT Statement can be used to retrieve the data from a database table.
- The simplest form of the SELECT Statement is

  SELECT <column-list>

  FROM <table-name>

  [WHERE <condition>]

  [GROUP BY <column-name(s)]

  [HAVING <condition>]

  [ORDER BY <expression>];

- In a SELECT statement, we can select specific rows from a table that satisfy some condition using the WHERE clause.
- The operators used to specify conditions in SQL statements, are categorised as Relational Operators, Logical Operators and Special Operators.
- For getting the resultant rows in a specified order, the ORDER BY clause is used in SQL statements.
- To group the rows in a table based on certain criteria, the GROUP BY clause can be used.
- Just as we can select specific rows with WHERE clause, we can select specific groups with a HAVING clause.
- The Column functions are a set of functions that can be used with values from database tables. The column functions can be categorised as
  - Aggregate / Group Function
  - Arithmetic Functions
  - Character Functions
  - Date Functions

Chapter 4

Multi-Table Queries

Objectives

At the end of this chapter, student will be able:

- To classify multi-table queries.
- To explain and write multi-table queries.
- To explain the use of joins, set operators and sub-queries.

Introduction

During database design, data is split into various tables. This data from various tables needs to be accessed together. Different mechanisms supported in ORACLE for this are joins, set operators and subqueries.

Joins are the foundation of multi-table query processing. The process of forming rows from two or more tables is called as joining tables. The set operators combine the results of a row or more queries. The subquery is one of the advanced query techniques in SQL and it lets us to use the results of one query as a part of another query.

In this chapter we will discuss these mechanisms and their use.

Joins

The process of forming rows from two or more tables by comparing the contents of related columns is called as joining tables. The resulting table is called a join between tables and it contains data from one or more of the original tables.

Joins are the core of multi-table query processing. All the data in a relational database is stored in its columns as explicit data values. All the possible relationships between tables can be formed by matching the contents of related columns.

SQL handles multi-table queries by matching columns. It uses the SELECT command for a multi-table query. In a join, the SELECT command must contain a search condition that specifies a column

match. To make a query, in which rows of two tables are joined, we must specify the join columns that contain corresponding information in the two tables. In a Join we specify the tables to be joined in the SELECT command's FROM clause and specify the join columns in the WHERE clause.

The general syntax is

```
SELECT columnlist
FROM table1, table2, …
WHERE logical expression;
```


Points to Remember

- The Columns from either table may be named in the SELECT clause.
- Columns which have the same name in multiple tables named in the FROM Clause must be uniquely identified by specifying tablename.columnname.
- Tablename.* is a short cut to specify all the columns in the table.
- More than one pairs of columns may be used to specify the join condition between any two tables.
- As many tables as desired may be joined together.

The process of join is discussed using the sample tables SEMP and SDEPT.

Suppose, we want to know the employee and their department details. This can be done using the following query.

SQL> SELECT Empno, Ename, SEMP.Deptno, SDEPT.Deptno, Dname, Loc

FROM SEMP, SDEPT ← table to be joined

WHERE SEMP.Deptno = SDEPT.Deptno; ← the join condition

The step carried out during the join are as follows :

SEMP		
Empno	Ename	Deptno
7369	SMITH	20
7698	BLAKE	30
7782	CLARK	10
7934	MILLER	10

SDEPT		
Deptno	Dname	Loc
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

M rows 4 rows N rows 4 rows

M × N rows i.e. 4 × 4 rows

Empno	Ename	Deptno	Deptno	Dname	Loc
7369	SMITH	20	10	ACCOUNTING	NEW YORK
7369	SMITH	20	20	RESEARCH	DALLAS
7369	SMITH	20	30	SALES	CHICAGO
7369	SMITH	20	40	OPERATIONS	BOSTON
7698	BLAKE	30	10	ACCOUNTING	NEW YORK
7698	BLAKE	30	20	RESEARCH	DALLAS
7698	BLAKE	30	30	SALES	CHICAGO
7698	BLAKE	30	40	OPERATI	BOSTON

				ONS	
7782	CLARK	10	10	ACCOUNTING	NEW YORK
7782	CLARK	10	20	RESEARCH	DALLAS
7782	CLARK	10	30	SALES	CHICAGO
7782	CLARK	10	40	OPERATIONS	BOSTON
7934	MILLER	10	10	ACCOUNTING	NEW YORK
7934	MILLER	10	20	RESEARCH	DALLAS
7934	MILLER	10	30	SALES	CHICAGO
7934	MILLER	10	40	OPERATIONS	BOSTON

Apply the join condition i.e.

SEMP.Deptno = SDEPT.Deptno

Result of the join

Empno	Ename	Deptno	Deptno	Dname	Loc
7369	SMITH	20	20	RESEARCH	DALLAS
7698	BLAKE	30	30	SALES	CHICAGO
7782	CLARK	10	10	ACCOUNTING	NEW YORK
7934	MILLER	10	10	ACCOUNTING	NEW YORK

Let us see another example on joins.

Example

Consider the query to list the employee number, employee name, job, his department name and its location.

The employee number, employee name, job information is in the emp table and dept name, its location information is in the dept table. Here we need to access this information together, so we use join. The Query is as follows

```
SQL> SELECT empno, ename, job, dname, loc FROM
emp, dept WHERE
   emp.deptno=dept.deptno;
```

The result of the query would be

Result

EMP NO	ENAME	JOB	DNAME	LOC
7782	CLARK	MANAGER	ACCOUNTING	NEW YORK
7839	KING	PRESIDENT	ACCOUNTING	NEW YORK
7934	MILLER	CLERK	ACCOUNTING	NEW YORK
7369	SMITH	CLERK	RESEARCH	DALLAS
7876	ADAMS	CLERK	RESEARCH	DALLAS
7902	FORD	ANALYST	RESEARCH	DALLAS
7788	SCOT	ANALYST	RESEARCH	DALLAS
7566	JONES	MANAGER	RESEARCH	DALLAS
7499	ALLEN	SALESMAN	SALES	CHICAGO
7698	BLAKE	MANAGER	SALES	CHICAGO
7654	MARTIN	SALESM	SALES	CHICAGO

		AN		
7900	JAME	SCLERK	SALES	CHICAGO
7844	TURNER	SALESMAN	SALES	CHICAGO
7521	WARD	SALESMAN	SALES	CHICAGO

Using aliases in Joins

When using a join we identify all the columns by using their table names. However, typing long table names can be bothersome sometimes. To overcome this we can use aliases for tables.

```
SELECT t1.col1, t1.col2, t2.col3
FROM table1 t1, table2 t2
WHERE t1.col1 = t2.col1;
```

Here t1 and t2 are used as aliases for table1 and table2 respectively.

Types of Join

The type of the join is decided by the way we write that join condition. ORACLE supports following types of joins.

- Equi-Join and Natural join
- Cartesian Join
- Non-equi Join
- Self Join
- Outer Join

Equi-Join

A join that is formed as a result of equality condition between the two columns is called an equi-join or a simple join. The comparison operator used in the join condition is '=' (equals).

Example

Consider a query in which we want to know the employee number, the name of the employee, job his department and its location.

Query

```
SQL> SELECT empno, ename, job, dname, loc
    FROM emp, dept
    WHERE emp.deptno=dept.deptno;
```

Result

EMPNO	ENAME	JOB	DNAME	LOC
7782	CLARK	MANAGER	ACCOUNTING	NEW YORK
7839	KING	PRESIDENT	ACCOUNTING	NEW YORK
7934	MILLER	CLERK	ACCOUNTING	NEW YORK
7369	SMITH	CLERK	RESEARCH	DALLAS
7876	ADAMS	CLERK	RESEARCH	DALLAS
7902	FORD	ANALYST	RESEARCH	DALLAS
7788	SCOTT	ANALYST	RESEARCH	DALLAS
7566	JONES	MANAGER	RESEARCH	DALLAS
7499	ALLEN	SALESMAN	SALES	CHICAGO
7698	BLAKE	MANAGER	SALES	CHICAGO
7654	MARTIN	SALESMAN	SALES	CHICAGO
7900	JAME	CLERK	SALES	CHICAGO
7844	TURNER	SALESMAN	SALES	CHICAGO
7521	WARD	SALESMAN	SALES	CHICAGO

Simple equi-join is also called as Inner join.

Example

Above same query can be written as follows using Inner join syntax.

Query

```
SQL> SELECT empno, ename, job, dname, loc
    FROM emp INNER JOIN dept ON
    emp.deptno=dept.deptno;
```

Natural join

When the join is based on all the common columns having same name in different tables, it is called as Natural join. In case of natural join you don't have to give explicit join condition as well as you don't have to qualify the common column names with their table names.

Example

Above same query can be written as follows using natural join syntax.

Query

```
SQL> SELECT empno, ename, job, dname, loc
   FROM emp NATURAL JOIN dept;
```


Cartesian Joins

In a join, rows are formed from two or more tables by comparing the contents of related columns. The joining condition is specified using the WHERE clause. This joining condition is also called as a join clause. If the join clause is omitted, a Cartesian join is performed. A Cartesian join is also called as Cartesian product or cross join. A Cartesian product matches every row of one table to every row of the other table. If table 1 contains m rows and table 2 contains n rows, a Cartesian join on table 1 and table 2 results into m x n rows. A Cartesian join is used to find out the various combinations of the information and ORACLE uses Cartesian join as an intermediate step while performing other joins.

For example

```
SQL> SELECT ename, loc
   FROM emp, dept;
```

Non-equi joins

The joins which use comparison operators other than '=' while defining the joining criteria are called non-equi joins.

These joins can result in large number of rows, because number of

rows in a non equi-join are equal to number of rows in a Cartesian product – equi join rows.

Therefore it is advisable to make a non-equi join in combination with a selection criteria to reduce the rows to a manageable range.

Self-join

A self-join is used to match and retrieve rows that have matching values in different columns of the same table.

For example

To list the employee number, name and job of each employee, along with the number, name and job of the employee's manager we can use the following query.

Query

```
SQL> SELECT  worker.empno, worker.ename,
worker.job, worker.mgr,
   manager.ename, manager.job
   FROM  emp worker, emp manager
   WHERE worker.mgr = manager.empno;
```

Result

EMPNO	ENAME	JOB	MGR	ENAME	JOB
7788	SCOTT	ANALYST	7566	JONES	MANAGER
7902	FORD	ANALYST	7566	JONES	MANAGER
7499	ALLEN	SALESMAN	7698	BLAKE	MANAGER
7521	WARD	SALESMAN	7698	BLAKE	MANAGER
7900	JAME	SCLERK	7698	BLAKE	MANAGER
7844	TURNER	SALESMAN	7698	BLAKE	MANAGER
7654	MARTIN	SALESMAN	7698	BLAKE	MANAGER

7934	MILLER	CLERK	7782	CLARK	MANAGER
7876	ADAMS	CLERK	7788	SCOTT	ANALYST
7566	JONES	MANAGER	7839	KING	PRESIDENT
7782	CLARK	MANAGER	7839	KING	PRESIDENT
7698	BLAKE	MANAGER	7839	KING	PRESIDENT
7369	SMITH	CLERK	7902	FORD	ANALYST

Points to Remember

- In a self-join a table is joined to itself as if it were two tables.
- The join on columns that contain the same type of information.
- The tables must be given an alias to synchronise, which columns are coming from which tables.

Outer joins

The join operation combines information from two tables by forming pairs of related rows from the two tables. The row pairs that form the joined table are those where the matching columns in each of the two tables have the same value. If one of the rows of a table is unmatched in this process, the join can produce results by eliminating that row.

In certain cases we may want to retrieve rows that do not having the matching values in the other table, in addition to the rows that satisfy the join condition. Outer joins can be used in such cases.

An other join can be specified using an outer join operator. A plus sign enclosed within parenthesis (+), is called an outer join operator. It is used to force a row containing NULL values to be generated to match every value of the second table for which there would normally be no match.

For example

List locations of all the departments with the employees working in

them, including the departments having no employee as well. The query to do that would be as follows.

Query

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

Result

DEPTNO	LOC	ENAME
10	NEW YORK	CLARK
10	NEW YORK	KING
10	NEW YORK	MILLER
20	DALLAS	SMITH
20	DALLAS	ADAMS
20	DALLAS	FORD
20	DALLAS	SCOTT
20	DALLAS	JONES
30	CHICAGO	ALLEN
30	CHICAGO	BLAKE
30	CHICAGO	MARTIN
30	CHICAGO	JAMES
30	CHICAGO	TURNER
30	CHICAGO	WARD
40	BOSTON	

Dummy row from emp

This row will not appear in an equi-join.

It is listed here because of an outer join.

Points to Remember

- The outer join is used to include rows in a joined result even when they have no match in a joined table.
- An outer join causes SQL to supply a 'dummy' row for rows not meeting the join condition.
- No data is added or altered in the table, the dummy rows exist only for the purpose of outer join.
- If multiple columns are required to match the join, all or none of the columns in a table's join predicates should have the (+).
- Only one of the tables in a join relation can be outer joined.
- A table can only be outer joined to one or more tables.

There are three types of Outer joins.

1. Left Outer Join
2. Right Outer Join
3. Full Outer Join

Oracle 9i also supports ANSI syntax for outer join in addition to outer join operator (+) symbol. In prior versions full outer join or two-way outer join was not supported directly.

Left outer join is required when all rows from table A need to be returned by a join query even though no matching data is available in table B.

Right outer join is required when all rows from table B need to be returned by a join query even though no matching data is available in table A.

Full outer join is required when all rows from table A as well as all rows from table B needs to be returned by a join query.

Syntax

```
   SELECT list_of_columns
   FROM tableA <LEFT | RIGHT | FULL > OUTER JOIN
tableB
```

```
    ON join_condition;
```

Example

List the employee names, department number of all employees along with all departments.

Query

```
SQL> SELECT e.ename, e.deptno, d.dname
    FROM emp e FULL OUTER JOIN dept d
    ON e.deptno = d.deptno;
```

Result

ENAME	DEPTNO	DNAME
MILLER	10	ACCOUNTING
KING	10	ACCOUNTING
CLARK	10	ACCOUNTING
FORD	20	RESEARCH
ADAMS	20	RESEARCH
SCOTT	20	RESEARCH
JONES	20	RESEARCH
SMITH	20	RESEARCH
JAMES	30	SALES
TURNER	30	SALES
BLAKE	30	SALES
MARTIN	30	SALES
WARD	30	SALES
ALLEN	30	SALES
		OPERATIONS

Set Operators

The set operators combine the results of one or more queries into one result. Data types of corresponding columns in these queries must be the same.

The set operators supported in ORACLE are

- Union
- Intersect
- Minus

The Union Operator

Multiple queries can be merged together and their results can be combined, using UNION operator. The Union operator retrieves the rows of first query plus rows of second query, excluding duplicate rows. The queries are executed independently and their output is merged. Only the final query ends with a semicolon. The number of columns in all the queries must be same and of the same data type.

An example of the UNION operator is shown below.

Example

List the different jobs from departments 20 and 30.

The query to do that would be as follows

Query

```
SQL> SELECT DISTINCT job
    FROM emp
    WHERE deptno = 20
    UNION
    SELECT DISTINCT job
    FROM emp
    WHERE deptno = 30;
```


Result

JOB

ANALYS
T

CLERK

MANAG
ER

SALESM
AN

The Intersect Operator

The Intersect operator returns the common or identical rows from all queries.

For example

List the jobs common to departments 20 and 30.

This can be done using the following query.

Query

```
SQL> SELECT DISTINCT job
   FROM emp
   WHERE deptno = 20
   INTERESECT
   SELECT DISTINCT job
   FROM emp
   WHERE deptno = 30;
```

Result

JOB

CLERK

MANAG
ER

The Minus Operator

The minus operator returns the rows unique to the first query. This can be seen in the following example.

Example

List the jobs unique to the department 20.

Query

```
SQL> SELECT DISTINCT job
   FROM emp
   WHERE deptno = 20
   MINUS
   SELECT DISTINCT job
   FROM emp
   WHERE deptno != 20;
```

Result

JOB

ANAL
YST

Subqueries

The SQL subquery is one of the advanced query techniques in SQL. This feature lets us use the results of one query as part of another query.

Subquery plays an important role in SQL mainly due to following reasons.

- Subqueries break a query down into pieces and then put the pieces together, hence very simple to implement for the user.
- Some queries just can not be written without subqueries. This is because values to be used in a query are the values returned by some other query.

Subqueries are used in the WHERE or HAVING clause of another SQL statement. The main query is also called as the outer query. Subqueries provide an efficient, natural way to handle query requests that are themselves expressed of the results of other queries.

For example

List all the employees who have the same job as ‘CLARK’.

Solution

Normally we would break this query into two queries,

first, to find out the job of ‘CLARK’.

```
SQL> SELECT job
   FROM emp
   WHERE ename = ‘CLARK’;
```

This query returns ‘MANAGER’ as the job.

second, to find out all the employees having ‘MANAGER’ as their job

```
SQL> SELECT job
   FROM emp
   WHERE job = ‘MANAGER’;
```

The same query can be written in a single statement using subquery.

Query

```
SQL> SELECT ename
   FROM emp
   WHERE job = (SELECT job FROM emp WHERE ename
= ‘CLARK’);
```

Result

ENAME

JONES

BLAKE

CLARK

Points to Remember

- The subquery is enclosed in the parenthesis.
- We can refer to columns from the outer query in the inner query.
- Outer columns can appear both in the SELECT and WHERE clause.
- We can qualify the columns from both queries by prefixing them with the name of the table.
- If the subquery returns no value, then the SQL*PLUS will return an error message “Single row subquery returned no rows”.
- If the subquery returns more than one value, the error displayed will be “Single-row subquery returns more than one row”.
- Subqueries can be divided into two broad groups
    a. Single-row subquery
    b. Multi-row subquery

Single-row subquery

This is a subquery, which returns only one value to the outer query. The query explained in the earlier section is an example of a single-row subquery.

Multi-row subquery

This is a subquery, which returns multiple values to the outer query. A multi-row subquery is written using operators discussed in the following section.

Operators in Multi-row subqueries

Different operators can be used to build the queries in which the inner query returns more than one value.

IN or NOT IN

If a subquery returns more than one row and the comparison is equality or non-equality; then the IN operators are used.

List the name, job and salary of people in department 20 who have the same job as people in department 30.

This can be done using the following query.

```
SQL> SELECT ename, job, sal
   FROM emp
   WHERE deptNO = 20
   AND
   job IN  (SELECT job
   FROM emp
   WHERE deptno = 30);
```

The outer query checks for only those jobs, which are in a list returned, by the inner query.

Result

ENAME	JOB	SAL
SMITH	CLERK	800
ADAMS	CLERK	1100
JONES	MANAGER	2975

ANY or ALL

If a subquery returns more than one row and the comparison other than equality is required (e.g., <, >, <=); then the ANY, ALL operators are used.

If the relation required for the outer column in the WHERE clause is such that it should be TRUE for any value in the list of values returned by the inner query; then the ANY operator is used.

Example

Find out the name, job and salary of people in department 20 who earn more salary than any one in department 30.

Query

```
SQL>  SELECT ename, job, sal
    FROM emp
    WHERE deptNO = 20
    AND sal > ANY (SELECT sal FROM emp
    WHERE deptno = 30);
```

Result

ENAME	JOB	SAL
JONES	MANAGER	2975
SCOTT	ANALYST	3000
ADAMS	CLERK	1100
FORD	ANALYST	3000

If the relation required for the outer column in the WHERE clause is such that it should be TRUE for all the values in the list of values returned by the inner query; then the ALL operator is used.

For example,

Find out the name, job and salary of people who earn more salary than all the people in department 30.

This can be done using the following query.

Query

```
SQL>  SELECT ename, job, sal
    FROM emp
    WHERE deptno = 20
    AND sal > ALL (SELECT sal FROM emp
    WHERE deptno = 30);
```

Result

ENAME	JOB	SAL
JONES	MANAGER	2975
SCOTT	ANALYST	3000
FORD	ANALYST	3000

Exists

The existence of rows in the inner query may be used to qualify rows of an outer query. The nested SELECT statement will be TRUE if one or more rows are found. If no rows exist in the nested SELECT statement, then that portion of the WHERE clause will be FALSE.

This can be seen in the following example.

Example

Find all the departments that have employees who exist in them.

Query

```
SQL> SELECT deptno, dname, loc
    FROM dept
    WHERE EXISTS
    (SELECT deptno
    FROM emp
    WHERE dept.deptno = emp.deptno);
```

Result

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

Types of Subqueries

There are two types of subqueries

- Nested subqueries
- Co-related Subqueries

These subquery types are discussed in the following sections.

Nested subqueries

The nested SELECT clause (the inner query) can in-turn have another SELECT clause nested into it. Nesting may be used to any

number of levels.

The example using nested subquery is as follows,

Find name, job and salary of people in department 20 who have the same job as people in the SALES department.

Query

```
SQL> SELECT ename, job, sal
    FROM emp
    WHERE deptno = 20
    AND job IN (SELECT job
    FROM emp
    WHERE
    deptno =  (SELECT deptno
    FROM dept
    WHERE dname = 'SALES'));
```

Result

ENAME	JOB	SAL
SMITH	CLERK	800
ADAMS	CLERK	1100
JONES	MANAGER	2975

Co-related subqueries

Let's consider the query where we have to find out the employees who earn more than the average salary in their own department.

This query can be split into two queries.

The main query

```
 SELECT department_id, ename, salary
 FROM employee_info
 WHERE salary >(average salary of candidate
employee's department);
```

and the subquery

```
 SELECT avg(salary)
 FROM employee_info
 WHERE department_id = (candidate row's value of
department_id);
```

Here, the main query must consider each candidate row, and invoke the subquery telling it the employee's department no. Then the inner query must calculate the average salary for that department and finally the main query must compare the employee's salary with the average salary calculated by inner query and must decide to include or exclude the candidate row.

Here, the execution of the inner query is co-related with the value of the field in one of the main query's candidate rows. This type of query is knows as co-related subqury.

The Query using co-related subquery is

Query

```
SQL> SELECT deptno, ename, sal
   FROM emp e
   WHERE sal > (SELECT avg(sal)
   FROM emp
   WHERE deptno = e.deptno);
```

Result

DEPTNO	ENAME	SAL
30	ALLEN	1600
20	JONES	2975
30	BLAKE	2850
20	SCOTT	3000
10	KING	5000
20	FORD	3000

6 rows selected.

Summary

- Data from various tables needs to be accessed together. The different mechanisms supported in ORACLE for this are joins, set operators and subqueries.
- The Process of forming rows from two or more tables by comparing the contents of related columns is called as joining tables. The resulting table is called a join between the tables. Joins are the foundation of multi-table query processing.
- Various types of joins supported by ORACLE 8 are Equi-Join, Cartesian join, Non-equi-join, Self-Join and Outer Join.
- The set operators combine the results of row or more queries into one result. The set operators supported in ORACLE 8 are Union, Intersect and Minus.
- The subquery is one of the advanced query techniques in SQL.
- Subqueries break a query down into pieces and then put the pieces together.
- Subqueries are used in the WHERE or HAVING clause of another SQL statement.
- There are two types of Subqueries Nested Subqueries and Co-related Subqueries.

❑❑❑

Chapter 5

Data Manipulation Language & Transaction processing

Objectives

At the end of this chapter, student will be able to:

- List the data manipulation language commands.
- Explain and use the data manipulation language commands.
- Identify and use the transaction processing commands.

Introduction

Data is stored in a Database in the form of values in tables. SQL, the structured query language, is used to access data from the database. SQL is used for doing various database operations like creating database objects, manipulating data in the database and controlling access to the data.

SQL commands are classified as Data Definition Language (DDL), Data Manipulation Language (DML) and Data Control Language (DCL) commands.

Using Data manipulation language, we can manipulate the data that resides in ORACLE database. The commands available to manipulate on the data are Insert, Update and Delete.

To accomplish a task, we perform a series of operations on a database. The effect of these operations needs to be treated as a success or failure of a whole group and not of individual operations. ORACLE supports such requirements using transaction processing.

In this chapter the data manipulation language commands and transaction processing commands are discussed.

Data Manipulation Language

The Data Manipulation Language is used for query, insertion, updating and deletion of data stored in the database. This includes commands like SELECT, INSERT, UPDATE, DELETE and MERGE.

The INSERT Statement

After a table has been created, we need to insert some data into the database table. For example, we want to add the data for newly joined employees to the emp table.

The INSERT command lets us to add a row of information into the table. The prerequisite is

- the table must be in our own schema or
- we must have INSERT privilege on the table.

The INSERT command with the VALUES clause is used to add a new row to a database table.

The general syntax of the INSERT Statement is

```
 INSERT INTO tablename [ ( column1,column2 ... )
]
 VALUES ( value1,value2 ... );
```

Where

(column1,column2 ...)

specifies the columns in which the set of values are to be inserted.

(value1,value2 ...)

specifies the values to be inserted into the specified columns.

Points to Remember

- A table must exist before rows can be inserted into it.
- Values must be entered in the same sequence as the column

names in the INSERT clause.

- Values must match the datatypes of the columns into which they will be inserted.
- Character values and date must be enclosed in the single quotes.
- The list of column names are used only when :

  - Inserting fewer column values than those existing in the table definition.

  - column values are specified in a sequence other than the sequence in which they are created.
- Columns that exist in the table but are not listed in the INSERT command default to NULL.

Inserting values into a Table

We will now see how to insert the rows into a table using the INSERT statement.

For example, insert a new row into the emp table. The command will be as follows.

Query

```
SQL> INSERT INTO emp VALUES
   ( 7900, 'STEVE', 'PROGRAMMER',7369,'08-JUN-
98',3000,NULL,20);
```

In the INSERT command, we must separate the values with commas, and enclose the character and date values in apostrophes. The value of the column must be in the same order in which the columns were defined at the time of creation of the table.

In the above example, the column names of the emp table have been omitted. This is acceptable if we are entering the values in exactly the same order in which the columns are defined in the table. So, in this example 7900 (empno) is placed in the first order, STEVE (ename) placed in the second order and so on. If we want

to enter the null values, just use the word NULL in the relevant field.

Inserting results of a query

This command is used to copy rows partially from one table to another. The Values clause is replaced by a query using SELECT statement. The sequence and data types of the columns in the INSERT clause and the SELECT clause must match.

```
 INSERT INTO firsttable [ ( column1,column2,
...) ]
 SELECT column1,column2, ..
 FROM othertable
 WHERE condition;
```

The following example shows the use of SELECT statement with INSERT.

For example,

Give a bonus of 10 percent of the salary to all the salesmen.

This can be done using the following query.

```
SQL> INSERT INTO bonus (ename, bonusAmout,
bonusdate)
    SELECT ename, sal*0.1, sysdate
    FROM emp
    WHERE job = 'SALESMAN';
```


Points to Remember

- The SELECT clause with INSERT statement
  - has the same syntax as SQL query.
  - can be any complex SQL query but without ORDER BY clause.
- The sequence and datatypes of the columns listed in the INSERT clause must match the sequence and the datatypes of columns returned by the SELECT statement.

Inserting values using parameter substitution

Parameter substitution provides an easier way to enter data into a table. The '&' symbol is used as the substitution operator. When a substitution operator is used, SQL*PLUS prompts for the value of the variable.

For example, insert a row into emp table using parameter substitution. This can be done using the following query.

```
SQL> INSERT INTO emp VALUES
   (&empno,
'&ename','&job',&mgr,'&hiredate',&sal,NULL,&dept
no);
```

The MERGE statement

MERGE statement is used to select rows from one table for insertion or update rows in to another table based on given condition. The decision whether to update or insert in the target table is based on the condition given in the ON clause of the command.

The general syntax is

```
 MERGE INTO table1 USING table2
 ON (join_condition)
 WHEN MATCHED THEN UPDATE SET col=value
 WHEN NOT MATCHED THEN INSERT (column_list)
VALUES (column_values);
```

Data from the table 2 is updated into table 1 if join condition is satisfied or data is inserted into table 1 from table 2 if join condition is not satisfied.

For example,

Merge table emp_temp into emp table.

```
 MERGE INTO emp e USING emp_temp t
 ON (e.empno = t.empno)
```

```
 WHEN MATCHED THEN UPDATE SET e.job = t.job,
e.sal = t.sal,
                  e.deptno = t.deptno)
 WHEN NOT MATCHED THEN INSERT
 VALUES (t.empno, t.ename, t.job, t.mgr,
t.hiredate, t.sal, t.comm,
  t.deptno);
```

The UPDATE statement

Updating column values can change the data in the table. The UPDATE command is used to modify column values in a table. Values of one or more columns can be updated. Updating can be carried out selectively using WHERE clause.

The general syntax is

```
 UPDATE tablename
 SET column = expression or value
 WHERE conditions;
```

For Example,

Change the job of the employee MILLER to a MANAGER.

```
SQL> UPDATE emp
   SET job = 'MANAGER'
   WHERE ename = 'MILLER';
```


Modifying the values in multiple columns

The values in multiple columns can be modified using the following syntax

```
 UPDATE tablename
 SET column1 = expression,
 column2 = expression,
   ...
   WHERE condition ;
```

Example,

Raise the salary by 10 percent and a new commission of 15 percent of the salary for employees from department 30, who are not eligible for commission at present.

```
SQL> UPDATE emp
   SET sal = sal* 1.1,
   comm = 0.15 * sal
   WHERE deptno = 30 and comm is NULL;
```


Points to Remember

- UPDATE operates on all the rows that are selected by WHERE clause.
- If WHERE clause is omitted, all rows will be updated.
- We may use ORACLE functions that operate on a single row in the SET clause.
- Multiple SET assignments are allowed in the UPDATE command.

Deleting rows from a table

Two commands to delete rows from a database table are DELETE and TRUNCATE. These commands are discussed in the following sections.

The DELETE command

The DELETE command is used to remove rows from a table. The entire row is deleted from the table and only specific columns cannot be deleted from a table. A Set of rows can also be deleted from a table by specifying the condition using the WHERE clause. The general syntax is

```
DELETE [FROM] tablename
WHERE condition;
```

For example, to delete the details of JONES from the emp table

the following query can be used.

```
SQL> DELETE FROM emp
   WHERE ename = 'JONES';
```

Points to Remember

- The DELETE command deletes the complete row and we can not delete only specific columns, therefore column names are not specified.
- The WHERE clause determines which rows will be removed.

Transaction Processing in ORACLE

This section provides the complete description of transaction processing in ORACLE.

What are Transactions?

A transaction is a series of database operations which are treated as a single logical unit. Transaction is a series of SQL statements that either succeeds or fail as a group. It is a logical unit of work or a sequence of logical operations or events which are treated as a single unit.

A series of SQL commands which can bring about changes to the data stored in tables, like INSERT, UPDATE or DELETE can be grouped together into logical transactions to ensure data consistency and integrity.

In a transaction, when we are manipulating the data in the database, the changes made by us are not normally made permanent in the database until we give a command to do so or we exit ORACLE.

The need of Transactions

The day to day (real–life) applications like electronic money transfer, reservation systems or accounting system involve a series of operations to complete a task. This series of operations needs to

be treated as an atomic unit – 'a logical transaction' and we also need to have the facility for undoing changes under certain circumstances.

For example, in a railway reservation system we have a request to book a ticket for a particular train against the cancellation of ticket for some other train. It is also specified that the existing ticket is to be cancelled only if we are able to book it for the other train.

This task involves two operations

- Cancellation of a ticket for one train and
- Booking of a ticket for some other train

These operations can be accomplished by executing a set of commands on the reservation database. If one operation in a set of operation fails it is not acceptable here. Here the set of operations should either succeed or fail as a group and not as the individual operations.

The transaction processing support provides a solution for such situations. A series of changes made using SQL commands like INSERT, UPDATE or DELETE can be grouped together using transactions.

In a transaction, the changes made by us are not normally made permanent in the database until we confirm them. If we don't confirm changes the database maintains its original state i.e. changes made during the database operations are undone.

How a transaction begins ?

A Transaction begins when

- a user logs into ORACLE and executes a Data Definition Language command (like CREATE, ALTER or DROP) or a Data Manipulation Language command (like INSERT, UPDATE or DELETE).
- a user executes first DDL or DML command after the last transaction.

How a transaction ends ?

A transaction ends when

- the user gives a COMMIT or ROLLBACK command.
- a DDL or DCL command is issued.
- an ORACLE error occurs.
- a user logs out of ORACLE.

Transaction Commands

There are three transaction-processing commands supported by ORACLE. These are

- COMMIT
- ROLLBACK
- SAVE POINT

Saving Changes - COMMIT

We can make the changes permanent in the database by using the COMMIT command.

The syntax is

```
COMMIT [ WORK ] ;
```

where,

WORK – this keyword is optional and provided only for the readability of the command.

The effect of COMMIT command is as follows

- the changes are made permanent to the database.
- the current transaction is marked as closed.
- the new transaction is started.
- any locks acquired during the transaction are released.

Undoing changes - ROLLBACK

It may so happen that we may not want to make the data permanent in the database, due to the failing of some SQL statement in a sequence or some unintended result; then we may use the command **ROLLBACK** to discard the changes made.

The syntax is

```
ROLLBACK [ WORK ];
```

The effect of ROLLBACK is as follows

- Any changes made by the transaction are undone.
- New transaction is started.
- Any Locks acquired by the transaction are released.

ROLLBACK statement undoes the entire transaction. If we want to undo only part of the transaction we may use the SAVEPOINT command.

Controlling partial Transactions – SAVEPOINT

The transactions can be divided in to smaller portions using save points. A save point is kind of a bookmark. We may define different savepoints at different locations during our transaction by using the command like this.

```
SAVEPOINT save_point_name ;
```

Then, during the transaction if we want to undo the change upto certain savepoint, we may use the ROLLBACK command as shown below.

```
ROLLBACK [ WORK ] TO save_point_name ;
```

For example,

--- login to ORACLE ---

…

--- new transaction started ---

```
SQL> INSERT INTO . . .
SQL> UPDATE . . .
SQL> . . .
SQL> SAVEPOINT A
SQL> UPDATE . . .
SQL> DELETE . . .
SQL>  . . .
```

```
SQL> UPDATE . . .
SQL>  . . .
SQL> SAVEPOINT B
SQL> DELETE . . .
SQL> INSERT INTO . . .
SQL> ROLLBACK TO  B
SQL> DELETE  . . .
SQL>  . . .
```

The effect of using ROLLBACK TO save_point_name

- The changes made by the statements issued after the savepoint are undone (in the above example changes made after B).
- The current savepoint remains still active. (in the above example save point B).
- Any locks and resources acquired by the SQL statements since the savepoint are released (in the above example after save point B).
- The transaction is still active as there are SQL statements pending (in the above example changes made upto the save point B).

Savepoints are useful in a longer and more complex programs. The Savepoint names must be distinct within given transaction. If we create a second savepoint with the same name, the earlier savepoint is erased.

If a COMMIT or an unconditional ROLLBACK is issued in the transaction, all the savepoints defined in that transaction are erased.

Committing Transactions Automatically

SQL*PLUS has the facility to automatically commit our work without us explicitly doing so. This is set by the environment's AUTOCOMMIT feature. We can find out the value set for the AUTOCOMMIT parameter for our SQL*PLUS session by typing.

```
SHOW AUTOCOMMIT
```

It shows whether it is on or off. If it is off, which is by default, then it

means that the changes made by we won't be made permanent unless we issue COMMIT command.

If it is on, the SQL*PLUS issues COMMIT command automatically after each command is issued by us.

Summary

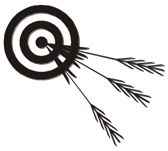
- Using the Data Manipulation Language, we can manipulate the data that resides in an ORACLE database.
- ORACLE allows us to insert a new row into the database table using the INSERT command.
- We may also insert the result of a query into a database table.
- We can modify the existing database contents by using the UPDATE command.
- We can use the DELETE command to remove the row from the database table.
- We can use the MERGE command to merge the data from two tables or updating a data in one table using data from another table based on the condition
- A transaction is a series of database operations that are treated as a single logical unit.
- In a transaction, the changes made by us are not normally made permanent in the database until we confirm them.
- The three transaction-processing commands are COMMIT, ROLLBACK and SAVEPOINT.

❑❑❑

Chapter 6

Data Definition Language

Objectives



At the end of this chapter, student will be able to:

- Identify the Data Definition Language commands.
- Create a database table.
- Modify the existing definition of the table.
- Identify the constraints and their use.

Introduction

The Data is stored in a database in the form of values in tables and SQL is used to access data from the database. The SQL is used for doing various database operations like creating database objects, manipulating data in the database and controlling access to the data.

The SQL commands are classified as Data Definition Language (DDL), Data Manipulation Language (DML) and Data Control Language (DCL) commands. Data Definition Language (DDL) is a set of SQL commands used to create, modify or remove various database objects. This includes the commands like CREATE, ALTER, DROP.

While creating a database table, we can also define the integrity constraints. Integrity constraints ensure the integrity of the data i.e. the correctness, completeness of the information with respect to the application.

In this chapter will discuss creating database tables and defining integrity constraints.

The Data Definition Language

In a database, the data is stored in the form of values in Tables. The data is organized in a table in terms of rows and columns. Before we manipulate (insert or update or delete) the information in a database, the structure of information needs to be defined. Creating database tables does this. When we create a table in an

ORACLE database we must specify the columns it will contain, their data type, size and constraints (if required).

The column's data type describes the type of the information, the valid data range, specific format and constraints. A column's data type describes the basic type of data that is acceptable in the column. For example, empno column in our sample 'emp' table uses the data type NUMBER.

If we define this column as

Empno NUMBER (4)

It means Empno column's data type is Number and it can accept a four-digit number from -9999 to 9999.

While creating a database table, we can also define the integrity constraints. Integrity constraints ensure the integrity of the data i.e. the correctness, completeness of the information with respect to the application. We can define various types of constraints like UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK and others.

For example, If we define empno column in our sample 'emp' table column as

Empno NUMBER (4) UNIQUE

It means Empno column's data type is Number and it can accept a four-digit number from -9999 to 9999 and no two employees can have the same employee number.

SQL (Structure Query Language) is used for doing various database operations like creating database objects, manipulating data in the database and controlling access to the data. SQL commands are classified as Data Definition Language (DDL), Data Manipulation Language (DML) and Data Control Language (DCL) commands.

The Data Definition Language (DDL) consists of a set of commands used to create the database objects such as tables, views, indexes etc. This includes commands like CREATE, ALTER and DROP.

In this chapter we will discuss SQL commands for working with database structure called tables. Working with tables includes

- creating or defining table structures
- modifying table structures
- deleting table structures

We will also discuss the various integrity constraints like UNIQUE, NOT NULL, PRIMARY KEY, FOREIGN KEY, CHECK and others.

Creating Database Tables

A table can be created by using CREATE TABLE statement. We need the CREATE TABLE system privilege to execute this command.

The general syntax of the CREATE TABLE statement is as follows:

```
CREATE TABLE [schema.]tablename (
column1 datatype ( size ) [ DEFAULT < expr > ] [
CONSTRAINT constraint_name ] [ column_constraint
] [ enable / disable ],
column2 datatype ... ,
...
[ CONSTRAINT constraint_name ][table_constraint
] [ ENABLE / DISABLE ],
...
);
```


When we create a table, we become the owner of the table with the name specified in the CREATE statement.

Let us examine the syntax in more details

- **tablename -** Specifies the name of the table we want to create.
- **column1 -** Specifies the name of the column.
- **DEFAULT -** Specifies a value to be assigned to the column if any subsequent INSERT statement omits a value for the column. The value can be specified by using the expression which can be a combination of one or more values, operators, and SQL functions that evaluates to a value.
- **CONSTRAINT -** Identifies the integrity constraint by the name specified as constraint_name. This name is stored in the data

dictionary along with the definition of integrity constraint. If this identifier is omitted, ORACLE generates a name with the form SYS_Cn, where n is an integer that makes the name unique within the database.

In ORACLE, maximum 1000 columns are allowed per table. When we create a table, we become the owner of that table.

Naming Rules for database objects

Following are the rules applicable to the database object names when we create them

- The name must start with an alphabet.
- The name may contain letters, numerals and special characters (like _ i.e. underscore, $ i.e. dollar).
- The maximum length is 30 characters.
- Database object names are case insensitive.
- The name cannot be a SQL reserve word.

Integrity Constraints

The Integrity constraints help us to enforce business rules on data in the database. Once we specify an integrity constraint for a table, all data in the table always conforms to the rule specified by the integrity constraint.

If we do not use integrity constraints, we need to program our applications to check whether the data complies to our set of rules or not. This adds to the development overheads. Also, there are changes of violation of data integrity.

For example, in an organization, an employee could be working for one of the departments ACCOUNTS, RESEARCH, SALES or OPERATIONS. This condition becomes a business rule for the organization. Therefore, it is required to enforce the business rule that an employee must belong to one of the valid department.

The constraints that can be defined while creating database tables are as follows.

- NULL, NOT NULL
- UNIQUE

- PRIMARY KEY
- FOREIGN KEY, REFERENCES, ON DELETE CASCADE
- CHECK

The use of these constraints can be explained in brief as follows :

CONSTRAINT	USE
NULL	Specifies that a column can contain null values.
NOT NULL	Specifies that a column can not contain null values.
UNIQUE	forces the column to have unique values.
PRIMARY KEY	designates a column or combination of columns as primary key. ORACLE also forces unique constraint on these columns.
FOREIGN KEY	designates a column or combination of columns as foreign key in a referential integrity constraint. Use this key-word only while defining foreign key with a table constraint.
REFERENCES	identifies the primary or unique key that is referenced by a foreign key in referential integrity constraint. If only parent table is specified and column name is omitted then the foreign key automatically references to the primary key of parent table.
ON DELETE CASCADE	specifies that ORACLE maintains referential integrity by automatically removing dependent primary or unique key value.
CHECK	specifies a condition that each row in the table must satisfy.
ENABLE	enables an integrity constraint.
DISABLE	disables an integrity constraint.
DEFAULT	specifies value to be assigned to the column if it is not supplied in the INSERT statement.

There are two ways by which integrity constraints can be specified while creating a table. An integrity constraint can be defined as

- Column Constraint or
- Table Constraint

Column Constraints

A Column constraint defines an integrity constraint as a part of the column definition. The use of column constraints is shown in some of the table definitions given below

NOT NULL Constraint

Example:

```
CREATE TABLE emp (
empno  NUMBER (4) NOT NULL,
ename VARCHAR2(10),
....
);
```

In the example, the employee number column is defined as a four digit number with NOT NULL constraint. This means for each employee row, we insert in the emp table, we have to supply the value for this column.

UNIQUE Constraint

Example:

```
CREATE TABLE emp (
empno  NUMBER (4) NOT NULL UNIQUE,
ename VARCHAR2 (10),
....
);
```

Here the employee number column is defined as a four digit number with NOT NULL and UNIQUE constraints. This means for each employee row, we insert in the emp table, we have to supply the value for this column and this value has to be unique for each row in the table.

Typically Primary key columns need to satisfy both of these constraints. If we want to define a Primary Key it can also be done using PRIMARY KEY constraint.

PRIMARY KEY Constraint

```
CREATE TABLE emp (
empno  NUMBER (4) PRIMARY KEY,
ename VARCHAR2 (10),
....
);
```

A primary key has been defined here as a column constraint and is on empno column. A column with Primary Key constraint has to satisfy both Unique and Not Null constraints implicitly.

FOREIGN KEY Constraint, REFERENCES and ON DELETE CASCADE Options

Example :

```
CREATE TABLE emp (
empno  NUMBER (4) PRIMARY KEY,
ename VARCHAR2 (10),
....
deptno NUMBER(2) REFERENCES dept(deptno));
```

This creates a table emp with the employee number and other columns. A primary key is based on empno column. It also defines a column deptno that stores a two-digit department number. The Foreign Key constraint is based on deptno column. This establishes a relationship between emp and dept tables. If a Foreign Key is defined as a column constraint, the keyword FOREIGN KEY is never used.

The ON DELETE CASCADE option can also be used with foreign key definition. For example,

```
CREATE TABLE emp (
empno  NUMBER (4) PRIMARY KEY,
ename VARCHAR2 (10),
....
deptno NUMBER(2) REFERENCES dept(deptno) ON
DELETE CASCADE
);
```

Here, the use of ON DELETE CASCADE option will delete all the dependent rows from emp table if a row is deleted from dept table.

The dependent rows in emp table are those rows referring to the column values of a row to be deleted from dept table.

CHECK Constraint

A CHECK constraint may be used to enforce some business rule. For example, to enforce a business rule that restricts the salary of an employee to a maximum of 6000, we may use the following table definition

```
CREATE TABLE emp (
empno  NUMBER (4) PRIMARY KEY,
ename VARCHAR2 (10),
....
sal NUMBER(7,2) CHECK ( sal < 6000),
...
deptno NUMBER(2) REFERENCES dept(deptno) ON
DELETE CASCADE);
```

Pseudo columns such as CURRVAL, NEXTVAL, ROWNUM or subqueries, call to SYSDATE, UID, USER functions are not allowed in the condition of check integrity constraint.

DEFAULT Constraint

A DEFAULT constraint may be used to assign a default value to be assigned to a column. Default value is assigned to a column if subsequent Insert statement omits the value for the column.

```
CREATE TABLE emp (....,
Hiredate DATE DEFAULT SYSDATE,
....);
```

Enabling and Disabling Constraints

A constraint defined in a table definition is in enabled state by default. This means that the business rule defined by the constraint will be enforced. Any row violating this rule will be rejected during the database operations. We can define a constraint and keep it disabled using the keyword DISABLE.

For example,

```
CREATE TABLE emp (
empno  NUMBER (4) PRIMARY KEY,
ename VARCHAR2 (10),
```

```
....
sal NUMBER(7,2) CHECK ( sal < 6000) DISABLE,
...
deptno NUMBER(2) REFERENCES dept(deptno));
```

Table Constraints

A Table constraint is the constraint that is applied to one or more columns of a table. The table constraints are defined after the last column definition. For example,

```
CREATE TABLE emp (
empno  NUMBER (4) PRIMARY KEY,
ename VARCHAR2 (10),
....
sal NUMBER(7,2) CHECK ( sal < 6000) DISABLE,
...
deptno NUMBER(2),
….
CONSTRAINT emp_fk FOREIGN KEY ( deptno )
 REFERENCES dept ( deptno ) ) ;
```

Here a foreign key has been defined as a table constraint. Like this we can define primary key and other constraints as table constraints.

Creating a table using Subquery

The table can also be created from the existing table. The general syntax for this is

```
CREATE TABLE tablename [( column_list )]
AS query ;
```

Here, the columns assume the data types, sizes and column names of the columns returned by the query. If column list is specified explicitly for respective columns then new table will have respective column names. Constraints are not inherited from original table except Not Null constraint. Rows selected by the query will be added to the newly created table.

For example, using above mentioned syntax, we will create tables emp_new and dept_new from the existing emp and dept tables

respectively.

To create the new employee table use the following query:

```
SQL> CREATE TABLE  emp_new
   AS SELECT * FROM emp;
```

To create the new department table use the following query

```
SQL> CREATE TABLE dept_new
   AS SELECT * FROM dept;
```

Modifying the Table Structure

After a table is created and has been in use, we might want to make changes to its structure (definition). We may change the structure of a table in following ways

- Add one or more columns
- Redefine the existing column definitions
- Redefine the integrity constraints

The ALTER TABLE command is used to modify the table structure. The use of ALTER TABLE command is discussed in the following sections.

The general syntax of the CREATE TABLE statement is as follows:

```
ALTER TABLE tablename (
[ MODIFY | ADD ] (column definitions),
([ ENABLE | DISABLE ] [ CONSTRAINT
constraint_name ]
[ DROP COLUMN column_name ]
);
```

Adding columns

To add a new column in the table definition, we can use ALTER TABLE command with an ADD clause.

Syntax :

```
 ALTER TABLE tablename
```

```
 ADD ( column_definition );
```

Example: To add a column department head (dept_head) to the dept table, use the following query

```
SQL> ALTER TABLE dept
   ADD ( dept_head VARCHAR2 (10) ) ;
```

Points to Remember while adding a column

- We may add one or more columns.
- We may add a column at any time with a NULL constraint.
- We may not be able to add a column with NOT NULL specified.
- We can add a NOT NULL column if we follow a roundabout way like
  - Add the column without specifying NOT NULL.
  - Fill every row in that column with data.
  - Modify the column to be NOT NULL.

Modifying Column Definitions

The ALTER TABLE command with MODIFY clause is used to modify a column definition. The syntax is

```
 ALTER TABLE tablename
 MODIFY ( new_column_definition );
```

For example, to modify the size of the dept_ head to 25, use the following query

```
SQL> ALTER TABLE dept
   MODIFY ( dept_head VARCHAR2(25));
```

Points to Remember

- We can increase the column size at any time.
- To decrease a column size, the column must be NULL in all existing rows.
- We can change the column from NOT NULL to NULL at any time.

- To add the NOT NULL constraint to a column, the column must have a value for each row in the table.
- To change the datatype of a column, all values in the column must be NULL in all rows.
- Only Null, Not Null and Default constraints can be modified with Modify clause.
- Rest of the constraints can be changed with the help of the Add clause after dropping earlier constraint.

The Drop Clause

This clause is used to remove the constraints from a table.

The syntax is

```
ALTER TABLE tablename
DROP
[COLUMN column_name ]
[CONSTRAINT constraint_name]
[PRIMARY KEY]
[UNIQUE (column,column,.....)]
[CASCADE];
```

Using CASCADE option with DROP, we can remove all the dependent constraints.

Example

```
SQL> ALTER TABLE dept
   DROP PRIMARY KEY
   CASCADE;
```

This will drop the primary key definition in the dept table and also the foreign key definition from the emp table. This is because the foreign key definition in the emp table is dependent on (i.e. references) the primary key definition from dept table.

Enabling / Disabling Column Constraints

To enable or disable constraints we can use ENABLE or DISABLE option of the ALTER TABLE command. For example, if we use the following query

```
SQL> ALTER TABLE emp
   DISABLE PRIMARY KEY;
```

This will disable the primary key constraint of the emp table.

The TRUNCATE command

The TRUNCATE command can also be used to remove all the rows from the table somewhat similar to Delete DML command.

Syntax :

```
TRUNCATE TABLE tablename;
```

Example

```
SQL> TRUNCATE TABLE emp;
```

This removes all the rows from the emp table. The table definition still remains in the database.

Points to Remember

- The TRUNCATE is faster than the DELETE command. This is because it generates no rollback information, does not fire any delete triggers.

The RENAME command

The RENAME command is used to rename the existing table name.

Syntax :

```
RENAME <old_tablename> TO <new_tablename;
```

Example

```
SQL> RENAME emp TO employee;
```

This renames emp table to table employee. The table emp no more exists, the same table is available with the name employee.

Deleting a Table structure

To delete a table structure DROP command is used. The DROP command deletes the table definition and all its data.

Syntax :

```
DROP TABLE tablename
```

```
  [ CASCADE CONSTRAINTS ];
```

Example: To delete the emp table we can use the following query..

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

The CASCADE CONSTRAINTS option in DROP command drops all referential integrity constraints that refer to primary and unique keys in the dropped table.

For example, to delete the dept table and all the dependent constraint definitions, we can use the following query

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

This will delete the dept table, all the constraints from dept table (primary key definition) and also the dependent constraints from the emp table (foreign key definition).

If this option is omitted, ORACLE returns an error message and does not drop the table. This is because, we have the dependent rows and constraints on the table we are trying to delete.

Summary

- The SQL is used for doing various database operations like creating database objects, manipulating data in the database and controlling access to the data.
- The SQL commands are classified as Data Definition Language (DDL), Data Manipulation Language (DML) and Data Control Language (DCL) commands.
- The Data Definition Language consists of a set of commands used to create the database objects such as tables, views, indexes etc. This includes commands like CREATE, ALTER and DROP.
- A table can be created by using CREATE TABLE statement.
- When we create a table in an ORACLE database we must specify the columns it will contain, their data type, size and constraints (if required).

- Two ways in which the integrity constraints can be specified while creating a table are Column Constraints and Table Constraints
- After a table is created and has been in use, we may change the structure of a table by adding one or more columns, redefine the existing column definitions and redefine the integrity constraints
- To delete a table structure DROP command is used. The DROP command deletes the table definition and all its data.

❑❑❑

Chapter 7

Views, Synonyms and Sequences

Objectives

At the end of this chapter, student will be able to:

- Discuss the concept of a View.
- Explain creation and the use of Views.
- List the restrictions on Views.
- Identify the use of Synonyms.
- Identify the use of Sequences.

Introduction

One of the important features of RDBMS is the Data Abstraction. The Data Abstraction gives different view of data to different users. All the information in a database need not be accessible to all the users. Sometimes, in an application, a different view of the data or the information is desired and the relevant data changes from user to user. This is handled in ORACLE using the VIEWS.

A SYNONYM is a simple alias for a table, view, sequence, or other database objects. Synonyms can be used for giving meaningful alternative names for database objects.

A SEQUENCE is a database object used to generate the series of unique integers. Sequences are typically used with primary key or unique columns.

In this chapter we will explain concepts and the use of Views, Synonyms and Sequences.

Views

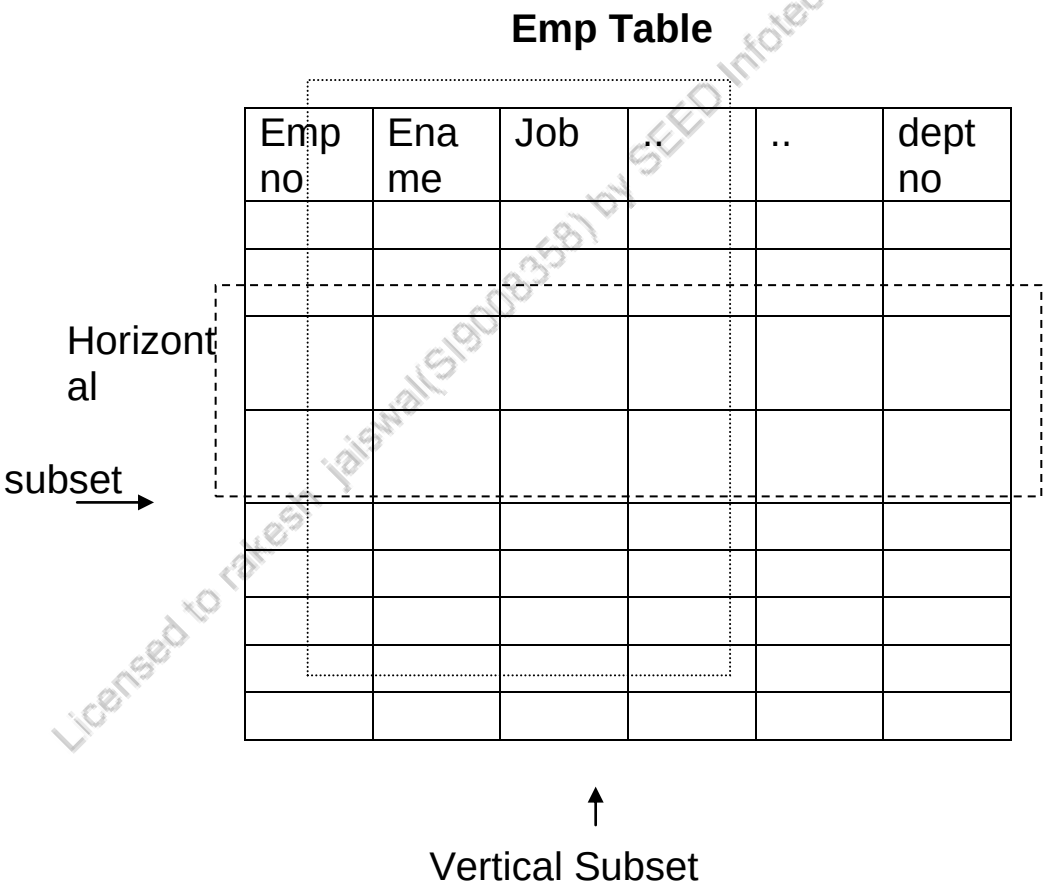
In this section we describe the concept and use of views.

What is a View?

The table of a database defines the structure and the organization of its data. Once they are defined they always present the data in a particular way. Sometimes, in an application, a different view of the data or the information in a different format is desired. This is handled in ORACLE using the VIEWS.

A VIEW is a virtual table in the database. The contents of a view are defined by a query.

A view can represent a subset of the data in a table. This could be a horizontal subset consisting of some of the rows from the base table or vertical subset consisting of some of the columns from the base table. The data from multiple tables can be also combined together using a view



Characteristics of a View

Views do not exist physically. Views are virtual tables that exist only as definitions in the system catalogue. Views are stored in the data dictionary in the table called USER_VIEWS.

Advantages of Views

Views provide several advantages and can be useful in various ways. In small (desktop) applications, views can be used to simplify the data requests. In large database applications like production data views can be used to restrict access to the data and enforcing security.

The major advantages of views are

Security

Each user can be given permission to access the database only through a small set of views that contains the specific data the user is authorized to see, rather than the entire table, thus restricting the user's access to stored data.

Query Simplicity

A view can draw data from several different tables and present it to the user as a single table, turning what would have been multi-table queries into single table queries against the view.

Structural Simplicity

Views can give a user a " personalized " view of the database structure, presenting the database as a set of virtual tables that make sense for that user.

Insulation from Change

A view can present a consistent, unchanged image of the structure of the database, even if the underlying source tables are split, restructured or renamed.

Data Integrity

If data is accessed and entered through a view, the DBMS can automatically check the data to ensure that it meets specified integrity constraints.

Disadvantages of Views

While views provide substantial advantages as discussed above,

there are also two major disadvantages of using a view instead of a real table:

Performance

Views give the appearance of a table, but the DBMS must still translate the queries against the view into queries against the underlying source tables. If the view is defined by a complex, multi-table query, then even a simple query against the view becomes a complicated join, and it may take a long time to execute.

Update restrictions

When a user tries to update rows of a view, the DBMS must translate the request into an update on the rows of the underlying source tables. This is possible for simple views, but more complex views cannot be updated; they are “read-only”.

The above disadvantages mean that we cannot indiscriminately define views and use them instead of the source tables. Instead, we must in each case weigh the pros and cons of creating a view for a given situation.

Creating a View

A view can be created by using a CREATE VIEW command.

The general syntax of the command is

```
CREATE [OR REPLACE] [FORCE / NOFORCE]
[(column_list)]
VIEW view_name
AS query
[WITH CHECK OPTION]
[WITH READ ONLY];
```

Where

- **CREATE** - creates the view with the name specified.
- **OR REPLACE** - replaces the view if it already exists. This option is used to change the definition of an existing view.
- **FORCE** - creates the view (with compilation errors) regardless

of whether the view's base table exists or the owner has the privilege on them. But to view the VIEW the owner must have the privileges and the base table must exist.

- **NOFORCE** - creates the view only if the base table exists and the owner has the privileges on them. (This is the default)
- **COLUMN_LIST** – specifies different column names than their original names. These new column names can be only used while referring to views.

Example on creating a View :

Create a view on emp table which gives access to the employee number, employee name and designation information of employees working in the department 30 only. This can be done using following query.

Query

```
CREATE OR REPLACE VIEW empview30 AS
SELECT empno, ename, job
FROM emp
WHERE deptno = 30;
```


Points to Remember

- The view's default column names are same as the table's column names.
- New column names if specified in the CREATE VIEW clause, have one-to-one relationship with the column names in the SELECT clause of the query.
- The GROUP BY clause can be used in the definition of a view
- Views may be joined or nested with other views or tables
- Views may be used in the SELECT statement while defining other views

Querying a View

A View can be queried and used just like database tables. For example, If we want to see the structure of view (in the SQL*PLUS environment) we use the DESCRIBE command. This is the same command we used earlier for tables.

Syntax:

```
Desc[ribe] view_name
```

Example :

Display the structure of the view empview30.

Query

```
SQL> Desc empview30
```

We can query a view just like a database table. This can be seen in the following examples.

Example1:

List the details of employees working as clerk from department 30.

Query

```
SQL> SELECT *
     FROM empview30
     WHERE job = 'CLERK';
```

Example2

List the details of employees working as salesman from department 30 in the ascending order of their name.

Query

```
SQL> SELECT *
   FROM empview30
   WHERE job = 'SALESMAN'
   ORDER BY ename;
```


Types of View

The definition of a view decides what operations can be performed on a view. On this basis views are categorised as

- Simple or Updateable Views
- Complex or Non-updateable Views

Complex or Non-updateable Views

The Complex or Non-updateable Views are used only to retrieve the corresponding data from the table. We cannot use the DML statements like INSERT, UPDATE, or DELETE with these views.

Simple or Updateable View

When we refer to the term ‘ updating the views’ what actually implies is the updating of the underlying source table using the view. Views can be updated much the same way we update the tables i.e. by using the DML commands.

For an updateable view there are certain restrictions imposed by the ANSI / ISO SQL standard.

These restrictions are as follows

- The FROM clause must specify only one updateable table.
- DISTINCT must not be specified; i.e. duplicate rows must not be excluded from the query result.
- Each SELECT item must be a simple column reference; the SELECT list cannot contain expressions, calculated columns, or column functions.
- The WHERE clause must not include a sub query. Simple row-by-row search conditions may be used.
- The query must not include a GROUP BY or HAVING clause.

Manipulating Base Tables Using Views

We can manipulate the base table using views. The DML commands Insert, Update, or Delete can be used with views.

The following restrictions apply, while manipulating base tables through views.

- The view must be based on a single table.
- It must not have columns that are aggregate functions.
- It must not have expression in its definition.
- It must not use distinct in its definition.
- It must not use group by or having clause in its definition.
- It must not use sub queries.
- We cannot insert if the underlying table has any NOT NULL

columns that don't appear in the view.

Some of the view definitions and remarks about the way they can be updated are as follows

Example 1:

```
SQL> CREATE OR REPLACE VIEW empview As
    SELECT *
    FROM emp
    WHERE deptno = 30;
```

Remark

The view empview is updateable because it does not violet any of the restrictions mentioned above.

Example 2

```
SQL> CREATE OR REPLACE VIEW empsalview  ( empno,
ename, sal, TotalSal) As
    SELECT empno, ename, sal, sal + NVL(comm,0)
    FROM emp
    WHERE deptno = 30;
```

Remark

The view empsalview is not updateable because it contains an expression.

Specific restrictions on a view for DML operations are discussed in the following sections.

Delete Restrictions

We cannot delete the rows when the view contains

- Group Function
- DISTINCT Clause
- GROUP BY Clause
- Join Condition

Example: The following view has a delete restriction because it is based on two tables.

```
SQL> CREATE OR REPLACE VIEW empview As
```

```
    SELECT emp.*, dept.dname
    FROM emp, dept
```

Update Restrictions

We cannot update a view when view contains an expression such as SAL+COMM.

Other restrictions are same as stated for delete.

For example, the following view has an update restriction because it contains an expression.

```
SQL> CREATE OR REPLACE VIEW empsalview ( empno,
ename, sal, TotalSal) As SELECT empno, ename,
sal, sal + NVL(comm,0) FROM emp;
```

Insert Restrictions

We cannot insert a row using view when a view does not contain all the NOT NULL columns of the base table

Other restrictions are same as stated for delete and update.

For example, the following view has a Insert restriction. This is because it does not include empno column.

```
SQL> CREATE OR REPLACE VIEW empsalview  (ename,
sal, comm) As
    SELECT ename, sal, comm
    FROM emp;
```

WITH CHECK OPTION

Some times, INSERT or UPDATE operations on a View can result in data that the View can't retrieve. In such case you might want to restrict the View so that the View does not accept any data that it can't display.

Example :

```
SQL> CREATE OR REPLACE VIEW empview30 AS
 SELECT empno, ename, deptno
 FROM emp
 WHERE deptno = 30;
```

The above view can display records for department number 30

only. Being Simple or updateable view you can execute following DML on it.

```
SQL> INSERT INTO empview30
   VALUES (1234, 'SACHIN', 10);
```

However, when you query on empview30, it can not display above inserted information. The WITH CHECK OPTION can be used with CREATE VIEW statement for preventing user against entering data through view which can't be displayed by view.

```
SQL> CREATE OR REPLACE VIEW empview30 AS
   SELECT empno, ename, deptno
   FROM emp
   WHERE deptno=30
WITH CHECK OPTION;
```

WITH READ ONLY option

Some times you may want to prevent users from making changes to the base table through view. In such case use WITH READ ONLY option with CREATE VIEW command.

Example :

```
SQL> CREATE OR REPLACE VIEW empview AS
   SELECT empno, ename, job, deptno
   FROM emp
   WITH READ ONLY;
```

Dropping a view

We can drop a view using following command.

Syntax:

```
 DROP VIEW view_name;
```

Synonyms

A SYNONYM is a simple alias for a database object. Synonyms can be used for giving meaningful alternative names for database objects. In this section we describe the concept and use of synonyms.

What is a Synonym?

A synonym is a simple alias for a table, view, sequence, or other database objects. Because a synonym is just an alternate name for an object it requires no storage space. ORACLE stores only definition of a synonym in the data dictionary. ORACLE allows us to create both, public or private synonyms. A public synonym is a synonym that is available to every user in a database. A private synonym is a synonym within the schema of a specific user.

Creating a Synonym

As we know, a synonym is an alternative name for a table, view, sequence, procedure, stored function, package, snapshot, or another synonym.

To create a synonym CREATE SYNONYM command is used

Syntax

```
CREATE [PUBLIC] SYNONYM [schema.]synonym_name
FOR [schema.]object;
```

Where

- **PUBLIC** - Creates a public synonym that is assessable to all users in a database. If we omit this option, the synonym is private and is accessible only within our schema.

  For creating PUBLIC synonym you must have DBA priviledges.

- **schema** - This is the schema to contain the synonym. If we omit schema, ORACLE creates the synonym in our own schema.
- **synonym_name** - Name of the synonym to be created.

- **Object** - Identifies the object for which the synonym is created. This object can be of following types:

  - Table
  - View
  - Sequence and

- Subprograms like a Stored procedure, a function, or a package

Example: Create a synonym employee for emp table. This can be done as follows.

Query

```
SQL> CREATE SYNONYM employee
   FOR emp;
```

Now, we can use the synonym employee for emp table and write queries. For example,

```
SQL> SELECT *
   FROM employee;
```


Deleting a Synonym

To delete the synonym from the database use the DROP command

Syntax

```
DROP [PUBLIC] SYNONYM [schema.]synonym_name
```

Where

PUBLIC - To drop the public synonym, we must be specify the PUBLIC keyword.

schema - This is the schema name containing the synonym. ORACLE assumes the synonym is in our own schema, If we omit this option.

synonym_name - Name of the synonym to be deleted from database.

Sequences

A sequence is a database object used to generate the series of unique integers for use as primary keys. When an application inserts a new row into a table, the application simply requests a database sequence to provide the next available value in the sequence for the new row. Multiple users can use same sequence.

Creating a Sequence

To create a sequence, use the CREATE SEQUENCE command.

The general syntax is as shown below

```
CREATE SEQUENCE sequence_name
    [INCREMENT BY n]
    [START WITH n];
```

Where

sequence_name - The name of the sequence object to be created.

Increment By - Specifies the incremental value between sequence numbers. This value can be positive or negative, but it cannot be 0. If this value is negative, then the sequence is created in descending order. If the increment is positive, then the sequence is created in ascending order. If we omit this clause, the interval defaults to 1.

Creating a sequence is shown in the following example.

Example:

Create a sequence for generating employee numbers starting with 7700.

Query

```
SQL> CREATE SEQUENCE empseq
   INCREMENT BY 1
   START WITH 7700;
```


Using a Sequence

A sequence can be used with database operations. It is used with INSERT and UPDATE DML commands. Sequences provide two attributes for using the value generated using the sequence. These attributes are CURRVAL and NEXTVAL. Their use is as follows

- **Sequence_name.currval -** Returns the current value in the sequence.
- **Sequence_name.nextval -** Increments the current value in the sequence and returns it.

The use of a sequence can be seen in the following example.

Example :

Insert a new employee ‘BILL’ with a designation ‘CLERK’ and salary of 1000. He is to be posted to the department 20.

This can be done using the following query.

Query

```
SQL> INSERT INTO emp (empno, ename, job, sal,
dept)
   VALUES ( empseq.nextval, 'BILL', 'CLERK',
1000, 20);
```

Deleting a Sequence

To delete a sequence from the database use DROP SEQUENCE command.

Syntax

```
DROP SEQUENCE [schema.]sequence_name
```

Where

schema - Is the schema containing the sequence. The default assumes the sequence is in our own schema.

Sequence_name - The name of the sequence to be dropped.

For example, to drop the sequence empseq use following command.

```
SQL> DROP SEQUENCE empseq;
```

Summary

- A View is a virtual table. The contents of a view are defined by a query.
- A View can represent a subset of the data in a table.
- On the basis of definition, a View is categorised as
  - Read–Only View
  - Updateable View
- Views can be updated much the same way we update the tables.
- For an updateable View there are certain restrictions imposed by the ANSI / ISO SQL standard.
- A Synonym is a simple alias for a table, view, sequence, or other database objects.
- A Sequence is a database object used to generate the series of unique integers for use as primary keys.

❑❑❑

Chapter 8

Indexing and Clustering

Objectives

At the end of this chapter, student will be able to

- Describe the concept of an index.
- Create Indexes.
- Describe the concept of clusters.
- Create cluster tables.

Introduction

Once the application starts working, what user demands is the performance. The database needs to be tuned to get the better performance. Indexing and Clustering are the two common ways to improve the performance of the database tables. The approach used by both is entirely different. In this chapter these techniques are discussed.

The use of the index is the primary method to reduce the disk I/O and improve the performance of the database. When the user query on a table uses the indexed column in a query, ORACLE automatically uses the index and quickly finds specific records using index.

Improving the Database performance

The database needs to be tuned to get the better performance.

Two common ways used by database administrators and the end user are

- Indexing
- Clustering

The syntax and the use of SQL commands doesn't change when we use them.

The use of the index is the primary method to reduce the disk I/O

and improve the performance of the database. ORACLE can quickly find specific records using index. When the user query on a table using the indexed column in a query, ORACLE automatically uses the index. It quickly finds the target row and that too with minimal disk I/O (Input / Output). Without an index, ORACLE has to read entire table to find the specific record.

Indexes are useful only for the key columns that application refers frequently to find the specific rows. Unnecessary indexes can actually slow down the system and also slow down the performance of the database. This is because the indexes need to be updated with the every operation that is performed on the database. This puts the additional burden in the form of storage and also processing.

Clustering is another method of improving the database performance. It stores the data together from different tables that are related and are often accessed together. Clustering can boost the performance for tables that are often used in join queries. This is because the rows that are joined are stored together, thus reducing data retrieval time.

Indexes

The concept of a database index is similar to an index in a book or a manual. The database index is a sorted list of rows accompanied by the location of the row. For all Queries, ORACLE automatically uses the index, if available. ORACLE can quickly find specific records using an index. Indexes reduce the disk I/O and improve the performance. Without an index, ORACLE has to read entire table to find the specific record. Indexes are not a must for running ORACLE. The performance improvement is considerable for larger tables. Indexes are updated with the every operation and unnecessary indexes can actually slow down the system.

Creating an Index

The index is created using CREATE INDEX command.

Syntax

```
CREATE [ UNIQUE] INDEX [schema.]indexname
ON tablename (column1, column2 …);
```

Where,

- **Schema** - Specifies the name of the schema that contains the index. Default is current schema.
- **Indexname** - Specifies the name of the index to be created.
- **Tablename** - Specifies the name of the table on which the index is created.
- **column1, column2** … - specifies the names of columns from tables on which the index is to be created. It cannot be of data-type LONG or LONG RAW.
- **UNIQUE** – Specifies that the value of the column (or combination of columns) in the table to be indexed must be unique.

Example: Create an index on table emp on empno column

```
SQL> CREATE INDEX emp_idx
   ON emp ( empno );
```

Example: Create a unique index on table emp on employee name, so that duplicate names cannot be entered.

```
SQL> CREATE UNIQUE INDEX emp_ename_idx
   ON emp ( ename );
```

Dropping an Index

An Index can be dropped when it is no more required.

Syntax

```
 DROP INDEX [schema.]indexname
```

For example,

```
SQL> DROP INDEX emp_idx ;
```

Using Indexes

The following points are to be considered while using indexes.

Points to Remember

- We may create as many indexes as we want.
- Each Index must have a unique name.
- Indexes can ensure that no duplicate values are entered into a column. This is done using the UNIQUE keyword.
- Always create indexes on the columns referenced most often in the WHERE clause.
- Do not create too many indexes on a table.
- Create indexes only on larger tables (having more than 100 rows).
- If we often use joins on tables create an index on the joining column in one or both tables.
- More indexes will help if data is static (changes are less frequent) and frequently queried.

Index handling in ORACLE

Oracle handles the indexes in following way.

- Indexes are stored separately from the data.
- Indexes are automatically updated by ORACLE, after new data is inserted, or some rows are deleted by users.
- If the indexes exist for columns referenced in the WHERE clause, ORACLE automatically uses them where ever appropriate.
- Indexes are used to find records for all SQL statements, not just queries.

Clusters

- Clustering is a method of storing data together from different tables.
- Cluster tables are related and are often joined together while accessing.

- Clustering can boost the performance of join queries.
- Related data is stored together (in the same page on disk) and not spread to different areas on disk.
- This reduces the data retrieval time and improves performance.

Features of a Cluster

- In a cluster, the rows from all of the tables having the same value for cluster columns are stored in the same page on disk. This reduces the data retrieval time.
- Each distinct value in the cluster columns is stored in the database only once. As a result it saves space (unlike indexes, which requires additional storage space.)
- An index on the cluster column(s) needs to be defined using CREATE CLUSTER command. This index is called the cluster index.

In Cluster tables

They must have a common column with the same data type, size. Such a column is called a cluster column.

The cluster column need not have the same name in each table in a cluster.

Steps in creating a Cluster and its Tables

- Create a Cluster
- Create a Cluster Index
- Create tables in the Cluster

Creating a Cluster

The first step in creating clustered tables.

The CREATE CLUSTER command is used.

Syntax

```
 CREATE CLUSTER clustername ( column
specification, ... );
```

Example: Create a cluster for emp and dept tables

```
SQL> CREATE CLUSTER cl_emp_dept ( deptno
NUMBER(2) );
```

Creating cluster Index

To find a row in a cluster, ORACLE uses the cluster index.

We can create an index on the cluster key columns.

Cluster index is created and stored in an index segment (in similar way like a table index).

A cluster and the cluster index can be stored in different tablespaces.

A cluster index is created using CREATE INDEX command.

Syntax

```
  CREATE INDEX indexname
  ON CLUSTER clustername;
```

Example : Create a cluster index on a cluster cl_emp_dept

```
SQL>  CREATE INDEX cl_idx
    ON CLUSTER cl_emp_dept;
```

The difference between a cluster index and a table index

- A cluster index contains one entry per cluster key value, rather than one entry per row.
- The absence of a table index does not affect users of a normal table.
- In cluster tables data cannot be accessed unless there is a cluster index.
- If a cluster index is dropped, data in the cluster remains but becomes unavailable until a new cluster index is created.

Creating Tables in a Cluster

Existing tables cannot be moved into a cluster.

Tables have to be explicitly created under that cluster.

To create a new cluster table, the CREATE TABLE command is used

Syntax

Syntax 1: Creating a new table in a cluster

(cluster table columns defined with the datatype and size)

```
CREATE TABLE tablename ( column1, column2 ... )
CLUSTER clustername (cluster column,);
```

Example: Create a new emp table under a cluster

```
SQL> CREATE TABLE cl_tbl_emp
   ( empno NUMBER(4) PRIMARY KEY,
   ename   VARCHAR2(10),
     ...
   deptno  NUMBER(2)
   ... )
   CLUSTER cl_emp_dept (deptno);
```

Syntax 2: Creating a cluster table from the existing table

```
CREATE TABLE tablename (column1, column2 , .. )
CLUSTER clustername ( cluster column, .. )
AS query;
```

Example: Create tables cl_emp and cl_dept on cluster cl_emp_dept from the existing emp and dept tables.

Creating a table cl_emp on cluster cl_emp_dept from the existing emp table.

```
SQL> CREATE TABLE cl_emp
   CLUSTER cl_emp_dept (deptno)
```

```
   AS SELECT * FROM emp;
```

Creating a table cl_dept on cluster cl_emp_dept from the existing dept table.

```
SQL> CREATE TABLE cl_dept
     CLUSTER cl_emp_dept (deptno)
     AS SELECT * FROM dept;
```

Dropping Cluster tables

The DROP TABLE command is used to remove the clustered tables.

Syntax

```
 DROP TABLE cl_emp ;
```

The DROP CLUSTER command is used to remove a cluster.

Syntax

```
 DROP CLUSTER clustername ;
```

Example

```
SQL> DROP CLUSTER cl_emp_dept ;
```

Summary

- Indexing and Clustering are the two common ways to improve the performance of the database tables.
- The database index is a sorted list of rows accompanied by the location of the row.
- Indexes are not a must for running ORACLE. If available, they speed up the database operations.
- Clustering is a method of storing data together from different tables that are related and are often joined together while accessing.
- Creating Cluster table includes three steps. These steps are
  - Create a Cluster
  - Create a Cluster Index
  - Create tables in the Cluster
- The absence of a table index does not affect users of a normal table. In cluster tables data cannot be accessed unless there is a cluster index.

❑❑❑

Index

SQL Assignments

Before You Begin

This 'ORACLE' lab manual aims at giving a complete understanding of the various aspects of ORACLE. This manual is divided into 13 lab sessions. Each session is based on an individual topic. You are expected to complete each lab session within the allotted time. Each session is again divided into five parts.

1) Objective

States what you will achieve after completing a particular session. At the end of each lab session you should keep a track whether the objectives of that session are achieved.

2) In short

Gives in short idea of chapter including syntax, which will help you to understand the chapter very well.

3) Problem statement

Gives a list of problems based on that topic. These assignments are compulsory and you are expected to at least complete this part. Hints are provided wherever necessary.

4) Try these things

Gives a list of more assignments, which you should try after completing the first assignments. Completion of these extra assignments ensures that you get the expected clarity of the topic.

5) By the way

Contains useful additional information related to the topic. Many times this section should be read first.

(* Syntax's given in this lab manual are general i.e. only with necessary options.)

You must note that the evaluation will be based on the following factors:

- Your performance in the classroom. This includes your regularity,

initiative and interaction.

- Your performance in the labs. This includes timely completion of assignments and the coding practices followed.
- The test conducted, which will depend on the discretion of the faculty.
- At the beginning of each session the assignments of the previous sessions will be checked.

DATA TYPES FOR ORACLE

DATATYPE	PARAMETERS	DESCRIPTION
VARCHAR2(n)	n= 1 TO 4,000	Text string with variable length
NUMBER(p,s)	p=1 TO 38, s=-84 TO 127 OR Float	Number. specify p(total digits) and s(digits to right of decimal)
DATE	None	Date includes the century, and includes the time in hours, minutes and seconds. Default format is dd-mon-yy. Default time is midnight.
CHAR(n)	n=2000	Text string with fixed length
LONG	None	Carried over from oralce7. Use BLOB, CLOB, NCLOB or BFILE instead.
BLOB,CLOB, NCLOB	None	Three kinds of large objects (LOBS). maximum size is 4 gigabytes
BFILE	None	Large Binary Object (LOB) stored outside the database. maximum size is 4 gigabytes

Following are the structure of tables provided by the oracle.

EMP TABLE

Name	Null?	Type
EMPNO	NOT NULL	NUMBER(4)
ENAME		VARCHAR2(10)
JOB		VARCHAR2(9)
MGR		NUMBER(4)
HIREDATE		DATE
SAL		NUMBER(7,2)
COMM		NUMBER(7,2)
DEPTNO		NUMBER(2)

DEPT TABLE

Name	Null?	Type
DEPTNO		NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)

SALGRADE TABLE

Name	Null?	Type
GRADE		NUMBER
LOSAL		NUMBER
HISAL		NUMBER

Following are the Data of Tables provided by the Oracle -

EMP TABLE

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMIT	CLERK	7902	17-	800		20

	H			DEC-80			
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	09-DEC-82	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7876	ADAMS	CLERK	7788	12-JAN-83	1100		20
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7934	MILLE	CLERK	7782	23-	1300		10

	R			JAN-82			

DEPT TABLE

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

SALGRADE TABLE

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

Editing Commands

APPEND **Syntax :** A[PPEND] TEXT	The APPEND command appends the specified TEXT to the current line.
CHANGE **Syntax :** C[HANGE] /OLD TEXT/NEW TEXT	The CHANGE command changes the old text to the new text in the current line.
DELETE **Syntax :** DEL[ETE]	DELETE command deletes the current line from the SQL buffer.
CLEAR BUFFER **Syntax :** CLEAR BUFFER	CLEAR BUFFER command clears the contents of the SQL buffer.
INPUT **Syntax :** I[NPUT]	INPUT command adds one or more lines to the SQL buffer.
LIST **Syntax :** L[IST]	LIST command lists all the lines in a SQL buffer.

Other Commands

DESCRIBE **Syntax :** DESC[RIBE] table_name	The DESCRIBE command lists the structure of a table.
EDIT **Syntax :** ED[IT] [filename]	The EDIT command opens the operating system text editor and fetches the contents of the file specified using filename. If file is not specified contents of the SQL buffer are displayed.
EXIT	EXIT command exits the

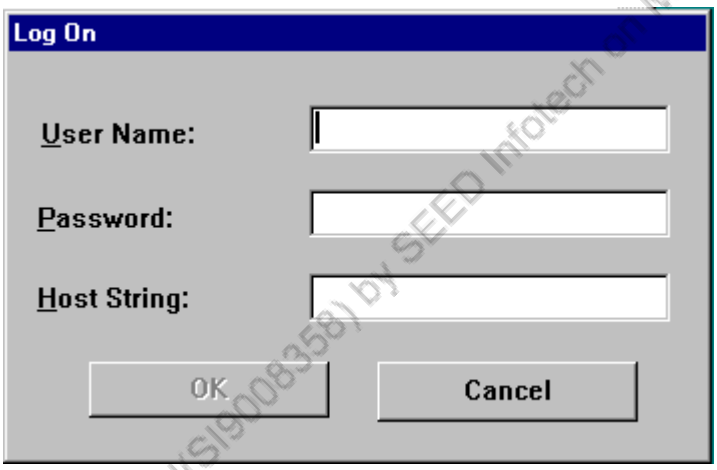
Syntax : EXIT	SQL*PLUS environment and closes the current session.
GET **Syntax :** GET filename	GET command loads the specified operating system file into a SQL buffer.
HOST **Syntax :** HO[ST] command	HOST command executes the specified host operating system command.
START **Syntax :** STA[RT] filename	START command loads the contents of the specified file into the SQL buffer and executes it.

SAVE **Syntax :** SAV[E] filename [REPLACE/APPEND]	SAVE command saves the contents of the SQL buffer to the specified operating system file
RUN **Syntax :** RUN	RUN command executes the command in SQL buffer.
/ (SLASH) **Syntax :** /	/ command runs the command from SQL buffer.
@ ('at the rate' sign **Syntax :** @	@ command loads the contents of the operating system file into the SQL buffer, lists the command and executes the command.

LAB : 1

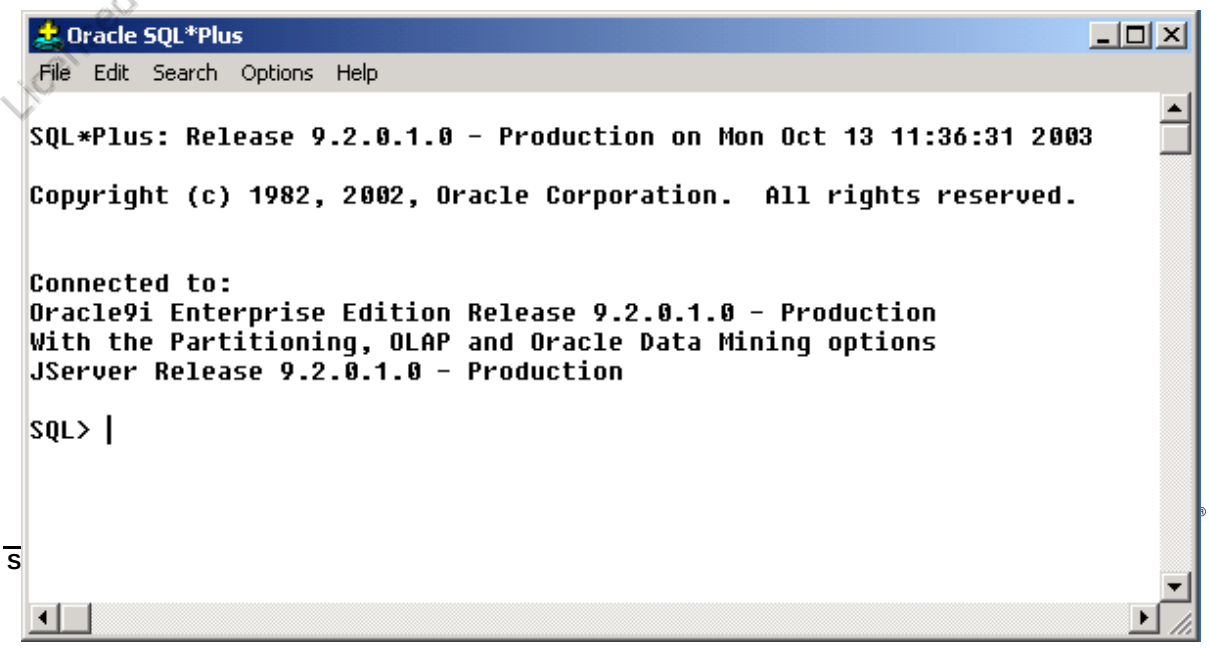
Pre-requisites

- In your assigned folder on network drive create a sub-folder for respective lab sessions. e.g. Z:\ON11_1\Lab1>.
- It is suggested to perform all the exercises in a sequential order and save your respective lab session's work under respective subfolder of assigned folder.
- From **Start** menu of windows select **Programs,** select **ORACLE-OraHome90.** From **ORACLE-OraHome90** select **Application Development** and click on **SQL Plus**.
- SQL*PLUS application will start & following Log On dialog box



will be displayed.

- Type a valid user name, password, host string information, which will be given to you by concern lab faculties.
- SQL*PLUS environment has been invoked.



Data Retrieval using SQL

Objectives

At the end of this exercise, student will be able to

- Identify the components of SQL
- Perform the data retrieval using Structured Query Language commands.
- Use various clauses in the SELECT statement.
- Use various column functions in SQL.

In short

SQL (Structured query Language) is a language used to access data from the database.

SELECT statement has many options. The simplest form of the SELECT statement is:

```
SELECT <column-list>
FROM <table-name>
[WHERE <condition>]
[GROUP BY <column-name(s)>]
[HAVING <condition>]
[ORDER BY <expression>];
```

Problem statements

Solve the following queries, which are based on the EMP and DEPT tables.

1) List all the information about employees in the EMP table.
2) List all the information about department in the DEPT table.
3) List the employee number, name, job title and hired date of employees in department 10.
4) Select name and salary of employees who are clerks.
5) List the department number and name for all departments with department numbers greater than or equal to 20.
6) List the name of the employees having salary less than 2500.
7) Select name, salary and commission of employees whose commission is greater than their salary.

8) List the employee number and name of the President.

9) List the employees who do not get commission.

10) List all the employees in department number 10 other than KING.

11) Display names of the employees whose job is either ANALYST or CLERK.

12) Display different kind of jobs available. (Hint: Use DISTINCT)

13) List names of all employees whose names are 4 characters long.

14) List names of all employees whose names end with letter ‘R’.

15) List names of all employees whose names start with ‘B’ or ‘M’.

16) Retrieve the names and job of the employees working in the department number 20. Display the result with ‘Employee – Job’ as column heading and arranging the columns with ‘-‘ in between like ‘Smith – Clerk’.

17) Select all employees whose names fall between ‘A’ and ‘G’ alphabetical range.
(Hint : use BETWEEN)

LAB : 2

Assignments : ORDER BY Clause

1) List the employee details in ascending order of salary.
2) List the names of employees in department 30 from emp table in the descending order of their salary.
3) Order the results of above query using relative positions of columns.
4) List the employee names and hired date in the descending order of hiredate.

Assignments : Using expressions

5) List name, salary and PF amount of all the employees, PF is calculated as 10% of salary. (Use columns alias)
6) List the names of employees, who are more than 2 years old in the organization. (Use SYSDATE)

Assignments : Aggregate Functions

7) List the number of employees working in the company.
8) List the number of jobs available in the company.
9) List the total salaries payable to employees.
10) List maximum, minimum, average, Sum of salary.
11) List the maximum salary and number of employees working as a salesman.
12) List the average salary and number of employees working in the department 20.
13) List the earliest date and the latest date on which someone was hired.
14) Write a query to count the number of people in department 30 who receive a salary and the number of people who receive a commission.
15) Compute the average, minimum and maximum salaries of those groups of employees having the job of CLERK or

MANAGER.

16) List the department number and the maximum salary earned in department 20.

17) Display 10% increased salary of each employee.

18) Find out the locations of the employees. Sort the list by location.

19) Who was the last employee hired in each department?

20) Find out the people whose salary is less than the average salary for department number 20.

21) List the names of the people who are reporting to BLAKE.

22) How many employees work in NEW YORK.

23) Determine the average earning of an employee working in department 30.

24) Show what length names appear in the EMP table. Eliminate the duplicate lengths from the rows returned.

25) Find out the total salary in each department. Display the department number and the total salary.

Assignments : GROUP BY Clause

26) List the department numbers and number of employees in each department.

27) List the department number and total salary payable to each department.

28) List the jobs and the number of employees in each job. The result should be in the descending order of the number of jobs.

29) List the job wise total salary, average salary, and minimum salary of employees.

30) List the total salary of employees job wise for department 20 only.

31) Find out the total salary in each department. Display the department number and the total salary.

32) Find out maximum salaries department wise excluding those who are having salary less than 3000.

33) List the job wise total salary, average salary of employees of department number 20 and display only those rows having salary greater than 1000.

Assignments : IS NULL operator

34) Give commissions equal to 1% of their salaries to employees having commission as NULL.

35) List the employee names, which are not eligible for commission.

36) List the name of the employee and the job of the employee who does not report to anybody.

37) List the employees not assigned to any department.

38) List the details of employees, whose salary is greater than 2000 and commission is NULL.

Assignments : LIKE operator

39) List the employees whose name starts with ‘S’.

40) List the employee names ending with ‘S’.

41) List the employee names having third character as ‘R’.

42) List the names of employees whose names have exactly five characters.

Assignments : DATE functions

43) Find the day of the week on which SMITH joined.

44) List the names and hired date of the employees in department 20, display the hired date formatted as 21/03/87.

45) List the name of the employee who has joined recently to the organisation.

46) Find out time of the day (hh24, mi, ss) on which FORD joined.

47) Find out the day of the month on which JAMES joined.

48) Find out the quarter of the year the employees joined.

49) How many months did the president work for the company? Round to the nearest whole number of months.

50) List the names, department of all employees whose hired date anniversary does not exist in the first quarter of the year.

51) List employee name, salary and his income group as 'LOW' or 'HIGH' depending on the salary amount. (If the salary is less than 4000 then he is in 'LOW' income group or else in 'HIGH' income group.

Ename	Salary	LOW	HIGH
SCOTT	3000	LOW	
KING	5000		HIGH

(Hint : Use DECODE and SIGN function)

52) Get the same result as shown above using CASE.

53) List employees under their own department name like.

Dept10	Dept20	Dept30
SCOTT		
	ROBERT	
		JOHN

* Assumes there are 3 departments only.

54) List employee name, salary for all employees showing salary in bar chart from

Ename	Salary	Graph
Scott	2000	****
John	6000	**********

Scale : one * for 500

Hint : Use Rpad function.

55) List employee name, salary, commission, total salary for all employees.

56) Write the above same query using NVL2 function.

57) List employee name and his commission. If employee commission is null then display 'N.A.' in place of commission.

LAB : 3

Multi-Table Queries

Objectives

At the end of this exercise, student will be able:

- To classify multi-table queries
- To write multi-table queries
- To understand the use of joins, set operators and sub-queries

In Short

Data from various tables needs to be accessed together .The process of forming rows from two or more tables by comparing the contents of related columns is called as joining tables. Joins are the foundation of multi-table query processing.

SQL handles multi-table queries by matching columns.

Syntax :

```
SELECT columnlist
FROM table1, table2…
[WHERE logical expression];
```

Types of Joins

- Equi-join
- Cartesian join
- Non-equi join
- Self join
- Outer join

(***for more detail information regarding join please refer chapter 'Multi Table Queries' from ORACLE courseware).

Set Operators

The set operators combine the results of one or more queries into one result.

The set operators supported in ORALCE are:

- Union
- Intersect
- Minus

(***for more detail information regarding join please refer chapter 'Multi Table Queries' from ORACLE courseware).

Subqueries

The SQL sub-query is one of the advanced query techniques in SQL. This feature lets us use the results of one query as part of another query.

Sub-queries can be divided into two broad groups

- Single-row sub-query: returns only one value to the outer query.
- Multi-row sub-query: returns multiple values to the outer query.

Operators in multi-row sub-queries

- IN or NOT IN
- ANY or ALL
- EXISTS

(***For more detail information regarding operators in multi-row sub-queries please refer chapter 'Multi Table Queries' from ORACLE courseware).

Problem statements

1) List employee number, name, his department and the

department name.

2) List employee name, his department name and the department location.

3) List employee name, department name for all the clerks in the company.

4) List employee number, name, job, his manager's name, and manager's job.

5) Display different designations in department 20 and 30.

6) List the jobs common to department 20 and 30.

7) List the jobs unique to department 20.

8) List the employees belonging to the department of 'MILLER'.

9) List all the employees who have the same job as 'SCOTT'.

10) Display the names of the employees who are working in Sales or Research department.

11) Display name and salary of the employee who is working in CHICAGO.

12) List the details of employees in department 10 who have the same job as in department 30.

13) List all the departments that have employees who exist in them.

14) List the employee details if and only if more than 10 employees are present in department 10.

15) List the employee names whose salary is greater than the lowest salary of an employee belonging to department 10.

16) List the employee names whose salary is greater than the highest salary of an employee belonging to department 20.

17) List the names of the employees drawing the highest salary.

18) List the employees whose salary is second highest in the company.

19) List the details about employees who have maximum

number of people reporting to them.

20) List the employees who earn more than the average salary in their own department.

21) List the employee name, length of his name, his manager's name whose name length is greater than their managers name length.

22) List employees and his managers' details, where that employee's salary is greater than his managers' salary.

23) List employees whose salary is highest in their department.

24) List the locations of all the departments and the employees working in them including the departments without employees.

25) List information of all employees along with information of all the departments.

26) What is the length of the longest employee name and by how many characters is it longer than its nearest one.

27) Find out the difference between the maximum salary earned by a person in department number 10 and minimum salary earned by a person in department number 30.

28) Find out the difference between average earnings of department no 30 and 40.

LAB : 4

Data Manipulation

Objectives

At the end of this exercise, student will be able to

- Demonstrate the data manipulation language commands.

In Short

Data Manipulation Language

The data manipulation language is used for query insertion, updating and deletion of data from the database. This includes commands like INSERT, UPDATE and DELETE.

- INSERT statement

  The INSERT command lets us to add a row of information into the table.

  Syntax :

```
INSERT INTO tablename [(column1, column2, ...)]
VALUES(value1, value2, ...);
```

  The general syntax of INSERT statement using query is:

```
INSERT INTO tablename [(column1, column2, ...)]
SELECT column1, column2, ...
FROM othertable
[ WHERE condition ];
```

- UPDATE statement

  The UPDATE command is used to modify column values in a table. Values of one or more columns can be updated.

  Syntax :

```
UPDATE tablename
SET column = expression or value
[, coulmn2 = expression, ...]
```

```
[ WHERE conditions ];
```

- DELETE statement

  The DELETE command is used to remove rows from a table. The entire row is deleted form a table and only specific columns cannot be deleted from a table. A set of rows can also be deleted from a table by specifying the condition using the WHERE clause.

  Syntax :

```
DELETE [FROM ] tablename
[WHERE condition];
```

- TRUNCATE command

  The TRUNCATE command can also be used to remove all the rows from the table.

  Syntax :

```
TRUNCATE FROM tablename;
```

Problem statements

1) Insert a new employee with following details

   Employee number 7987

   Employee name 'BILL'

   Salary 2500

   Department number 30

2) Raise the salary of all the salesman by 10%.
3) Give a rise of 15 % to all the employees from the ACCOUNTING department.
4) Delete the details of all the employees whose salary is less than 1000.
5) Give as increment of 500 to all the MANAGERs in the company.
6) List all the employees to whom three employees report, should report to PRESIDENT.
7) Update the salary to 5000, for all the employees to whom the highest number of people report.

LAB : 5

Data Definition Language

Objectives

At the end of this exercise, student will be able to:

- Create tables using different options
- Apply the constraints.
- Modify the existing definition of the table.

In Short

The Data Definition Language (DDL) consists of a set of commands used to create the database objects such as tables, views, indexes etc. This includes commands like CREATE, ALTER and DROP.

Creating Database Tables

Syntax :

```
CREATE TABLE [schema.]tablename(
Column1 datatype (size) [DEFAULT <expr> ]
[CONSTRAINT constraint_name][ENABLE/DISABLE],
column2 datatype….,
...
[CONSTRAINT constraint_name]
[table_constraint][ENABLE/DISABLE],
...
);
```

Creating database tables using subquery

Syntax :

```
CREATE TABLE tablename(column definition)
AS query;
```

Modifying the Table Structure

After a table is created and has been in use, we might want to make changes to its structure. We may change the structure of a table in following ways:

- Add one or more columns
- Redefine the existing column definitions
- Redefine the integrity constraints.

The ALTER TABLE command is used to modify the table structure.

(****For more detail please refer courseware),

Integrity Constraints

The integrity constraints help us to enforce business rules on data in the database. Once we specify an integrity constraint for a table, all data in the table always conforms to the rule specified by the integrity constraint.

The constraint that can be defined while creating database tables are as follows:

- NULL, NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY, REFERENCES, ON DELETE CASCADE
- CHECK

Problem Statements

1) Create tables NEW_EMPLOYEE and NEW_DEPARTMENT using the existing table definitions for EMP and DEPT tables (HINT : use subquery with create table statement)
2) Write the above table creation scripts so that the new table structures will just be created and the data of existing tables is not copied to the new tables.
3) During the assignments we have been using EMP and DEPT tables. Write SQL statements to create these tables with appropriate constraints and with appropriate constraint names using DDL.

4) Modify the definition of an NEW_EMPLOYEE table by adding two columns employee address and his telephone number. Use appropriate data type and size.
5) Create a table INCREMENT to store the increment details of an employee. This table stores the employee number, the date of increment and the amount of increment.
6) Create table BONUS to store the bonus details of an employee. This table stores the employee number, the date of bonus and the bonus amount.
7) Write a query to disable the PRIMARY KEY and FOREIGN KEY constraints of an EMP table.
8) Add a constraint to the EMP table to ensure that the employee name is always entered in all caps(i.e. all capital letters).
9) Add a constraint to the EMP table to ensure that the salary is always in the range 3000 to 10000.

LAB : 6

Views, Synonyms And Sequences

Objectives

At the end of this exercise, student will be able to:

- Explain creation and the use of views.
- Identify the use of synonyms.
- Identify the use of sequences.

Views

A VIEW is a virtual table in the database. The contents of a view are defined by a query. A VIEW can represent a subset of the data in a table. The data from multiple tables can be also combined together using a view. VIEWs are stored in the data dictionary in the table called USER_VIEWS. They're two types of views Read-only views and Updateable views.

Creating A View

Syntax :

```
CREATE [OR REPLACE][FORCE/NOFORCE] VIEW view_name
AS query;
```

Synonym

A SYNONYM is a simple alias for a table, view, sequence or other database objects. Because a SYNONYM is just an alternate name for an object it requires no storage space. ORACLE stores only definition of a SYNONYM in the data dictionary. ORACLE allows us to create both, Public or Private SYNONYM.

Creating A Synonym

Syntax :

```
CREATE [PUBLIC] SYNONYM [schema.]synonym
FOR[schema.]object;
```

Deleting a Synonym

Syntax :

```
DROP [PUBLIC] SYNONYM [schema.]synonym;
```

Sequence

A SEQUENCE is a database object used to generate the series of unique integers for use as primary keys. When an application inserts a new row into a table, the application simply requests a database sequence to provide the next available value in the SEQUENCE for the new row.

Creating a Sequence

Syntax :

```
CREATE SEQUENCE [schema.]sequence_name
[INCREMENT BY n]
[START WITH n];
```

Using a Sequence

Syntax :

```
Sequence_name.CURRVAL
```

Returns the current value in the sequence

```
Sequence_name.NEXTVAL
```

Increments the current value in the sequence and returns it.

Deleting a SEQUENCE

Syntax :

```
DROP SEQUENCE [schema.]sequence_name;
```

Problem Statements

8) Create a view EMPVIEW, which will contain the employee number, employee name, salary, his department number and department name.

9) Create the same view with new column names as eno, empname, salary, dno, deptname.

10) List the details of employees working in department 30 using above view.

11) Create a view containing employees working as a SALESMAN.

12) Add a new salesman using view.

13) Create a view for the increment table containing number of increments and total increment amount for employees.

14) List the increment details of all the employees using view.

15) List the employee details along with the increment details using view.

16) Create a synonym DEPARTMENT for dept table.

17) Create a sequence for generating product numbers starting with 100.

LAB : *7*

Indexing And Clustering

Objectives

At the end of this exercise, student will be able to:

- Create indexes
- Create cluster tables

In Short

Once the application starts working, what user demands is the performance. The database needs to get tuned to get the better performance. Indexing and clustering are the two common ways to improve the performance of the database tables.

Index

- The use of the index is the primary method to reduce the disk I/O and improve the performance of the database. ORACLE can quickly find specific records using index.
- Indexes are useful only for the key columns that application refers frequently to find specific rows.

General syntax for creating an index

Syntax :

```
CREATE [UNIQUE] INDEX [schema.]indexname
ON tablename(column1,column2…);
```

General syntax for dropping an index

Syntax :

```
DROP INDEX [schema.]indexname;
```

Cluster

Clustering is a method of storing data together from different tables that are related and are often joined together while accessing. Clustering can boost the performance of join queries. This is because the rows that are joined are stored together.

If a group of tables is to be clustered then they must have a common column with the same data type, size. Such a column is called a cluster column. The cluster column need not have the same name in each table in a cluster.

Creating a Cluster

Syntax :

```
CREATE CLUSTER clustername (column
specification,…);
```

Creating a cluster index

Syntax :

```
CREATE INDEX indexname
ON CLUSTER clustername;
```

Creating tables in a cluster

Syntax :

```
CREATE TABLE tablename (column1,column2,…)
CLUSTER clustername(cluster column,);
```

Dropping cluster tables

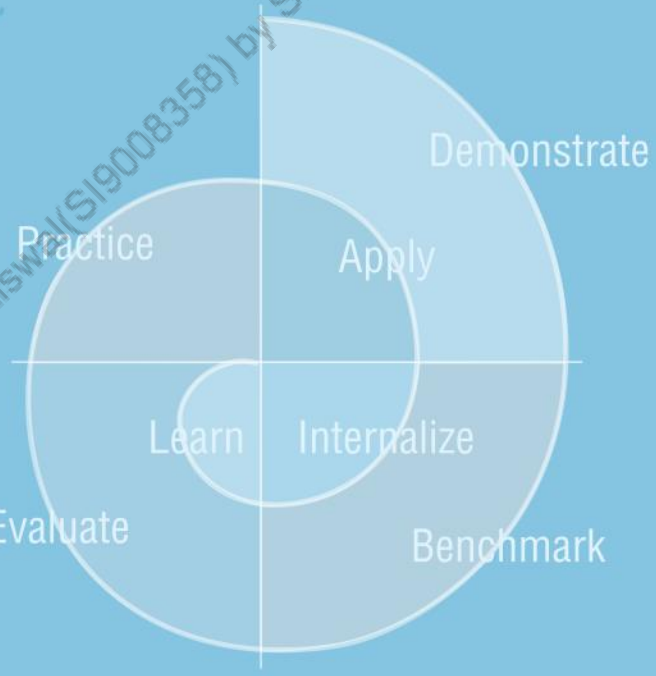
Syntax :

```
DROP CLUSTER clustername;
```

Problem statements

1) Create unique index for EMP table on employee name column.
2) Create an index on deptno and dname on DEPT table.

Write DDL commands to create cluster tables cl_emp, cl_dept (completely) in a cluster cl_emp_dept (and not using existing tables).



Avani publications

N·S·D·C National Skill Development Corporation
Transforming the skill landscape
Training Partner

seed
Official Curriculum