# Chapter 2

# Introduction to SQL *Plus

## Learning Objectives

***After completing this chapter, you will be able to:***

- *Understand SQL *Plus buffer commands.*
- *Understand various data types.*
- *Understand various types of constraint.*
- *Create a table.*
- *Modify and delete a table.*



# INTRODUCTION TO SQL

SQL (pronounced as "ess-que-el") stands for Structured Query Language. It is a specialized non-procedural language used to communicate with a database. The statements of SQL are used to perform various tasks such as inserting, updating, or retrieving data from a database. According to ANSI (American National Standards Institute), SQL is a standard language for the relational database management system. A variety of established database products support SQL, including the products of Oracle and Microsoft. Unfortunately, there are many different versions of SQL, but according to ANSI, they must support the same major keywords in a similar manner such as **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **WHERE**, and so on. The standard SQL commands such as **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **CREATE**, and **DROP** can be used to work with a database.

This chapter will describe the basics of each of these commands and allow you to put them for practice using the SQL Interpreter.

## History of SQL

The model of RDBMS (Relational Database Management System) was first introduced by Dr. E. F. Codd (Dr. Edgar Frank Codd). In June 1970, Codd published a paper "A Relational Model of data for Large Shared Data Banks", which was later accepted as the model for RDBMS. The first version of SQL was developed in the early 1970s. This version, initially called SEQUEL, was designed to manipulate and retrieve the stored data. Later, the SQL language was standardized by American National Standards Institute (ANSI) in 1986. The subsequent versions of the SQL standard were released as per the norms of International Organization for Standardization (ISO). Later in 1979, Relational Software Corporation, now known as Oracle Corporation, introduced SQL as the first commercial database language. Since then, this language has been accepted as the standard RDBMS language.

## Introduction to SQL *Plus

SQL *Plus is an extension of the standard SQL and has an online command interpreter. SQL *Plus program allows you to store and retrieve data in the Relational Database Management System. It is frequently used by the database administrators and developers to interact with the Oracle database system. It is an interface for SQL and PL/SQL languages. SQL *Plus is a reporting tool that is used as an interface between the Client and the Server of Oracle database. Using SQL *Plus, a user can create program files and generate the formatted reports.

SQL *Plus is used by the application developers to:

1. Create and modify the database.

2. Create, replace, alter, and drop objects.

SQL *Plus is used by the end-users to:

1. Query the data.

2. Retrieve data from the database.

SQL *Plus is used by the Database Administrators to:

1. Create users.
2. Specify rights and privileges to users.
3. Monitor the database.
4. Control the access to the database and its objects.
5. Maintain consistency and integrity of the database.
6. Maintain Backup and Recovery of database.
7. Maintain Performance and Tunning of database.

## Loading SQL *Plus

The following steps are required to start SQL *Plus:

1. Choose **Start > Programs > Oracle 11g > Application Development > SQL Plus** from the task bar; the **Log On** window will be displayed, as shown in Figure 2-1.

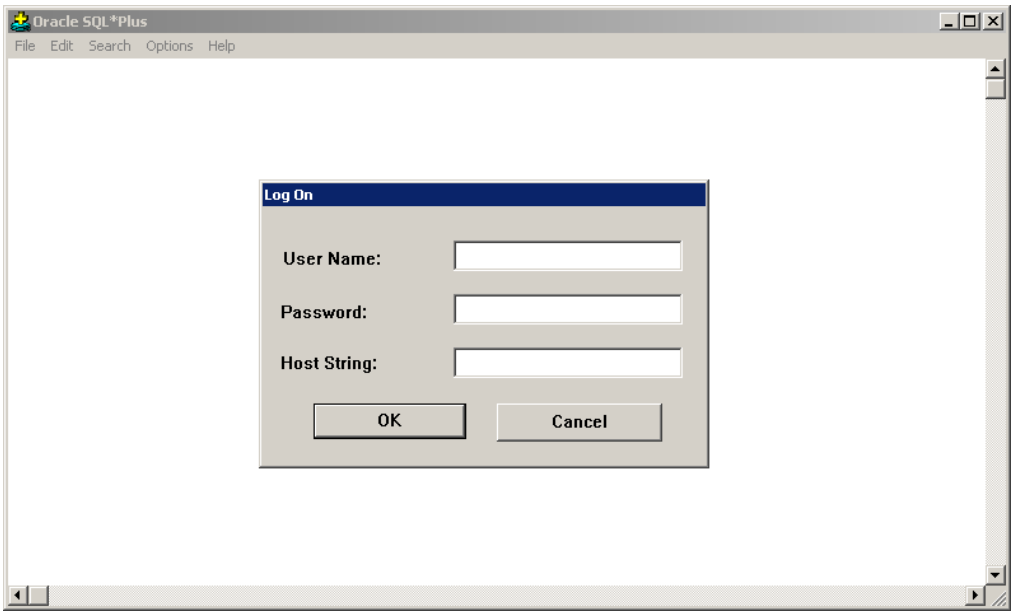



***Figure 2-1*** *The **Log On** window of SQL *Plus*

2. In this window, enter the user name, password, and host string in their corresponding edit boxes. If you are using Oracle Personal Edition, leave the **Host String** edit box blank. Next, choose the **OK** button in this window; the **Oracle SQL *Plus** window will be displayed, as shown in Figure 2-2.

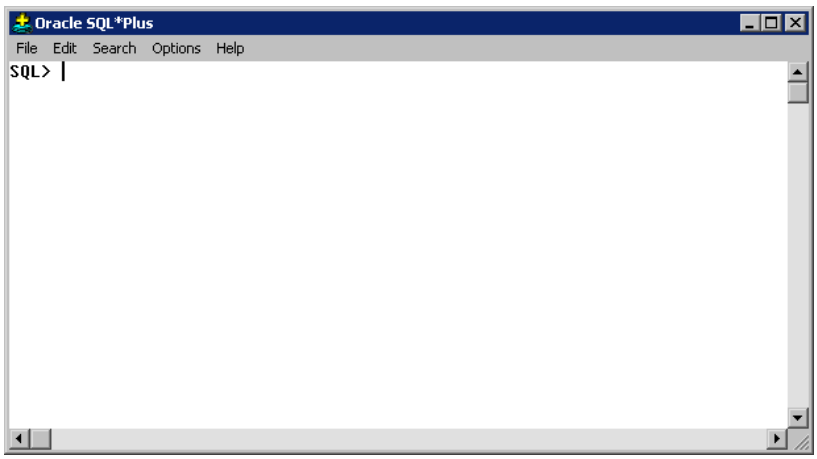



***Figure 2-2*** *The **Oracle SQL *Plus** window*

**SQL>** shown in Figure 2-2 is called the SQL command line or the SQL prompt of SQL *Plus.

## Exiting from the Oracle SQL *Plus Window

You can exit **Oracle SQL *Plus** window in three ways. These are as follows:

1. Enter **EXIT** or **QUIT** at the SQL command window and then press ENTER.

2. Choose **File > Exit** from the menu bar.

3. Next, choose the **Close** button from the top right corner of the **Oracle SQL *Plus** window title bar.



**Note**
***EXIT** or **QUIT** is not case-sensitive.*

## SQL *Plus Buffer Commands

In SQL *Plus, when you enter a statement, the statement is stored in the memory. This memory is referred to as SQL buffer or command buffer. When you enter another statement, the first statement is replaced with the new one and all the entered inputs are stored as a single SQL *Plus statement in the command buffer. If you press ENTER while entering SQL statement in SQL *Plus, the control will be transferred to the new line. However, if the previous line is ended with a semicolon or single slash, the SQL statement will be executed. SQL *Plus has provided some buffer commands that are discussed next.

## L[IST]

The **List** or **L** command is used to display the content of the SQL buffer. The syntax for using the **List** or **L** command is as follows:

```
SQL> LIST or L
```

If the command is a single line command, the line itself will be the current line. In the multi-line command, by default, the last line will be the current line. The current line is marked by the * sign.

For example:

```
SQL>SELECT * FROM Employee;
SQL>LIST
```

The result of the second command will be as follows:

```
1 * SELECT * FROM Employee;
```

Here, the first command line was stored in the buffer. As a result, the second command line will display the contents of the SQL buffer.

## Making a Line as the Current Line

To make a line as the current line, type the line number at the SQL command window and press ENTER; the specified line will become the current line.

For example:

```
SQL>2
```

The given command will make the second line in SQL *Plus window as the current line.

## I[NPUT]

The **INPUT** or **I** command is used to add lines to the existing command or the current command in the buffer. The syntax for using the **INPUT** command is as follows:

```
SQL>INPUT text or I text
```

In the above syntax, **text** is the text or string that you want to add to the existing command.

For example:

```
SQL>SELECT * FROM Employee
```

To add one or more lines to the above SQL query, enter the following statement:

```
SQL>INPUT WHERE Salary>20000;
```

The result of the above SQL query will be same as that of the following query:

```
SQL>SELECT * FROM Employee
WHERE Salary>20000;
```

## DEL

The **DEL** command is used to delete the current line from the buffer. This command is used alone or with * to delete the current line.

The syntax for using the **DEL** command is as follows:

```
SQL>DEL
```

For example:

Enter the following command to view the buffer data and to delete the current line in the buffer:

```
SQL> L
1  SELECT * FROM Employee
2* WHERE Empno = 08
SQL> DEL
SQL> L
```

The output after deleting the specified line from the buffer is as follows:

```
1* SELECT * FROM Employee
SQL>
```

You can get the same result by using the * with the **DEL** command as follows:

```
SQL>DEL *
```

The **DEL** command can also have the following syntax:

```
DEL m n
```

The **DEL m n** command is used to delete lines from **m** through **n**. If you substitute * for **m** or **n**, it will imply the current line.

The following command will delete the specified line from the buffer:

```
1 SELECT First_name, Salary, Start_Date
2  FROM Employee
3*  WHERE Empno = 08
SQL> DEL 2
SQL> L
```

The output after deleting the second line from the buffer is as follows:

```
1  SELECT First_name, Salary, Start_Date
2* WHERE Empno = 08
SQL>
```

The following command will also delete the specified line from the buffer:

```
1 SELECT First_name, Salary, Start_Date
2  FROM Employee
3*  WHERE Empno = 08
SQL> DEL 2 *
SQL> L
```

The output after deleting the specified line from the buffer is as follows:

```
1* SELECT First_name, Salary, Start_Date
SQL>
```

The **DEL LAST** command will delete the last line from the buffer:

```
SQL> L
1  SELECT First_name, Salary, Start_Date
2  FROM Employee
3* WHERE Empno = 7698
SQL> DEL LAST
SQL> L
```

The output after deleting the specified line from the buffer is as follows:

```
1  SELECT First_name, Salary, Start_Date
2* FROM Employee
SQL>
```

## A[PPEND]

The **APPEND** or **A** command is used to append more statement lines to the current line. The syntax for using the **APPEND** command is as follows:

```
SQL>APPEND text or A text
```

In the above syntax, **text** is the text or string that you want to append to the current line.

For example:

```
SQL>A; or APPEND;
```

The above command will add the (**;**) semicolon at the end of the current line.

## C[HANGE]

The **CHANGE** or **C** command is used to find and replace the string in the current line of the SQL buffer. The syntax for using the **CHANGE** command is as follows:

```
CHANGE/Old_Value/New_Value or C/Old_Value/New_Value
```

In the above syntax, **Old_Value** is the existing value in the command line, whereas **New_Value** is the new value, which replaces the **Old_Value.**

For example:

If you want to change the first occurrence of **Empno** to **Emp_No**, enter the following SQL statement in SQL *Plus window:

```
Select Empno, Emp_Name from Emp;

SQL>C/Empno/Emp_No
```

The above command will change the first occurrence of **Empno** to **Emp_No**.

## / (BACKSLASH)

The / (**Backslash**) command is used to execute the current command in the SQL buffer. The syntax for using the backslash is as follows:

```
SQL>/
```

## SAV[E]

The **SAVE** or **SAV** command is used to save the command line in a file for future use.

The syntax for using the **SAVE** command is as follows:

```
SQL>SAVE File_Name or SQL>SAV File_Name
```

The above command creates the file **File_Name** with the extension *.sql*.

For example:

```
SQL> SAVE Info
```

The above command will save the command lines to a file **Info** with the extension *.sql*.

The syntax for using the **SAVE** command to create, append, or replace data in the existing file is as follows:

```
SQL>SAVE File_Name [option]
```

The above command stores the command line to the file **File_Name** and the **option** can have the following possible options: **CRE[ATE]**, **APP[END]**, **REP[LACE]**.

For example:

```
SQL> SAVE Info APP
```

The above command will add the command lines to the existing file **Info**.

### REP[LACE]

The **REPLACE** or **REP** command is used to overwrite the command lines on the existing file. The syntax for using the **REPLACE** command is as follows:

```
SQL>SAVE File_Name REP[LACE]
```

In the above syntax, **File_Name** is the name of the file in which you want to overwrite the command lines.

### GET

The **GET** command is used to read the command lines from the SQL file and insert into the buffer. The syntax for using the **GET** command is as follows:

```
SQL>GET File_Name
```

In the above syntax, **File_Name** is the name of the file from where you want to read the command lines.

### START

The **START** command is used to load and execute the specified file of SQL *Plus commands. The syntax for using the **START** command is as follows:

```
SQL>START File_Name
```

In the above syntax, **File_Name** is the name of the specified file from where you want to load and execute the command lines.

### ED[IT]

The **EDIT** or **ED** command is used to edit the command lines or the contents of the SQL buffer or the existing file. The syntax for using the **EDIT** command is as follows:

```
SQL>EDIT File_Name
```

In the above syntax, **File_Name** is the name of the file whose contents you want to edit.



**Note**

*If **File_Name** has the extension .sql, there is no need to write the file name with extension; but if it is other than this, extension has to be specified.*

## @ ('at' sign) or @@ (double 'at' sign)

The @ or @@ command is used to execute the commands saved in the SQL file. The SQL file is a normal text file, and is created using Notepad. The file can be called from the local system or from the web server. The syntax for using the @ command is as follows:

```
@{File_Name [.ext]}
```

The syntax for using the @@ command is as follows:

```
@@{File_Name [.ext]}
```

In the above syntax, **File_Name** is the name of the file, whose contents you want to execute. Note that you need to specify the path of that particular file.

For example:

For using the @ and @@ commands, you need to perform the following steps:

1. Enter the following statement in the Notepad editor and save this file in the *C:\Oracle_11g\c02_oracle_11g* with file name as *sqls.txt*.

   ```
   SELECT Empno, Ename, Job, HireDate, Sal, Comm
   FROM Employee;
   ```

2. Execute the contents of the file by using the @ or @@ command. To do so, enter the following statement in SQL *Plus:

   ```
   @@"C:\Oracle_11g\c02_oracle_11g\sqls.txt"
   ```

   or

   ```
   @"C:\Oracle_11g\c02_oracle_11g\sqls.txt"
   ```

   The above command will execute the contents of the file *sqls.txt* and list all rows of the **Employee** table, as shown in Figure 2-3.

## RUN

The **RUN** command is used to list and execute the commands stored in the SQL buffer. The syntax for using the RUN command is as follows:

```
SQL>RUN
```

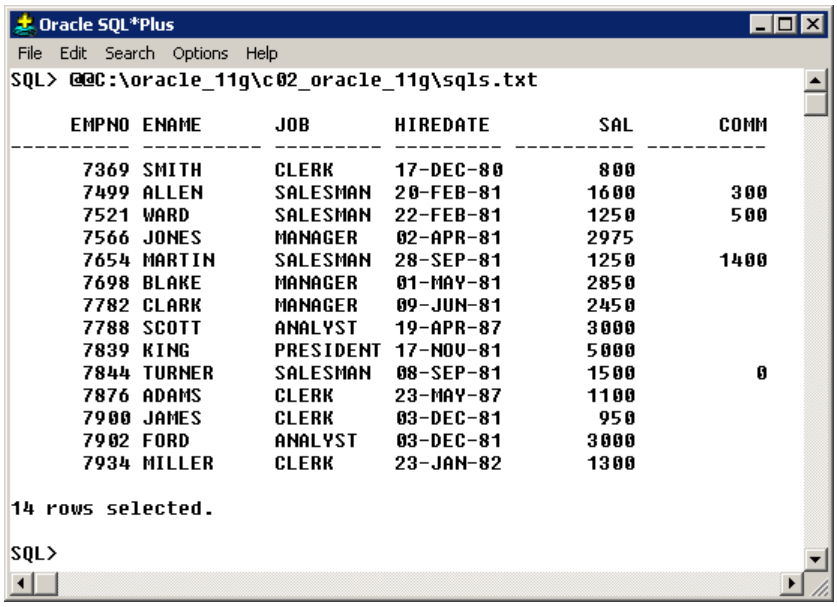



***Figure 2-3*** *Data of the* ***Employee*** *table displayed on using the @@ command*

## DESC[RIBE]

The **DESCRIBE** or **DESC** command is used to view the information about the objects of the Oracle database such as tables, views, and so on. When you use the **DESCRIBE** command with a table or a view, it gives the information such as column name, data type, width of data column. Also, it gives information about each column of a table whether it will allow NULL or NOT NULL value. When you use the **DESCRIBE** command with a procedure, function, or package, you will get information like name, data type, mode **IN/OUT**, and default values of arguments.

The syntax for using the **DESC** command is as follows:

DESC Object_Name

In the above syntax, **DESC** is the keyword and **Object_Name** is the name of a table, view, type, function, procedure, package, or synonym that you want to describe.

For example:

To view the structure of the **Emp** table, enter the following command:

SQL> DESC Emp

The output of this command will be as follows:

| 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)
```

Also, if you want to view the information regarding the package DBMS_OUTPUT, enter the following command in SQL *Plus (SQL prompt):

```
SQL>DESC DBMS_OUTPUT
```

The output of this command will be as follows:

```
PROCEDURE DISABLE
PROCEDURE ENABLE
Argument Name                  Type                    In/Out  Default?
------------------------------ ----------------------  ------  --------
BUFFER_SIZE                    NUMBER(38)              IN      DEFAULT
PROCEDURE GET_LINE
Argument Name                  Type                    In/Out          Default?
------------------------------ ----------------------  ------          --------
LINE                           VARCHAR2                OUT
STATUS                         NUMBER(38)              OUT
PROCEDURE GET_LINES
Argument Name                  Type                            In/Out  Default?
------------------------------ ----------------------          ------  --------
LINES                          TABLE OF VARCHAR2(255)          OUT
NUMLINES                       NUMBER(38)                      IN/OUT
PROCEDURE NEW_LINE
PROCEDURE PUT
Argument Name                  Type                    In/Out  Default?
------------------------------ ----------------------  ------  --------
A                              VARCHAR2                IN
PROCEDURE PUT_LINE
Argument Name                  Type                    In/Out  Default?
------------------------------ ----------------------  ------  --------
A                              VARCHAR2                IN
```

## CL[EAR] BUFF[ER]

The **CLEAR BUFFER** command is used to clear the SQL buffer. This command deletes all lines from the buffer. The syntax for using the **CLEAR BUFFER** command is as follows:

```
SQL> CLEAR BUFFER or CL BUFF
```

For example:

List the contents of the SQL buffer by entering the following command:

```
SQL> L
```

The output of this command will be as follows:

```
1 SELECT empno, ename
2* FROM emp WHERE empno = 7629;
```

Now, enter the **CLEAR BUFFER** command to clear the SQL buffer:

```
SQL> CL BUFF or CLEAR BUFFER
```

After executing the above command, a message buffer cleared will be displayed, confirming that the buffer has been cleared.

Again, list the contents of the SQL buffer; a message **No lines in SQL buffer** will be prompted, as shown below.

```
SQL> L
No lines in SQL buffer.
SQL>
```

## SPOOL

The **SPOOL** command is used to direct the output from the SQL command line to a disk file. This enables you to save the output for future review. The syntax for using the **SPOOL** command is as follows:

```
SQL>SPOOL file_name
```

To start spooling the output into an operating system file, you need to enter the **SPOOL** command followed by the corresponding file name.

For example:

```
SQL> SPOOL my_log_file.log
```

The given command will create a new file named *my_log_file.log*.

The following command will append the output to the existing file *my_log_file.log:*

```
SQL> SPOOL my_log_file.log APPEND
```

The following command will stop the spooling and close the file:

```
SQL> SPOOL OFF
```

### afiedt.buf

*afiedt.buf* is the default edit file of SQL *Plus. When you execute the **ED** or **EDIT** command without arguments, the last SQL or PL/SQL statement will be saved in this file and the file will open in the default editor, as shown in Figure 2-4.

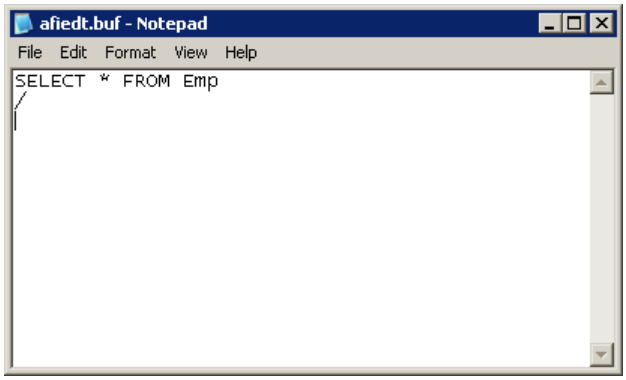



***Figure 2-4*** *The default editor **afiedt.buf** of SQL *Plus*

The following example will illustrate the use of the **INPUT** and **SAVE** commands to save SQL *Plus commands in a file.

## Example 1

Write queries to illustrate the use of the **INPUT** and **SAVE** commands to save commands in a file using the **Emp** table.

1. To compose and save the SQL query using the **INPUT** command, you need to clear the buffer by entering the following command in SQL *Plus.

```
SQL> CLEAR BUFFER
```

2. Now, enter the **INPUT** command to enter commands:

```
SQL> INPUT
1  SELECT Empno, Ename, Sal, Comm
2  FROM Emp
3  WHERE Job = 'SALESMAN'
4  ORDER BY Sals
5
```



**Note**

*Make sure you do not enter a semicolon at the end of the command.*

3. Enter the **SAVE** command in SQL *Plus to store the query in a file called **empRep** with the extension SQL, as shown in Figure 2-5.

```
SQL> SAVE empRep
Created file empRep.sql
```

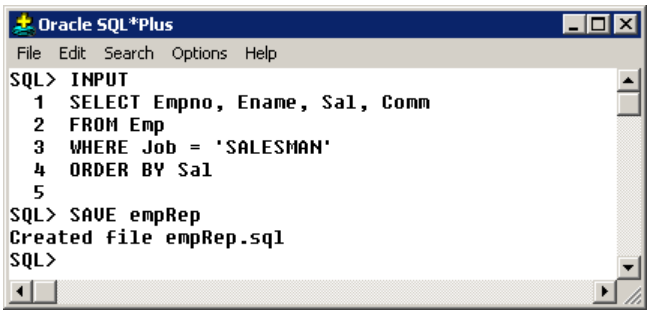



***Figure 2-5*** *Saving commands using the* ***INPUT*** *and* ***SAVE*** *commands*

4. Next, to check whether commands are saved in the file *empRep.sql*, enter the **START** command in SQL *Plus.

```
SQL>START empRep
```

The output of the above command is shown in Figure 2-6.

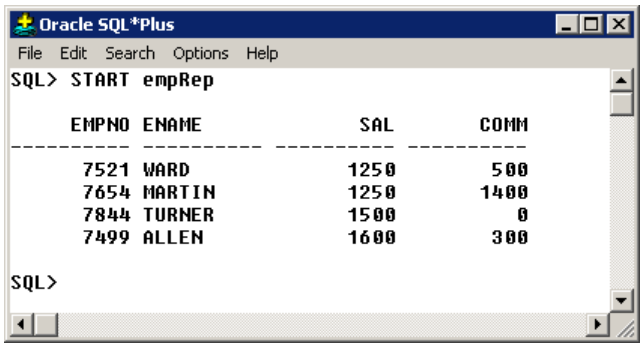



***Figure 2-6*** *Running a command file using the* ***START*** *command*

The following example will illustrate the use of the **APPEND** and **LIST** commands to append text to a command line.

## Example 2

Write a query to illustrate the use of the **APPEND** and **LIST** commands by using the **Emp** table.

The following steps are required to add text the end of a line in the buffer by using the **APPEND** command:

1. Enter the **GET command** to open the file *empRep.sql*, as shown in Figure 2-7.

```
SQL>GET empRep
```

2. Now, to append a space and the clause **DESC** to line 4 of the current query, first you need to list line 4 as the current line by entering the following command in SQL *Plus, refer to Figure 2-7.

```
SQL>LIST 4
4* ORDER BY SAL
```

3. Next, enter the following command to add the space and the clause **DESC** to the end of the current line, refer to Figure 2-7.

```
SQL>APPEND  DESC
4* ORDER BY SAL DESC
```

4. Now, enter the **RUN** command to verify the query, refer to Figure 2-7.

```
SQL>RUN
1  SELECT Empno, Ename, Sal, Comm
2  FROM Emp
3  WHERE Job = 'SALESMAN'
4* ORDER BY Sal DESC
```

The **RUN** command will list the contents of the buffer and execute them refer to Figure 2-7.

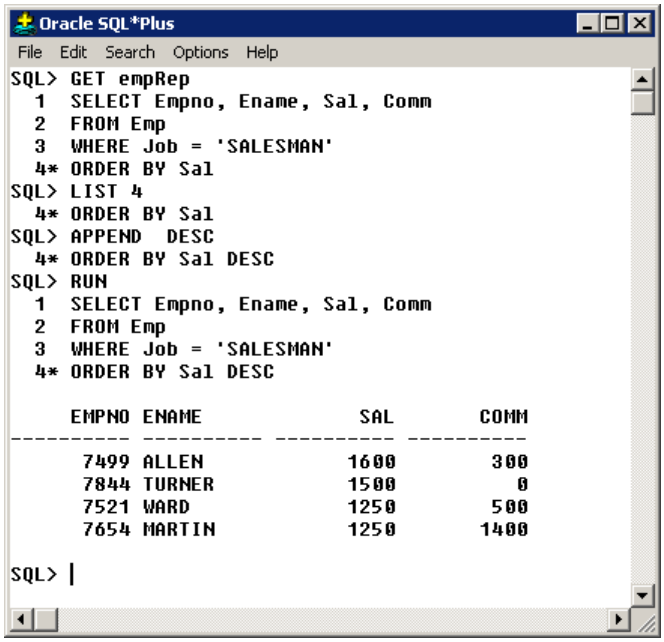



***Figure 2-7*** *Appending text to a line using the* ***APPEND*** *command*

## Comments within SQL Statements

Comments can make your application easy to read and maintain. For example, you can add a comment in a statement that describes the purpose of that statement in your application. Note that the comments added within the SQL statements do not affect their execution.

A comment can appear between keywords, punctuation marks, or parameters in a statement. You can add a comment in a statement in two ways:

1. The comment begins with a slash and an asterisk (/*) and ends with an asterisk and a slash (*/). The comment text can have multiple lines. The opening and terminating characters need not to be separated from the text by a space or a line break.

2. The comment begins with -- (two hyphens) and ends with a line break. The comment text in this case cannot have multiple lines.

# CUSTOMIZING THE SQL *PLUS ENVIRONMENT

You can customize the SQL *Plus environment setting the environment variables as per your convenience or requirement. SQL *Plus has a set of environment variables that control the way SQL *Plus displays data and assigns special characters.

The commands that are used to modify the environment variables are discussed next.

## SET Command

The **SET** command is used to customize or alter the environment of SQL *Plus for the current session by changing the values of environment variables.

For example, you can use this command to set the display width for data, customize HTML formatting, enable or disable printing of column headings, and also set the number of lines per page and page size.

The syntax for using the **SET** command is as follows:

```
SET Variable Value
```

Table 2-1 lists various environment variables that are commonly adjusted using the **SET** command.

## SHOW Command

The **SHOW** command is used to display the current value of variables from the SQL *Plus environment setting. The variables used with the **SET** command can also be used with the **SHOW** command, refer to Table 2-1. For example, the following statement displays the current value of **PAGESIZE** and **LINESIZE**:

```
SQL>SHOW PAGESIZE LINESIZE
```

The output of this command will be as follows:

```
pagesize 14
linesize 80
```

| Environment Variable | Description |
|---|---|
| ARRAY[SIZE] {15\|n} | Sets the size of the data that SQL *Plus will fetch from the database at one time |
| AUTO[COMMIT] {OFF\|ON\|IMM[EDIATE]\|n} | Whenever a change is made in the database by SQL or PL/SQL statements, Oracle will automatically save the change. |
| AUTOT[RACE] {OFF\|ON\|TRACE[ONLY]} [EXP[LAIN]] [STAT[ISTICS]] | Displays a trace report on the successful execution of DML statements |
| COLSEP {_\|text} | Sets the text to be printed between the selected columns |
| DEF[INE] {'&'\|c\|OFF\|ON} | Sets the character used to prefix substitution variables to **c** |
| ECHO {OFF\|ON} | Controls whether the command will be displayed when it is run by the **START** or @ command |
| EDITF[ILE] filename[.ext] | Sets the default file name for the **EDIT** command |
| EMB[EDDED] {OFF\|ON} | Sets the report feature on or off |
| ESC[APE] {\|c\|OFF\|ON} | Defines the character that is entered as escape character |
| FEED[BACK] {6\|n\|OFF\|ON} | Displays the number of records returned by a query when the query selects at least **n** records |
| FLAGGER {OFF\|ENTRY\|INTERMED [IATE]\|FULL} | Ensures that the SQL statements confirm to the ANSI/ISO SQL92 standard |
| HEA[DING] {OFF\|ON} | Sets the column headings on or off in reports |

| Environment Variable | Description |
|---|---|
| HEADS[EP] {\|\|c\|OFF\|ON} | Defines the character that you enter for the heading separator |
| LIN[ESIZE] {80\|n} | Sets the total number of characters that SQL *Plus displays in one line before starting a new line |
| NEWP[AGE] {1\|n\|NONE} | Sets the number of blank lines between the top of each page and the title of the page |
| NUMF[ORMAT] format | Sets the default number format |
| NUM[WIDTH] {10\|n} | Sets the default width for numbers to display |
| PAGES[IZE] {24\|n} | Sets the number of lines in each page |
| PAU[SE] {OFF\|ON\|text} | Allows you to control the scrolling of your terminal when the reports are running |
| SERVEROUT[PUT] [FOR[MAT] {WRA[PPED]\| WOR[D_WRAPPED] \|TRU[NCATED]}] | Controls whether to display the output (DBMS_OUTPUT.PUT_LINE) of the stored procedures or PL/SQL block in SQL *Plus |
| SHOW[MODE] {OFF\|ON} | Displays the old and new settings of a SQL *Plus system variable |
| SQLBL[ANKLINES] {ON\|OFF} SQLC[ASE] | Allows blank lines within an SQL command |
| SQLCO[NTINUE] {> \|text} | Controls the line continuation prompt when a command does not fit in a line and needs to be continued. The default continuation character is the hyphen (-) |
| SQLN[UMBER] {OFF\|ON} | Sets the prompt for the second and the subsequent lines of the SQL statement |
| SQLPRE[FIX] {#\|c} | Sets the SQL *Plus prefix character that you use with an SQL *Plus command in a separate line to execute the command immediately without affecting the SQL statement being entered |

| Environment Variable | Description |
|---|---|
| SQLP[ROMPT] {SQL>\|text} | Sets the SQL *Plus command prompt |
| SQLT[ERMINATOR] {;\|c\|OFF\|ON} | Sets the character that is used to terminate and execute SQL statements |
| SUF[FIX] {SQL\|text} | Sets the default file extension for SQL *Plus scripting |
| TERM[OUT] {OFF\|ON} | Controls the display of the output generated by executing the contents of a file |
| TI[ME] {OFF\|ON} | Shows the current time at the command prompt |
| TIMI[NG] {OFF\|ON} | Controls the display of timing statistics when an SQL command and PL/SQL block is run |
| TRIM[OUT] {OFF\|ON} | Specifies whether to allow blank space at the end of each displayed line |
| TRIMS[POOL] {ON\|OFF} | Specifies whether to allow blank spaces at the end of each spooled line |
| UND[ERLINE] {-\|c\|ON\|OFF} | Sets the character that is used to underline column headings in SQL *Plus |
| VER[IFY] {OFF\|ON} | Determines whether SQL *Plus will list the text of a command before and after replacing substitution variables with values |
| WRA[P] {OFF\|ON} | Specifies whether to truncate or wrap the display of a selected row if the width of the current line is long |

***Table 2-1*** *Common environment variables used with the* ***SET*** *command*

The following example will illustrate the use of the **SET** command:

Enter the following statement in SQL *PLUS:

```
SELECT * FROM Emp;
```

The above statement will return all rows of the **Emp** table, as shown in Figure 2-8.

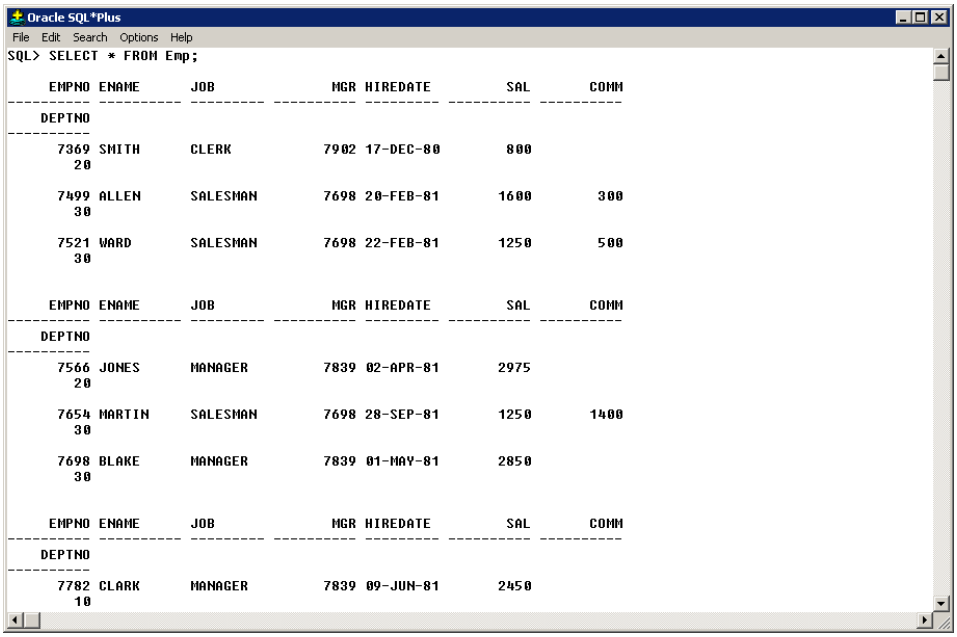



***Figure 2-8*** *Output of the* ***SELECT*** *statement before setting the environment*

Now, issue the following command:

SQL>SET PAGESIZE 24

This command will set the page size to 24.

SQL>SET LINESIZE 130 PAGESIZE 30

This command will set the line size to 130 and the page size to 30. Now, reissue the following statement to display all rows of the **Emp** table, as shown in Figure 2-9.

SELECT * FROM Emp;

Alternatively, you can modify the SQL *Plus environment by using the **Environment** dialog box shown in Figure 2-10. To invoke this dialog box, choose **Options > Environment** from the menu bar. Next, perform the following steps to set the environment variables using this dialog box.

1. Select the **linesize** option from the **Set Options** list box. Next, select the **Custom** radio button in the **Value** area and enter **120** in the edit box below the **Off** radio button to display 120 characters per line.

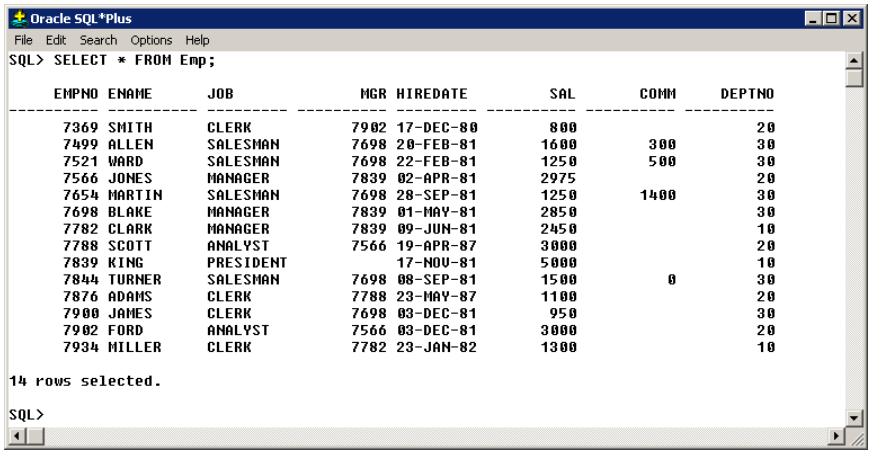



***Figure 2-9*** *Output of the* ***SELECT*** *statement after setting the environment*

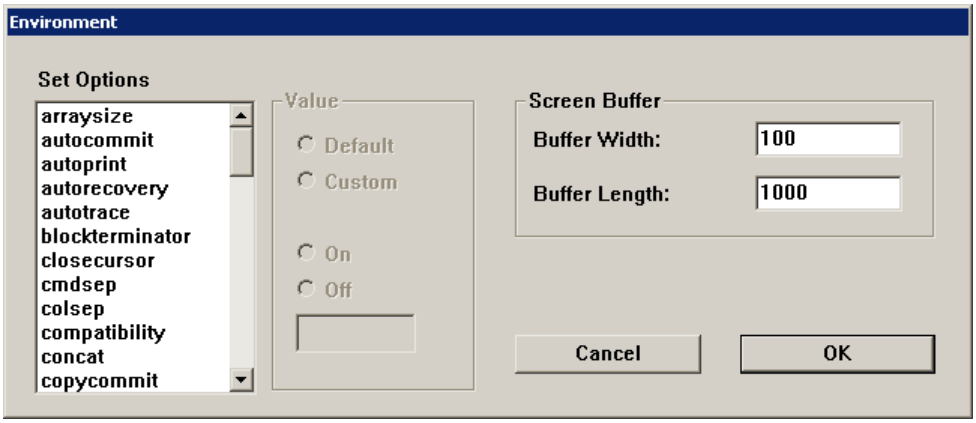



***Figure 2-10*** *The* ***Environment*** *dialog box*

2. Next, select the **pagesize** option from the **Set Options** list box. Select the **Custom** radio button in the **Value** area and enter **40** in the edit box below the **Off** radio button to display 40 lines per page.

3. Choose the **OK** button to save the settings.



**Note**

*These settings will be lost when you log out or exit the* ***Oracle SQL *Plus*** *window. It means that whenever you open the* ***Oracle SQL *Plus*** *window, you need to specify these settings to modify the display environment of SQL *Plus.*

You can set multiple environment variables using the following single **SET** command:

```
SET TIME ON LINESIZE 130 PAGESIZE 30
```

This command will display the time on the left side of the SQL prompt and set the page and line size, as shown in Figure 2-11.

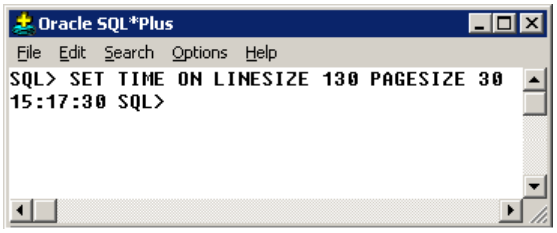



***Figure 2-11*** *Using the* ***TIME, LINESIZE****, and* ***PAGESIZE*** *options*

# DATA TYPES

A data type is the name or label for a set of values. It specifies what type of value an attribute or a variable holds. Also, it specifies how the information will be stored in a computer. Each attribute or variable has a data type. The data type of a value associates a fixed set of properties with the value. These properties cause Oracle to treat the values of one data type differently from the values of another data type. When you create a table, you need to specify the data type for each of its columns. The data types of columns define the domain for values that each column can contain. In-built data types of Oracle can be broadly classified into four categories, as listed in Table 2-2.

| Category | Data Type |
|---|---|
| Character | CHAR, NCHAR, VARCHAR, VARCHAR2, and NVARCHAR2 |
| Number | NUMBER, Fixed-point, Floating-point, BINARY_FLOAT, and BINARY_DOUBLE |
| Date | DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, and INTERVAL YEAR TO MONTH, and INTERVAL DAY TO SECOND |
| LOB | BLOB, CLOB, BFILE, and NCLOB |

***Table 2-2*** *In-built data types of Oracle*

## Character Data Type

The Oracle Database provides character data types to store character values. These data types are discussed next.

### CHAR

The CHAR data type is used to store the fixed length character data. The maximum length of data that it can store is 2000 bytes or characters. The default value for CHAR data type is 1.

The syntax for declaring column with the CHAR data type is as follows:

```
column_name CHAR(width)
```

In the above syntax, **column_name** is the name of the column having the CHAR data type and **width** is optional, which can be any integer value ranging from 1 to 2000. If you do not specify the value of **width**, the default value will be set.

### NCHAR

The NCHAR data type is used to store the fixed length character string in the national character set of the database. This data type can hold up to 2000 characters. Defining national character set in the database determines the maximum length of the column. When you create a table with a column having NCHAR data type, you define the column length in characters.

The syntax for declaring column with the NCHAR data type is as follows:

```
column_name NCHAR(width)
```

In the above syntax, **column_name** is the name of the column having the NCHAR data type and **width** is an integer value that ranges from 1 to 2000.

For example:

```
Status  CHAR(1)
```

In the above example, the column **Status** is of the CHAR data type and has width of 1 unit.

### VARCHAR

The VARCHAR data type is used to store a variable-length character string. The maximum width of the VARCHAR data type is 4000 bytes or characters. It is recommended to use the VARCHAR2 data type rather than the VARCHAR data type.

The syntax for declaring column with the VARCHAR data type is as follows:

```
column_name VARCHAR(width)
```

In the above syntax, **column_name** is the name of the column having the VARCHAR data type and **width** is an integer value that ranges from 1 to 4000. If you do not specify the value of **width**, Oracle will return an error.

### VARCHAR2

The VARCHAR2 data type is also used to store a variable-length character string. While creating the VARCHAR2 column, you can specify the maximum number of bytes or characters of data that can be stored in this column. If you try to enter a value that exceeds the maximum length of the column, then the Oracle database will return an error. However,

if you enter a value that is smaller than the column size, the Oracle database will store the actual value of the data and set the remaining space free. This implies that the Oracle database does not add the trailing blank spaces to the data value and thus, let the remaining space free for other purpose. The maximum width of the VARCHAR2 data type is 4000 bytes.

The syntax for declaring column with the VARCHAR2 data type is as follows:

```
column_name VARCHAR(width)
```

In the above syntax, **column_name** is the name of the column having the VARCHAR2 data type and **width** is an integer value ranging from 1 to 4000. If you do not specify the value of **width**, Oracle will return an error.



**Note**
*The VARCHAR2 data type is the successor of VARCHAR. Therefore, it is recommended that you use VARCHAR2 as a variable-sized array of characters, rather than VARCHAR.*

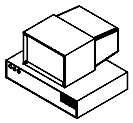
### NVARCHAR2

The NVARCHAR2 data type is used to store variable-length or multibytes character set data. While creating the NVARCHAR2 column, you can specify the maximum number of bytes or characters of data that can be stored in this column. If you enter a value that exceeds the maximum length of the column, Oracle will throw an error. However, if the value entered is smaller than the column size, Oracle will store the actual value of the data and set the remaining space free. The maximum length of the column is determined by the national character set defined in the column. The maximum width of the NVARCHAR2 data type is 4000 bytes.

The syntax for declaring column with the NVARCHAR2 data type is as follows:

```
column_name NVARCHAR(width)
```

In the above syntax, **column_name** is the name of the column having the NVARCHAR2 data type and **width** is an integer value ranging from 1 to 4000. If you do not specify the value of **width**, Oracle will throw an error.



**Tip**: *Both NCHAR and NVARCHAR2 are Unicode data types, which store Unicode characters. The character set of NCHAR and NVARCHAR2 data types can be either AL16UTF16 or UTF8. The character set AL16UTF16 or UTF8 can be specified while creating a database.*

## NUMBER Data Type

The NUMBER data type stores variable-length numeric data with a precision between 1 and 38, and the scale has a range between –84 and 127. It can store the zero, positive numbers, or negative fixed numbers with absolute values from $1.0 \times 10^{-130}$ to $1.0 \times 10^{126}$ digits as well as fixed and floating point numbers. If you specify the value of an expression greater than $1.0 \times 10^{126}$, the Oracle database returns an error. The Oracle database provides three subtypes of the NUMBER data type: Fixed-point, Floating-point, and Integer.

## Fixed-point Number

To define the Fixed-point number data type, you have to specify the values of both precision and scale.

The syntax for declaring column with the Fixed-point number data type is as follows:

```
Column_Name NUMBER(P,S)
```

In the above syntax, **Column_Name** is the name of the column having the NUMBER data type. **P** is the precision or the total number of digits with precision up to 38 digits and **S** is the scale or the number of digits on the right of the decimal point. The value of **S** can range from -84 to 127. The precision value denotes all digits on the left of the decimal point, whereas the scale value denotes all digits to right of the decimal point.

## Integer Numbers

An integer is a whole number with no digit on the right of the decimal point. You can define a column of integer data type by omitting the scale value.

The syntax for declaring column with an integer number data type is as follows:

```
Column_Name NUMBER(P)
```

In the above syntax, **Column_Name** is the name of the column having the NUMBER data type and **P** is the precision or the total number of digits with precision up to 38 digits.

## Floating-point Number

The Floating-point numbers can have a decimal point anywhere between the first and the last digits, or it can be a number without any decimal point as there is no restriction for the decimal point. Scale is not applicable for this data type. To declare a column with this data type, omit the precision and scale values.

The syntax for declaring column with the Floating-point number data type is as follows:

```
Column_Name NUMBER
```

In the above syntax, **Column_Name** is the name of the column having the NUMBER data type. Oracle provides two numeric data types for floating-point numbers: BINARY_FLOAT and BINARY_DOUBLE.

#### BINARY_FLOAT

The BINARY_FLOAT data type is a single-precision floating-point number data type. Each BINARY_FLOAT value requires 5 bytes, including a length byte.

#### BINARY_DOUBLE

The BINARY_DOUBLE is a double-precision floating-point number data type. Each BINARY_DOUBLE value requires 9 bytes, including a length byte.



**Note**
*BINARY-FLOAT is a 32-bit data type and BINARY-DOUBLE is a 64-bit data type.*

## Datetime and Interval

The Oracle datetime data types store date and time values. The datetime data types are DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, and TIMESTAMP WITH LOCAL TIME ZONE. The interval data types are INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND. The datetime and interval data types are discussed next.

### DATE

The DATE data type is used to store date and time. Oracle stores the following information for each date value: century, year, month, date, hour, minute, and second. You can represent the date and time in both character and number data types. The character and numeric dates can be converted into date value by using the **TO_DATE** function. This function will be discussed in later chapters. The default date format is DD-MON-YY and the time format is HH:MI:SS using the 12-hours clock, whereas the date format for ANSI is YYYY-MM-DD. The valid date range is from January 1, 4712 BC to December 31, 9999 AD.

The syntax for declaring column with the DATE data type is as follows:

```
Column_Name DATE
```

In the above syntax, **Column_Name** specifies the name of the column having the DATE data type. If you specify a date value without the time component, the default time will be midnight (00:00:00 or 12:00:00 for 24-hour and 12-hour clock time, respectively). If you specify a date value without specifying the day, then the default date will be the first day of the current month.

### TIMESTAMP

The TIMESTAMP data type stores all information that the DATE data type stores, including the fractional part of seconds. It is an expansion of the DATE data type. It stores century, year, month, day, hour, minute, second, and fractional seconds. This data type is useful for storing precise time values.

The syntax for declaring a column with the TIMESTAMP data type is as follows:

```
Column_Name TIMESTAMP [(Fractional_Seconds_Precision)]
```

In the above syntax, **Column_Name** is the name of the column having the TIMESTAMP data type. **Fractional_Seconds_Precision** is an optional value and it indicates the number of digits that Oracle will store in the fractional part of the seconds datetime field. The value of **Fractional_Seconds_Precision** can range from 0 to 9. If you do not specify this value, Oracle will take its default value, which is 6.

## TIMESTAMP WITH TIME ZONE

The TIMESTAMP WITH TIME ZONE data type is an alternative to the TIMESTAMP data type. The value stored by this data type includes time zone offset. There are two ways to set time zone: first by using the UTC offset, say '+10:0', and the second is by using the name of region, say 'Australia/Sydney'. This data type is useful for collecting and evaluating date information across geographic regions.

The syntax for declaring a column with the TIMESTAMP WITH TIME ZONE data type is as follows:

```
Column_Name TIMESTAMP [(Fractional_Seconds_Precision)] WITH TIME ZONE
```

In the above syntax, **Column_Name** is the name of the column having the TIMESTAMP WITH TIME ZONE data type. **Fractional_Seconds_Precision** is an optional value and is used to specify the number of digits that Oracle can store in the fractional part of the seconds datetime field. The value of **Fractional_Seconds_Precision** can range from 0 to 9. If you omit this value, it will take the default value 6.

## TIMESTAMP WITH LOCAL TIME ZONE

The TIMESTAMP WITH LOCAL TIME ZONE data type is another alternative to the TIMESTAMP data type. It also includes a time zone offset in its value. Unlike the TIMESTAMP WITH TIME ZONE data type, the TIMESTAMP WITH LOCAL TIME ZONE data type does not store the time zone offset as part of the column data. When a user retrieves the data from TIMESTAMP WITH LOCAL TIME ZONE data type column, Oracle returns it in the local time zone of the client's system in a two-tier application.

The syntax for declaring the column having the data type TIMESTAMP WITH LOCAL TIME ZONE is as follows:

```
Column_Name TIMESTAMP [(Fractional_Seconds_Precision)] WITH LOCAL TIME
ZONE
```

In the given syntax, **Column_Name** is a name of the column having the TIMESTAMP WITH LOCAL TIME ZONE data type. **Fractional_Seconds_Precision** is optional and is used to specify the number of digits that can be stored in the fractional part of the seconds datetime field. The value of **Fractional_Seconds_Precision** can range from 0 to 9. If you omit this value, it will take the default value 6.




**Note**

*The time zone offset is the difference (in hours and minutes) between the local time and UTC (Coordinated Universal Time, formerly Greenwich Mean Time).*

## INTERVAL YEAR TO MONTH

The INTERVAL YEAR TO MONTH data type is used to store the period of time that represents year and month.

The syntax for declaring the column having the data type INTERVAL YEAR TO MONTH is as follows:

```
Column_Name INTERVAL YEAR [(year_precision)] TO MONTH
```

In the above syntax, **Column_Name** is the name of the column having the INTERVAL YEAR TO MONTH data type and **year_precision** is the number of digits in the YEAR datetime field. The value of **year_precision** can range from 0 to 9 and its default value is 2.

### INTERVAL DAY TO SECOND

The INTERVAL DAY TO SECOND data type is used to store the period of time that represents days, hours, minutes, and seconds with a fractional part.

The syntax for declaring the column having the data type INTERVAL DAY TO SECOND is as follows:

```
INTERVAL DAY [(day_precision)] TO SECOND [(fractional_seconds_precision)]
```

In the above syntax, **day_precision** is the number of digits in the day datetime field. The value of **day_precision** can range from 0 to 9. If you do not specify this value, Oracle will assign the default value 2 for this field. **fractional_seconds_precision** is the number of digits in the fractional part of the second datetime field. The value of **fractional_seconds_precision** can range from 0 to 9. The default value of **fractional_seconds_precision** is 6.

## LOB

LOB stands for Large Object. It is a data type and stores unstructured information upto 4 gigabytes such as sound clips, video files, and so on. The LOB data types allow efficient, random, and easy access to the data. The values stored in this data type are known as locators. These locators store the locations of large objects. The location may be stored in-line (in the database) or out-line (outside the database). The LOBs can be manipulated by using the DBMS_LOB package and Oracle Call Interface (OCI). The LOBs can be external or internal depending upon their locations with respect to the database.

The LOB data types available in Oracle database are BLOB, CLOB, NCLOB, and BFILE.

### BLOB

BLOB stands for Binary Large Objects. This data type is used to store binary data up to 4 GB in length. BLOB is stored in the database.

The syntax for declaring a column with the BLOB data type is as follows:

```
Column_Name BLOB
```

In the above syntax, **Column_Name** is the name of a column having the BLOB data type.

### CLOB

CLOB stands for Character Large Objects and can store character data up to 4 GB in length. CLOB is also stored in the database.

The syntax for declaring a column with the CLOB data type is as follows:

```
Column_Name CLOB
```

In the above syntax, **Column_Name** is the name of the column having the CLOB data type.

### BFILE

BFILE stands for Binary FILE. It is a pointer (reference) to the external file. The files referenced by BFILE exist in the file system and enables you to access the binary file that are stored outside the Oracle database. The database only maintains a pointer to the file. The size of the external file is limited only by the operating system because the data is stored outside the database.

The syntax for declaring a column with the BFILE data type is as follows:

```
Column_Name BFILE
```

In the above syntax, **Column_Name** is the name of the column having the BFILE data type.

### NCLOB

The NCLOB data type supports both fixed-width and variable-width character sets. The NCLOB data type can store up to 4 gigabytes of character text data.

The syntax for declaring a column with the NCLOB data type is as follows:

```
Column_Name CLOB
```

In the above syntax, **Column_Name** is the name of the column having the NCLOB data type.




**Note**

*You cannot save the BLOB, CLOB, and NCLOB locators in a PL/SQL or Oracle Call Interface (OCI) variable in one transaction and then use it in another transaction or session. Also, you cannot specify the object size because the database automatically allocates space to store the LOB object.*

## CONSTRAINTS

Constraints are a set of predefined rules, which ensure that the valid data values are stored in the columns of a table. Oracle provides some predefined commands that enable you to define the constraints for a table or a column. There are two types of constraints: integrity and value. The integrity constraints include primary key and foreign key. The value

constraints define specific data values or data ranges. The values entered in columns should not be Null. There are two levels of constraints: table level constraint and column level constraint.

The table level constraints restrict the values that a table can store. These constraints can be referred to one or more columns in a table. The table level constraint includes the following constraints: PRIMARY KEY, UNIQUE, FOREIGN KEY, and CHECK.

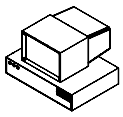
The column level constraints can be referred to a single column in a table, and they do not specify a column name, except the CHECK constraint. As a result, they limit the values that can be placed in a specific column, irrespective of values that exist in other table rows. The column level constraint can be one of the following: UNIQUE, NOT NULL, PRIMARY KEY, and FOREIGN KEY.

The syntax and behavior of the table level constraint and the column level constraint is similar with only the following difference:

1. The syntax for table level constraints is separated from the column definitions by comma.

2. The table level constraints must follow the definition of the columns to which they are referred.

3. The table level constraint can be defined for more than one column and SQL evaluates the constraint based on the combination of values stored in all columns.



**Tip**: *The column level constraints and the table level constraints have the same functionality; the only difference between them is that the table level constraints allow you to specify more than one column, whereas the column level constraints refer to only one column. The table level constraints are specified at the end of the* ***CREATE TABLE*** *command.*

The basic structure of a constraint used in Oracle is as follows:

The keyword **CONSTRAINT** is followed by a unique constraint name and then by a constraint definition. The constraint name is used to manipulate the constraint once a table has been created. The syntax for declaring a constraint is as follows:

```
CONSTRAINT [Constraint_Name] Constraint_Type
```

In the above syntax, **Constraint_Type** may be either a column level constraint (column_constraint) or a table level constraint (table_constraint).




**Note**

*If you omit the name of the constraint, Oracle will assign an arbitrary name to it. This constraint name is used to drop the constraint by using the* ***ALTER*** *statement, which will be discussed later in this chapter.*

## Integrity Constraint

An integrity constraint is a mechanism that is used by Oracle to prevent users from entering invalid data into a table. Integrity constraints are a set of rules for columns of a table. It includes the primary key and foreign key constraints.

### Primary Key Constraint

The primary key constraints ensure that the Null values are not entered in a column and also the value entered is unique. Thus, these constraints avoid the duplication of records. A primary key constraint can be defined in the **CREATE TABLE** and **ALTER TABLE** commands. This constraint can be declared at both levels: within the column level and at the table level.

The syntax for declaring a primary key constraint at the column level is as follows:

```
CONSTRAINT Constraint_Name PRIMARY KEY
```

In the above syntax, **CONSTRAINT** and **PRIMARY KEY** are keywords and **Constraint_Name** is the name of the constraint.

The syntax for declaring a primary key constraint at the table level is as follows:

```
CONSTRAINT Constraint_Name PRIMARY KEY (Column_Name)
```

In the above syntax, **CONSTRAINT** and **PRIMARY KEY** are keywords. **Constraint_Name** is the name of the constraint and **Column_Name** is the name of the column for which you want to declare the constraint.

You can also create a primary key constraint for more than one column. The syntax for declaring the primary key for more than one column is as follows:

```
CONSTRAINT Constraint_Name PRIMARY KEY (Column_Name1,
Column_Name2, Column_Name3, Column_Name4 ...)
```

In the above syntax, **CONSTRAINT** and **PRIMARY KEY** are keywords and **Constraint_Name** is the name of the constraint. **Column_Name1**, **Column_Name2**, **Column_Name3**, **Column_Name4**, and so on are the names of the columns for which you want to declare the primary key constraint.



**Note**

*In Oracle, the primary key constraint cannot be declared for more than 32 columns.*

### Foreign Key Constraint

The foreign key constraint is the property that guarantees the dependency of data values of one column of a table with another column of a table. A foreign key constraint, also known as referential integrity constraint, is declared for a column to ensure that the value in one column

is found in the column of another table with the primary key constraint. The table containing the foreign key constraint is referred to as the child table, whereas the table containing the referenced (Primary key) is referred to as the parent table. The foreign key reference will be created only when a table with the primary key column already exists. The foreign key constraint can be declared in two ways: within the column declaration and at the end of the column declaration.

The syntax for using the foreign key constraint within the column declaration is as follows:

```
CONSTRAINT Constraint_Name REFERENCE Primary_Key_Table_Name
(Primary_Key_Column_Name)
```

In the above syntax, **CONSTRAINT** and **REFERENCE** are keywords, whereas **Constraint_Name** is the name of the constraint and **Primary_Key_Table_Name** is the name of the table that contains the referenced column. The referenced column **Primary_Key_Column_Name** is the primary key of the table **Primary_Key_Table_Name**.

The syntax for declaring the foreign key constraint at the end of the column declaration:

```
CONSTRAINT Constraint_Name FOREIGN KEY (Column_Name) REFERENCE
Primary_Key_Table_Name (Primary_Key_Column_Name)
```

In the above syntax, **CONSTRAINT**, **FOREIGN KEY**, and **REFERENCE** are keywords, whereas **Column_Name** is the name of the column that is declared as the foreign key. **Constraint_Name** is the name of the constraint and **Primary_Key_Table_Name** is the name of the table that contains the referencing column. The referencing column is the primary key of the table **Primary_Key_Table_Name**. And, **Primary_Key_Column_Name** is the name of the primary key column of the table **Primary_Key_Table_Name**.



**Note**
*If you declare the foreign key constraint at the column level, the column name is not required. Also, in a foreign key constraint, you cannot use the keyword **FOREIGN KEY**.*

## Value Constraint

Value constraints are column level constraints. The value constraints include the CHECK, NOT NULL, DEFAULT, and UNIQUE constraints. These constraints are discussed next.

### NOT NULL Constraint

A column in a table can be declared with the NOT NULL constraint. On declaring this constraint, you cannot insert Null value in the column. You can add this constraint while creating the table by using the **CREATE TABLE** command. You can also add this constraint after creating the table by using the **ALTER** command. The **ALTER** command will be discussed later in the chapter.

The syntax for declaring the NOT NULL constraint within the column declaration is as follows:

```
CONSTRAINT Constraint_Name NOT NULL
```

In the above syntax, **CONSTRAINT** is a keyword and **Constraint_Name** is the name of the constraint.

For example:

```
Last_Name VARCHAR2(30) CONSTRAINT L_Name NOT NULL
```

In the above example, the column **Last_Name** is declared with the NOT NULL constraint named **L_Name**. The **L_Name** constraint ensures that you cannot insert a Null value in the column **Last_Name**.

## CHECK Constraint

The CHECK constraint ensures that all values inserted into the column satisfy the specified condition. This constraint checks data against the expression defined in the **INSERT** and **UPDATE** statement. The CHECK constraint can be declared at the column level.

The syntax for declaring the CHECK constraint within the column declaration is as follows:

```
CONSTRAINT Constraint_Name CHECK(Col_Condition)
```

In the above syntax, **CONSTRAINT** and **CHECK** are keywords. **Constraint_Name** is the name of the constraint and **Col_Condition** is the rule or the condition for entering values in the column.

For example:

```
Commission NUMBER Check_Column_Value CHECK(Commission>500)
```

In the above example, the **Commission** column is declared with the CHECK constraint **Check_Column_Value**, which ensures that the data values entered in the **Commission** column are greater than 500.

## UNIQUE Key Constraint

The UNIQUE key constraint is used to prevent the duplication of data values within the rows of a specified column or a set of columns in a table. The column defined with the UNIQUE key constraint can also allow a null value. Moreover, this constraint can be added to the existing columns. The UNIQUE key constraint can be declared both at the column level and the table level.

The syntax for declaring the UNIQUE key constraint at the column level is as follows:

```
CONSTRAINT Constraint_Name UNIQUE
```

In the above syntax, **CONSTRAINT** and **UNIQUE** are keywords and **Constraint_Name** is the name of the constraint.

For example:

```
First_Name VARCHAR2(50) CONSTRAINT Unique_FirstName UNIQUE
```

In the above example, the column **First_Name** is defined with the UNIQUE key constraint **Unique_FirstName**, which stores the first name of an employee. The constraint **Unique_FirstName** ensures that you cannot enter the same data in the column **First_Name**.

The syntax for declaring the UNIQUE key constraint at the table level is as follows:

```
CONSTRAINT Constraint_Name UNIQUE(Column_Name)
```

In the above syntax, **Constraint_Name** is the name of the constraint and **Column_Name** is the name of the column for which the UNIQUE key constraint is declared. The declaration of the UNIQUE key constraint at the table level is made at the end of the declaration of columns.

For example:

```
CONSTRAINT Unique_FirstName UNIQUE (First_Name)
```

In the above example, the constraint **Unique_FirstName** is declared for the column **First_Name**. This constraint ensures that you cannot enter the same data values in the column **First_Name**.

## DEFAULT Constraint

The DEFAULT constraint is used to set the default value for a column. This constraint ensures that a default value is set automatically by Oracle for each column of a table. The DEFAULT constraints are declared at the column level declaration.

The syntax for declaring the DEFAULT constraint is as follows:

```
DEFAULT 'default_value'
```

In the above syntax, **DEFAULT** is a keyword and **default_value** is the value set as the default value for a column.

For example:

```
Country VARCHAR2 (50) DEFAULT 'USA'
```

In the above example, the column **Country** is declared with the DEFAULT constraint. If a user enters a Null value in this column, then Oracle will insert a default value other than the Null value. But, if the user enters a data value rather than the Null value, the default value will be replaced by the data value entered by the user.

# CREATING A TABLE

A table is the basic unit of data storage in the Oracle database. Database holds data in the tabular form, which is in rows and columns. A table such as an Employee table can contain various columns such as Emp_ID, First_Name, Last_Name, and so on. Each column has a width and data types, such as VARCHAR2, DATE, NUMBER, and so on. The width can be pre-determined by the data type, as in case of the data type DATE. But, if a column has a NUMBER data type, you can define precision and scale instead of width.

A row is a set of columns corresponding to a single record. A table can contain number of such records. For each column, you can specify rules, called integrity constraints. For example, the NOT NULL constraint ensures that each column of a row contains some data values.

Once a table has been created with columns and rows, you can retrieve, delete, or update data using the SQL statements. This will be discussed in later chapters. While creating a table, the naming convention for tables and columns should be properly followed. The naming conventions used while creating tables and columns are as follows:

1. The table and column names can be up to 30 characters long.

2. The table and column names must begin with an alphabet.

3. Names cannot contain quotes.

4. Names are not case-sensitive.

5. Names can contain characters a to z, 0 to 9, _, $, and #.

6. The reserve words used in Oracle cannot be used as names of columns or tables.

The syntax for creating a table in Oracle is as follows:

```
CREATE TABLE Table_Name
(
Field_Name1 Field_Datatype (width),
Field_Name2 Field_Datatype (width),
Field_Name3 Field_Datatype (width),
Field_Name4 Field_Datatype (width),
.......
.......
);
```

In the above syntax, **CREATE** and **TABLE** are keywords and **Table_Name** is the name of the table to be created. **Field_Name1**, **Field_Name2**, **Field_Name3**, and **Field_Name4** are the names of columns. **Field_Datatype** represents the data type of the column and **width** represents the length of the column.

For example:

To create a table **Student** containing student data such as roll number, name, date of birth, and so on, you need to follow the steps given below:

1. Enter the **CREATE TABLE** command at the SQL prompt, as shown Figure 2-12.

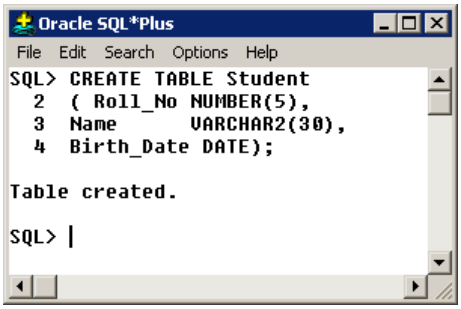



***Figure 2-12*** *The **Student** table created using the **CREATE TABLE** command*



**Note**

*You are recommended not to enter line numbers in SQL *Plus.*

2. Enter **;** (semi colon) at the end of the last line. It marks the end of the SQL command.

3. To execute command lines, press ENTER. If there is no error in the command lines, Oracle will return a message, **Table created**, which confirms that the table has been created.

4. To check whether the **Student** table has been created, enter the following command in the SQL *Plus window:

   SQL>DESC Student;

5. After entering the command in SQL *Plus, press ENTER; the output will be displayed immediately after this command line, refer to Figure 2-13.

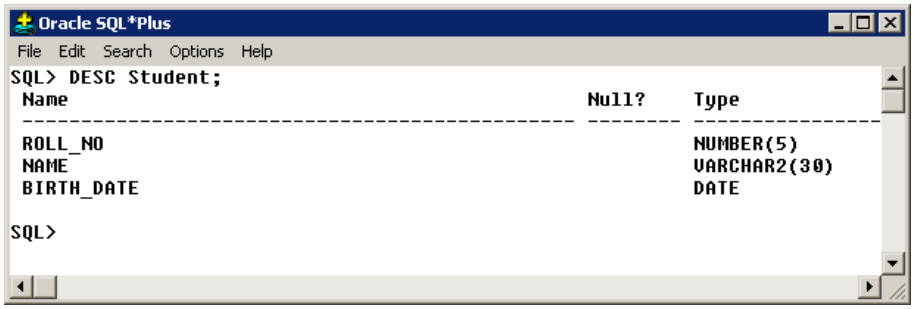



***Figure 2-13*** *Using the **DESC** command*

## Creating a Table with the Primary Key Constraint

You can create a table with the primary key constraint in two ways: by declaring the column with the primary key constraint at the column level and by declaring constraint at the table level.

The syntax for creating a table with the primary key constraint declared at the column level is as follows:

```
CREATE TABLE Table_Name
(
Field_Name1 Field_Datatype (width) CONSTRAINT Constraint_Name PRIMARY
KEY,
Field_Name2 Field_Datatype (width),
Field_Name3 Field_Datatype (width),
.......
.......
)
```

In the above syntax, **CREATE TABLE**, **CONSTRAINT**, and **PRIMARY KEY** are keywords; **Table_Name** is the name of the table to be created; **Field_Name1**, **Field_Name2**, and **Field_Name3** are names of the columns; **Field_Datatype** is the data type of specific columns; and **Constraint_Name** is the name of the primary key constraint. Here, the primary key constraint ensures that the column value is not Null and the values in that column are unique.

The syntax for creating a table with the primary key constraint declared at the table level is as follows:

```
CREATE TABLE Table_Name
(
Field_Name1 Field_Datatype (width)
Field_Name2 Field_Datatype (width),
Field_Name3 Field_Datatype (width),
Field_Name4 Field_Datatype (width),
.......,
.......,
CONSTRAINT Constraint_Name PRIMARY KEY(Field_Name)
)
```

In the above syntax, **CREATE TABLE**, **CONSTRAINT**, and **PRIMARY KEY** are keywords and **Table_Name** is the name of the table to be created. **Field_Name1**, **Field_Name2**, **Field_Name3** and **Field_Name4** are names of columns; **Field_Datatype** is the data type of the specific column; **Constraint_Name** is the name of the primary key constraint; and **Field_Name** is the name of the column or field declared as the primary key.

For example:

To create a table **Student** containing the student data such as roll number, name, and date of birth with roll number as its primary key, you need to follow the steps given below:

1. In SQL *Plus, enter the **CREATE TABLE** command at the SQL prompt, as shown in Figure 2-14 and Figure 2-15.

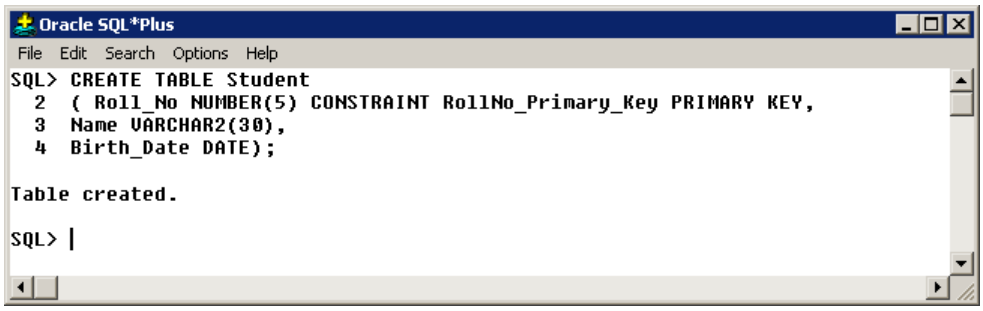



***Figure 2-14*** *The **CREATE TABLE** command with the primary key constraint declared at the column level*

Figure 2-14 shows the **CREATE TABLE** command with the primary key constraint declared at the column level.

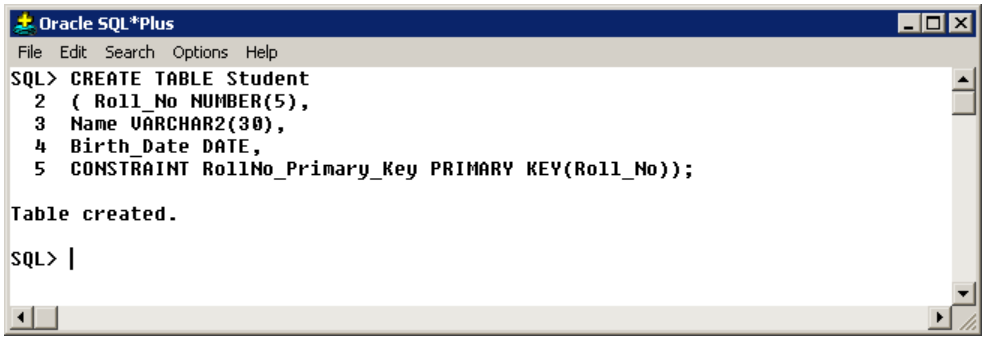



***Figure 2-15*** *The **CREATE TABLE** command with the primary key constraint declared at the table level*

Figure 2-15 shows the **CREATE TABLE** command with the primary key constraint declared at the table level.

2. Enter **;** (semi colon) at the end of the last line. It marks the end of the SQL command.

3. To execute command lines, press ENTER. If there is no error in the command lines, Oracle will return the message, **Table created**, which confirms that the table has been created.

The following **CREATE TABLE** command will create a table with the primary key constraint declared for more than one field:

```
CREATE TABLE supplier
(
```

```
Supplier_ID      NUMERIC(10) NOTNULL,
Supplier_Name  VARCHAR2(50) NOTNULL,
Contact_Name   VARCHAR2(50),
CONSTRAINT Supplier_PK PRIMARY KEY (Supplier_ID, Supplier_Name)
);
```

## Creating a Table with the Foreign Key Constraint

You can create a table with the foreign key constraint in two ways: by declaring the column with foreign key constraint at the column level and by declaring constraints at the table level.

The syntax for creating a table with the foreign key constraint declared at the column level is as follows:

```
CREATE TABLE Table_Name
(
Field_Name1 Field_Datatype (width) CONSTRAINT Constraint_Name REFERENCE
Primary_Key_Table_Name (Primary_Key_Column_Name),
Field_Name2 Field_Datatype (width),
Field_Name3 Field_Datatype (width),
.......
.......
);
```

In the above syntax, **CREATE TABLE**, **CONSTRAINT**, and **REFERENCE** are keywords and **Table_Name** is the name of the table to be created. **Field_Name1**, **Field_Name2**, and **Field_Name3** are the names of columns; **Field_Datatype** is the data type of specific column; **Constraint_Name** is the name of the foreign key constraint; **Primary_Key_Table_Name** is the name of the parent table having primary key column; and **Primary_Key_Column_Name** is the name of the primary key column of the **Primary_Key_Table_Name** table.

The syntax for creating a table with the foreign key constraint declared at the table level is as follows:

```
CREATE TABLE Table_Name
(
Field_Name1 Field_Datatype (width),
Field_Name2 Field_Datatype (width),
Field_Name3 Field_Datatype (width),
.......
.......
CONSTRAINT Constraint_Name FOREIGN KEY (Column_Name) REFERENCE
Primary_Key_Table_Name (Primary_Key_Column_Name)
)
```

In the above syntax, **CREATE TABLE, CONSTRAINT,** and **REFERENCE** are keywords and **Table_Name** is the name of the table to be created. **Field_Name1**, **Field_Name2**, and

**Field_Name3** are the names of columns; **Field_Datatype** is the data type of specific columns; **Constraint_Name** is the name of the foreign key constraint; **Primary_Key_Table_Name** is the name of the parent table having the primary key column; and **Primary_Key_Column_Name** is the name of the primary key column of the **Primary_Key_Table_Name** table.

For example:

To create a table that contains customer's address with reference to their Dealers, you need to follow the steps given below.

1. Enter the command lines given in Figure 2-16 or Figure 2-17 at the SQL prompt.

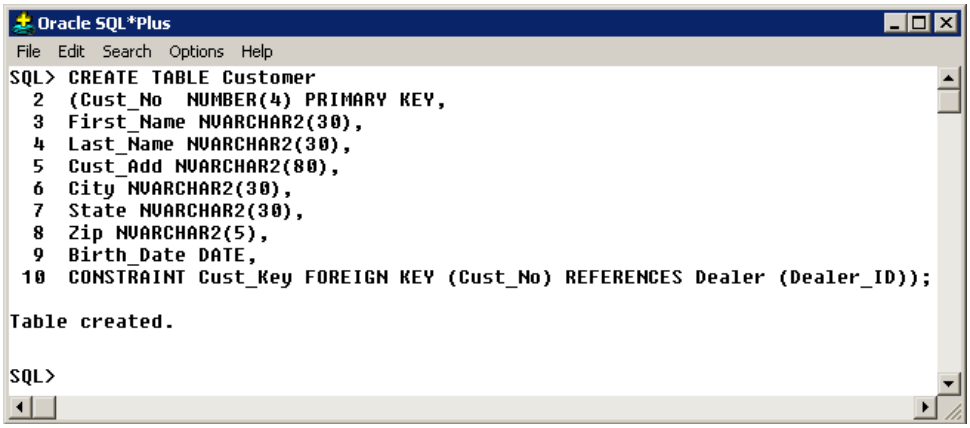



***Figure 2-16*** *The* ***CREATE TABLE*** *command with the foreign key constraint declared at the column level*

Figure 2-16 shows the **CREATE TABLE** command with the foreign key constraint declared at the column level.

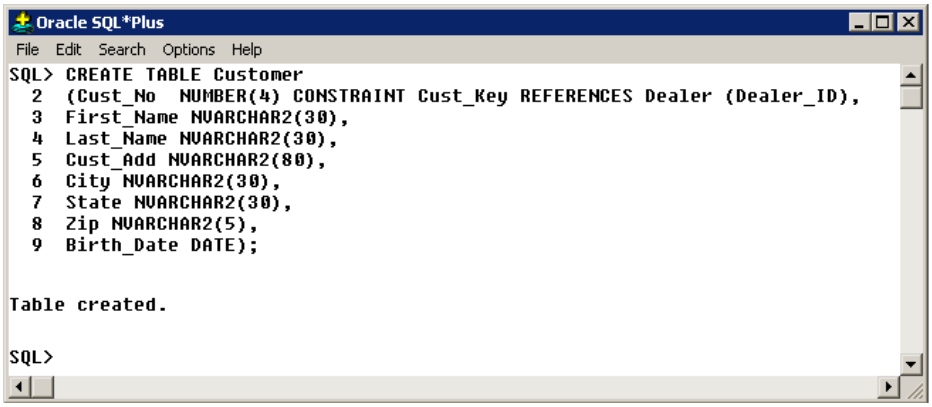



***Figure 2-17*** *The* ***CREATE TABLE*** *command with the foreign key constraint declared at the table level*

Figure 2-17 shows the **CREATE TABLE** command with the foreign key constraint declared at the table level.

2. Enter **;** (semi colon) at the end of the last line. It marks the end of the SQL command.

3. To execute command lines, press ENTER. If there is no error in the command lines, the Oracle will return a message **Table created**, which confirms that the table has been created.

## Creating a Table with the NOT NULL Constraint

You can create a table with the NOT NULL constraint by declaring it at the column level.

The syntax for creating the table with the NOT NULL constraint is as follows:

```
CREATE TABLE  Table_Name
(
Field_Name1 Field_Type (width) Constraint Constraint_Name NOT NULL,
Field_Name2 Field_Type (width),
Field_Name3 Field_Type (width),
Field_Name4 Field_Type (width),
(Col_Condition)
.....
.....
);
```

In the above syntax, **Field_Name1**, **Field_Name2**, **Field_Name3**, and **Field_Name4** are the names of columns; **Field_Type** is the data type of columns; **width** is the length of columns; and **Constraint_Name** is the name of the constraint declared for the column.

For example:

To create a table that contains customer's address with the NOT NULL constraint, you need to follow the steps given below:

1. Enter the **CREATE TABLE** command at the SQL prompt, as shown in Figure 2-18**.**

2. Enter **;** (semi colon) at the end of the last line. It marks the end of the SQL command.

3. To execute command lines, press ENTER.

In the above example, the NOT NULL constraint **Cust_NotNull** ensures that the value for the column **CUST_NO** should not allow Null values.

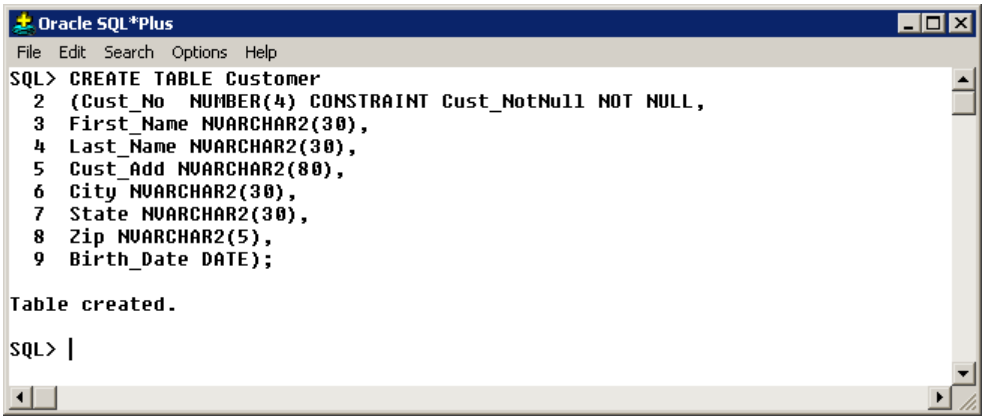



***Figure 2-18*** *The* ***CREATE TABLE*** *command with the NOT NULL constraint*

## Creating a Table with the DEFAULT Constraint

You can create a table with the DEFAULT constraint by declaring the column with DEFAULT constraint at the column level.

The syntax for creating the table with DEFAULT constraint is as follows:

```
CREATE TABLE  Table_Name
(
Field_Name1 Field_Type (width),
Field_Name2 Field_Type (width) Constraint Constraint_Name Default 'default_value',
Field_Name3 Field_Type (width),
Field_Name4 Field_Type (width),
.....
.....
)
```

In the above syntax, **Field_Name1**, **Field_Name2**, **Field_Name3**, and **Field_Name4** are the name of the columns; **Field_Type** is the data type of the columns; **width** is the length of columns; and **Constraint_Name** is the name of the constraint declared for the column.

For example:

To create a table to store customer's address with California as the default value for the state, you need to follow the steps given below.

1. Enter the **CREATE TABLE** command at the SQL prompt, as shown in Figure 2-19.

2. Enter **;** (semi colon) at the end of the last line. It marks the end of the SQL command.

3. To execute command lines, press ENTER. If there is no error in the command lines, Oracle will return the message **Table created**, which confirms that the table has been created.

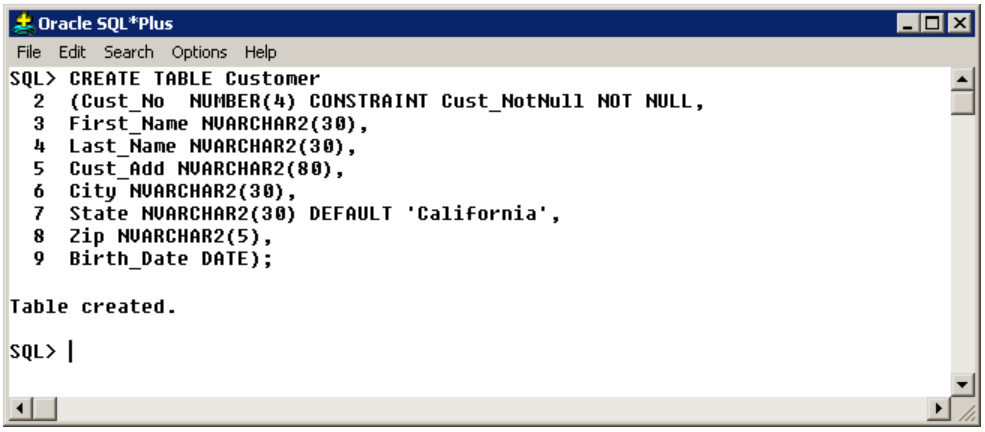



***Figure 2-19*** *The* ***CREATE TABLE*** *command with the DEFAULT constraint*

## Creating a Table with the UNIQUE and CHECK Constraints

You can create a table with the UNIQUE and CHECK constraints by declaring the column with the UNIQUE and CHECK constraints at the column level.

The syntax for creating the table with the UNIQUE and CHECK constraints is as follows:

```
CREATE TABLE  Table_Name
(
Field_Name1 Field_Type (width),
Field_Name2 Field_Type (width),
Field_Name3 Field_Type (width),
Field_Name4 Field_Type (width) Constraint Constraint_Name CHECK
(Col_Condition)
.....
.....
);
```

In the above syntax, **Field_Name1**, **Field_Name2**, **Field_Name3**, and **Field_Name4** are the names of columns; **Field_Type** is the data type of columns; **width** is the length of columns; and **Constraint_Name** is the name of the constraint declared for the column.

For example:

```
CREATE TABLE Customer
(
CUST_NO NUMBER (5) CONSTRAINT Cust_NotNull NOT NULL,
FIRST_NAME NVARCHAR2 (30) CONSTRAINT Unique_FName UNIQUE,
LAST_NAME NVARCHAR2 (30),
ADDRESS NVARCHAR2 (50),
CITY NVARCHAR2 (30),
```

```
STATE NVARCHAR2 (30) DEFAULT 'California',
ZIP NVARCHAR2 (10),
BIRTH_DATE DATE,
STATUS VARCHAR2 (1) CONSTRAINT Check_Status CHECK(STATUS
IN('V', 'I','A'))
);
```

In the above example, the NOT NULL constraint **Cust_NotNull** ensures that the value for the specified column **CUST_NO** does not allow Null values. Moreover, the UNIQUE constraint **Unique_FName** does not allow duplicate values in the rows of the column **FIRST_NAME**. If you do not enter data values for the column **STATE**, it will take the default value because of the DEFAULT constraint. The **CHECK** constraint **Check_Status** ensures that the data values for the column **STATUS** should be '**V**', '**I**' or '**A**'. If you enter a value other than '**V**', '**I**' or '**A**', then Oracle will throw an error.

## Creating a Table from an Existing Table

You can also create a table from an existing table by copying the columns of an existing table.

The syntax for copying all columns from an existing table is as follows:

```
CREATE TABLE new_table
AS (SELECT * FROM existing_table);
```

In the above syntax, **new_table** is the name of the table to be created from the existing table **existing_table**.

For example:

```
CREATE TABLE Dealer
AS (SELECT * FROM Customer);
```

The above SQL statement will create a new table called **Dealer**. This new table will include all columns of the **Customer** table.

If the **Customer** table has records, the new table **Dealer** will also contain the records selected by the **SELECT** statement.

The syntax for copying the selected columns of an existing table is as follows:

```
CREATE TABLE new_table
AS (SELECT column1, column2, ... column_n FROM existing_table);
```

In the above syntax, **new_table** is the name of the new table to be created; **existing_table** is the name of the existing table; and **column1**, **column2**, ... **column_n** represent the column names.

For example:

```
CREATE TABLE Dealer
AS (SELECT ID, Address, City, State, Country FROM Customer);
```

The above SQL statement will create a new table called **Dealer**. But, the new table will only include the specified columns of the **Customer** table.

Again, if the **Customer** table has records, the new table **Dealer** will also contain the records selected by the **SELECT** statement.

You can copy the selected columns from the multiple tables, by using the following syntax:

```
CREATE TABLE new_table
AS (SELECT column_1, column2, ... column_n
FROM old_table_1, old_table_2, ... old_table_n);
```

For example:

```
CREATE TABLE Emp_Dept
AS (SELECT Emp.Empno, Emp.Ename, Emp.Sal, Emp.Job,
Dept.Deptno, Dept.Dname, Dept.Loc
FROM Emp, Dept
WHERE Emp.Deptno = Dept.Deptno AND Emp.Empno < 10);
```

The above SQL statement will create a new table, called **Emp_Dept**, based on the columns from both the **Emp** and **Dept** tables.

You can also copy the structure of an existing table by using the following syntax:

```
CREATE TABLE new_table
AS (SELECT * FROM old_table WHERE 1=2);
```

For example:

```
CREATE TABLE Emp_Audit
AS (SELECT * FROM Employee WHERE 1=2);
```

The above SQL statement will create a new table, called **Emp_Audit**. This table will contain all columns of the table **Employee**, except the data rows.

You can copy the selected columns from an existing table, excluding the data, by using the following syntax:

```
CREATE TABLE new_table
AS (SELECT column_1, column2, ... column_n FROM existing_table
WHERE 1=2);
```

For example:

```
CREATE TABLE Dealer
AS (SELECT ID, Address, City, State, Country FROM Customer
WHERE1=2);
```

The above SQL statement will create a new table, called **Dealer**. This new table will include only the specified columns of the table **Customer**, except the data rows.

# MODIFYING AND DELETING A DATABASE TABLE

You can modify the structure of an existing database table by using the **ALTER** command. This command is used to add new columns, modify existing columns, change the width of a data type, and add or drop integrity constraints. You can also delete an existing table by using the **DELETE** command.

In this section, you will learn how to delete and rename an existing table, add or delete columns from it, and modify its definition and constraints.

## Deleting and Renaming Existing Tables

You can use the **DROP TABLE** command to delete or remove the database tables. The syntax for using the **DROP TABLE** command is as follows:

```
DROP TABLE Table_Name;
```

In the above syntax, **DROP** and **TABLE** are keywords and **Table_Name** is the name of the table to be deleted or removed from the database.

If any column of the table **Table_Name** has a reference in another table, the **DROP TABLE** command cannot delete or remove the table **Table_Name** from the database.

To drop a table that has a reference in another table, you can use the following two methods:

The first method is to delete or remove all tables that have foreign key references with other tables.

The second method is that you have to drop all references or foreign key constraints that refer to other table. To avoid such a situation, Oracle provides the **DROP TABLE** command with the **CASCADE CONSTRAINTS** option. The **CASCADE CONSTRAINTS** option is used to delete the table that has the foreign key constraint references. The syntax for using the **CASCADE CONSTRAINTS** option with the **DELETE** statement is as follows:

```
DROP TABLE Table_Name CASCADE CONSTRAINTS;
```

In the above syntax, **DROP**, **TABLE**, **CASCADE**, and **CONSTRAINTS** are keywords and **Table_Name** is the name of the table to be deleted or removed from the database.

For example:

```
DROP TABLE Customer;
```

After executing the above statement, Oracle will throw an error because of its foreign key reference with the **Dealer** table. Therefore, to remove the table **Customer**, you need to use the **DROP TABLE** command with the **CASCADE CONSTRAINTS** option, as given below:

```
DROP TABLE Customer CASCADE CONSTRAINTS;
```

Now, the table will be deleted.

The **RENAME** command is used to rename an existing database table. The syntax for using the **RENAME** command is as follows:

```
RENAME Old_Table_Name TO New_Table_Name;
```

In the above syntax, **RENAME** and **TO** are keywords; **Old_Table_Name** is the name of the table to be renamed; and **New_Table_Name** is the new name for the **Old_Table_Name** table.

For example:

To rename a table **Emp** to **Employee**, you need to follow the steps given below:

1. To rename the **Emp** table, enter the following command at the SQL prompt.

   ```
   RENAME Emp TO Employee;
   ```

2. Press ENTER to execute the above statement. After executing the statement, Oracle will return a message, **Table renamed**, which confirms that the table name **Emp** is replaced with **Employee**.

## Adding and Modifying Existing Columns

You can add new columns to the existing table by using the **ALTER TABLE** command with the **ADD** option. The syntax for using the **ALTER TABLE** command with the **ADD** option is as follows:

```
ALTER TABLE Table_Name ADD(Column_Name Column_Type Constraints);
```

In the above syntax, **Table_Name** is the name of an existing table to which you want to add the new column; **Column_Name** is name of the new column; **Column_Type** is the data type of the new column **Column_Name**; and **Constraints** is any constraint that you want to set for the new column.

For example:

To add a new column **STATUS** to the **Customer** table, you need to follow the steps given below:

1. In SQL *Plus, enter the **ALTER TABLE** command, as shown in Figure 2-20, to add a new column in the table **Customer**.

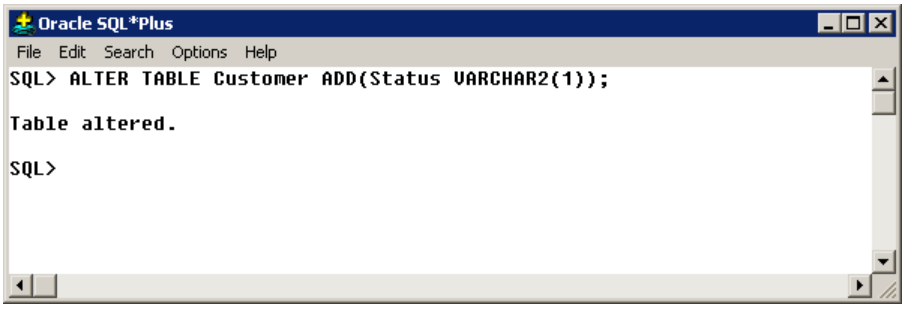



***Figure 2-20*** *The **ALTER TABLE** command with the **ADD** option*

2. Press ENTER to execute the above statement. On executing the statement, Oracle will return a message, **Table altered**, which confirms that the new column **STATUS** has been added to the table **Customer**.

You can also modify a column of an existing table by using the **ALTER TABLE** command with the **MODIFY** option. The syntax for using the **ALTER TABLE** command with the **MODIFY** option is as follows:

ALTER TABLE Table_Name MODIFY(Column_Name New_Data_Definition);

In the above syntax, **Table_Name** is the name of the existing table to be modified; **Column_Name** is the name of the column of the existing table that you want to modify; and **New_Data_Definition** is the new data type definition of the existing column.

For example:

To change the width of the column **Cust_Add** of the table **Customer**, you need to follow the steps given below:

1. In SQL *Plus, enter the **ALTER TABLE** command, as shown in Figure 2-21, to modify the column **Cust_Add** of the table **Customer**.

2. Press ENTER to execute the statement. After executing the statements, Oracle will return a message **Table altered**, refer to Figure 2-21, which confirms that the column **Cust_Add** of the table **Customer** has been modified.

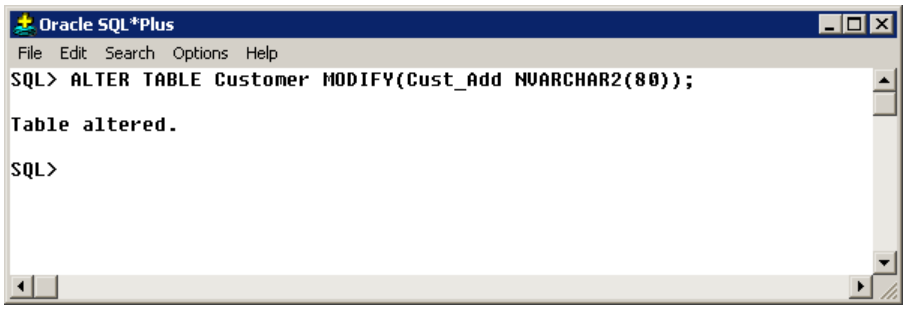



*Figure 2-21 The **ALTER TABLE** command with the **MODIFY** option*

## Deleting and Renaming the Columns of an Existing Table

You can delete an existing column from the database table by using the **ALTER TABLE** command with the **DROP** option. The syntax for using the **ALTER TABLE** command with the **DROP** option is as follows:

```
ALTER TABLE Table_Name DROP COLUMN Column_Name;
```

In the above syntax, **Table_Name** is the name of the existing table from which you want to delete or remove a column and **Column_Name** specifies the name of the column that you want to delete or remove from the table **Table_Name**.

For example:

To delete the column **PIN** from the table **Customer**, you need to follow the steps given below:

1. Enter the following command in SQL *Plus to delete the column from the table:

   ```
   ALTER TABLE Customer DROP COLUMN ZIP;
   ```

2. Press ENTER to execute the above statement. After executing the statement, Oracle will return a message, **Table altered**, which confirms that the column **ZIP** has been deleted from the table **Customer**.

You can also rename an existing column in the table by using the **ALTER TABLE** command with the **RENAME** option.

The syntax for using the **ALTER TABLE** command with the **RENAME** option is as follows:

```
ALTER TABLE Table_Name RENAME COLUMN Old_Column_Name TO
New_Column_Name
```

In the above syntax, **Table_Name** is the name of the table in which you want to rename a column. Here, **Old_Column_Name** is the name of column that you want to rename and **New_Column_Name** is the new name of the **Old_Column_Name** column.

For example:

To rename the column **Cust_Address** to **Cust_Add** in the table **Customer**, you need to follow the steps given below:

1. Enter the following command in SQL *Plus to rename the column in the table **Customer**:

   ALTER TABLE Customer RENAME COLUMN Cust_Add TO Cust_Address;

2. Press ENTER to execute the above statement; Oracle will return a message, **Table altered**, which confirms that the table **Customer** has been altered.

3. To check whether the name of the column has been changed, enter the following statement in SQL *Plus, as shown in Figure 2-22.

   SQL>DESC Customer;

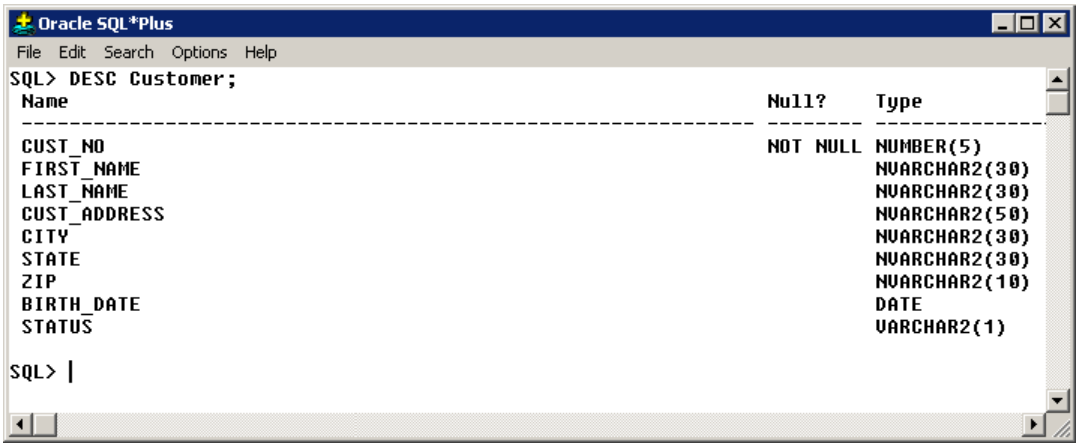



***Figure 2-22*** *The description of the table* ***Customer***

On doing so, the description of the **Customer** table will be displayed, refer to Figure 2-22 with the name of column **Cust_Address** modified.

## Adding and Deleting Constraints

You can remove an existing constraint from a database table by using the **ALTER TABLE** command with the **DROP** option.

The syntax for using the **ALTER TABLE** command with the **DROP** option is as follows:

ALTER TABLE Table_Name DROP CONSTRAINT Constraint_Name;

In the above syntax, **Table_Name** is the name of an existing table from which you want to remove the constraint and **Constraint_Name** is the name of the constraint that you want to remove.

For example:

To delete the constraint **CHECK** named **Check_Status** from the table **Customer**, you need to follow the steps given below:

1. Enter the following command in SQL *Plus to delete the constraint **Check_Status** from the table **Customer**:

```
ALTER TABLE Customer DROP CONSTRAINT Check_Status;
```

2. Press ENTER to execute the above statement; Oracle will return a message, **Table altered**, which confirms that the **Check_Status** constraint has been removed from the table **Customer**.

On doing so, the description of the table **Customer** will be displayed without the constraint **Check_Status**.

You can also add a constraint to the existing column of the table by using the **ALTER** command with the **ADD** option. The syntax for using the **ALTER** command with the **ADD** option is as follows:

```
ALTER TABLE Table_Name ADD CONSTRAINT Constraint_Name
Constraint_declaration;
```

In the above syntax, **Table_Name** is the name of the table to which you want to add a constraint; **Constraint_Name** is the name of the new constraint; and **Constraint_Declaration** specifies the constraint type.

For example:

To add the primary key constraint to the table **Customer**, enter the following statement in SQL *Plus:

```
ALTER TABLE Customer ADD CONSTRAINT Primary_Key_Id PRIMARY
KEY (Cust_No);
```

To check whether the constraint has been added, enter the following statement in SQL *Plus:

```
SQL>DESC Customer;
```

On doing so, the description of the table **Customer** will be displayed with the primary key constraint added to it.

Similarly, to add the foreign key constraint to the table **Customer**, you need to enter the following statements in SQL *Plus:

```
ALTER TABLE Customer ADD CONSTRAINT Foreign_key
FOREIGN KEY (Cust_No) REFERENCES Supplier ON DELETE CASCADE;
```

## Enabling and Disabling Constraints

You can enable or disable the constraints by using the **ALTER** command with the **ENABLE** and **DISABLE** option. The syntax for using the **ALTER** command with the **ENABLE** option is as follows:

```
ALTER TABLE Table_Name ENABLE CONSTRAINT Constraint_Name;
```

In the above syntax, **Table_Name** is the name of the table on which you want to enable the constraints and **Constraint_Name** is the name of the constraint to be enabled.

The syntax for using the **ALTER** command with the **DISABLE** option is as follows:

```
ALTER TABLE Table_Name DISABLE CONSTRAINT Constraint_Name;
```

In the above syntax, **Table_Name** is the name of the table on which you want to disable the constraints and **Constraint_Name** is the name of the constraint to be disabled.

For example:

To enable the primary key constraint of the table **Customer**, enter the following statements in SQL *Plus:

```
ALTER TABLE Customer ENABLE CONSTRAINT Cust_PrimaryKey;
```

After the execution of the above statement, the **Cust_PrimaryKey** constraint will be enabled. Similarly, enter the following statement in SQL *Plus to disable the primary key constraint:

```
ALTER TABLE Customer DISABLE CONSTRAINT Cust_PrimaryKey;
```

After the execution of the above statement, the **Cust_PrimaryKey** constraint will be disabled.

The following example will illustrate how to create a table and then add columns and constraints, alter columns and constraints, drop columns and constraints in it.

## Example 3

Write a query to create a table **Employee** and use the **ALTER** and **DROP** commands to add and drop columns and constraints in it.

The following steps are required to create the table and then alter, modify, and drop constraints and columns in it.

1. Enter the following command in SQL *Plus to create the table **Employee**:

```
CREATE TABLE Employee (
        Empno NUMBER(4),
        Ename VARCHAR2(10),
         Job VARCHAR2(9),
        MGR NUMBER(4),
        HireDate DATE,
        Salary NUMBER(7, 2),
        Comm NUMBER(7, 2),
        Deptno NUMBER(2)
        );
```

The above command **CREATE TABLE** will create the table **Employee**, as shown in Figure 2-23.

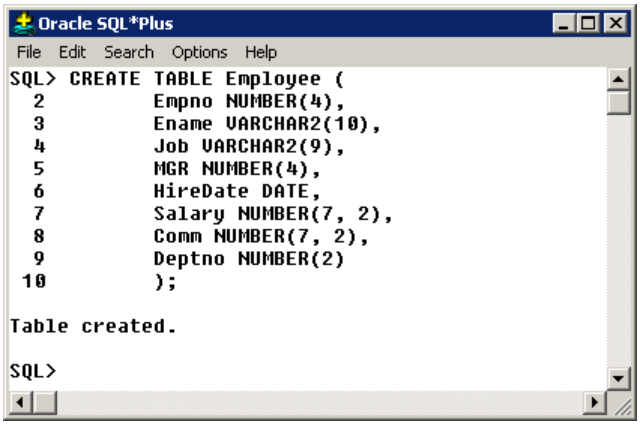



***Figure 2-23*** *Creating the table* ***Employee***

2. Now, you can add the column **Address** in the **Employee** table by entering the following command in SQL *Plus:

```
ALTER TABLE Employee ADD(Address NVARCHAR2(80));
```

The above command will add the new column **Address** to the table **Employee**, as shown in Figure 2-24.

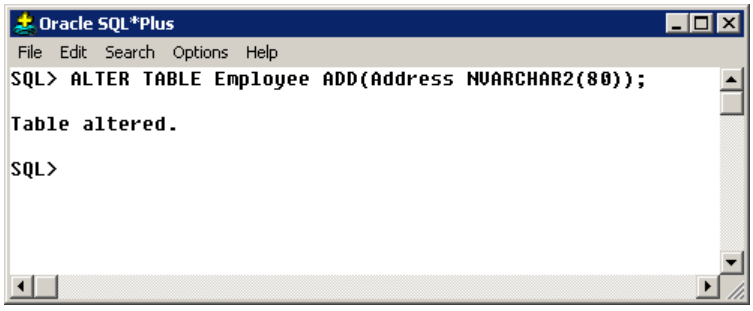



***Figure 2-24*** *Adding new column to the* ***Employee*** *table*

3. You can also modify the width of the column **Ename** in the **Employee** table by entering the following command in SQL *Plus:

   ALTER TABLE Employee MODIFY(Ename NVARCHAR2(20));

   The above command will modify the column **Ename** of the table **Employee**, as shown in Figure 2-25.

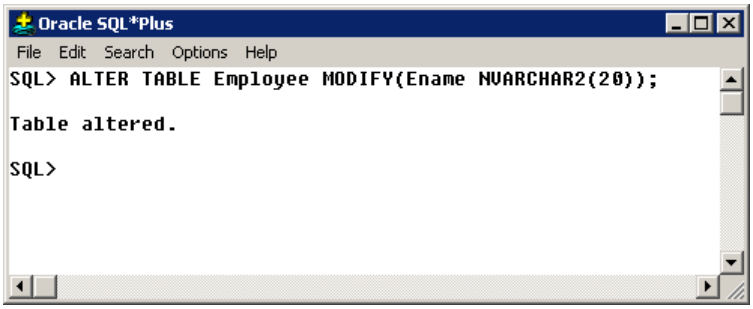



***Figure 2-25*** *Modifying the* ***Ename*** *column of the table* ***Employee***

4. You can also rename the column **Address** to **Emp_Address** of the **Employee** table by entering the following command in SQL *Plus:

   ALTER TABLE Employee RENAME COLUMN Address TO Emp_Address;

   The above command will rename the column **Address** to **Emp_Address** of the table **Employee**, as shown in Figure 2-26.

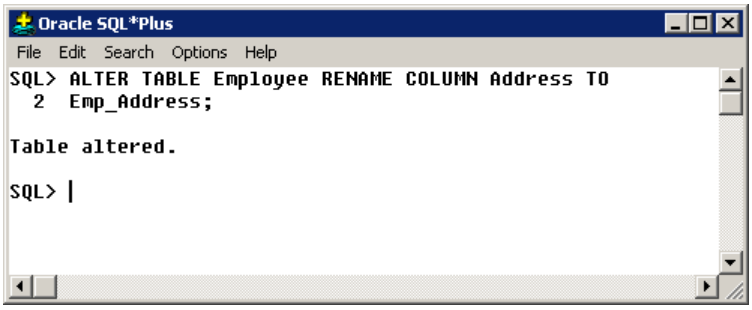



***Figure 2-26*** *Renaming the column* ***Address*** *to* ***Emp_Address*** *of the* ***Employee*** *table*

5. After creating the table **Employee**, you can add the primary key constraint to the column **Empno**. To do so, enter the following command in SQL *Plus:

   ALTER TABLE Employee ADD CONSTRAINT Prmy_Empno PRIMARY KEY (Empno);

The above command will add the primary key **Prmy_Empno** to the table **Employee** on the column **Empno**, as shown in Figure 2-27.

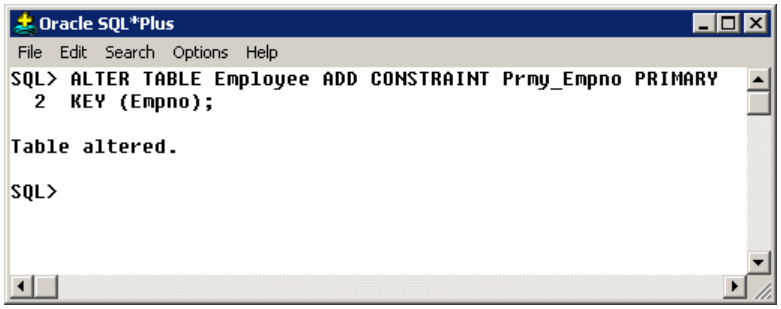



***Figure 2-27*** *Adding the primary key constraint to the* ***Employee*** *table*

6. Now, you can disable or enable constraints on the table **Employee** by entering the following command in SQL *Plus:

ALTER TABLE Employee DISABLE CONSTRAINT Prmy_Empno;

The above command will disable constraint **Prmy_Emp** of the table **Employee** in the column **Empno**, as shown in Figure 2-28.

ALTER TABLE Employee ENABLE CONSTRAINT Prmy_Empno;

The above command will enable the constraint **Prmy_Emp** of the table **Employee** on the column **Empno**, refer to Figure 2-28.

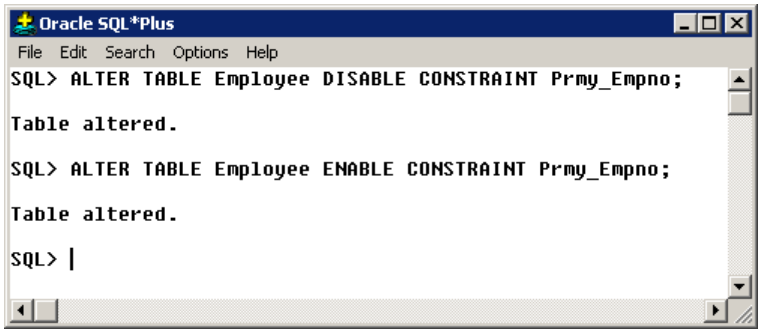



***Figure 2-28*** *Disabling and enabling the constraint* ***Prmy_Empno***

## Self-Evaluation Test

**Answer the following questions and then compare them to those given at the end of this chapter:**

1. The _____ command is used to read the contents of the SQL buffer.

2. The _____ command is used to view the structure of a database table.

3. The / command is used to execute the current command in the _____.

4. The NOT NULL constraint is a column level constraint. (T/F)

5. The DEFAULT constraint is used in a column to ensure that a Null value is not contained in that column. (T/F)

6. The **APPEND** command is used to find and replace a string in the current line of the SQL buffer. (T/F)

7. The **START** command is used to execute the contents of a file. (T/F)

8. Which of the following commands is used to add lines to the existing command in the buffer?

   (a) **GET** (b) **START**
   (c) **INPUT** (d) All the above

9. Which of the following commands is used to display the content of the buffer?

   (a) **EDIT** (b) **CHANGE**
   (c) **LIST** (d) None of these

10. Which of the following is character data type?

    (a) NVARCHAR2 (b) CHAR
    (c) Both (a) & (b) (d) None of these

## Review Questions

**Answer the following questions:**

1. The BINARY_DOUBLE value requires _____ bytes.

2. The BINARY_FLOAT value requires _____ bytes.

3. The BLOB stands for _____.

4. The _____ command is used to change the name of a column.

5. The _____ command is used to delete the current line from the buffer.

6. BLOB data type can store data up to _____ in length.

7. Which of the following **ALTER TABLE** options is used to remove a column from a database table?

   (a) **DROP** (b) **MODIFY**
   (c) **DELETE** (d) All the above

8. Which of the following constraints allows a Null value?

   (a) **DROP** (b) **MODIFY**
   (c) **DELETE** (d) All the above

9. Which of the following commands is used to exit from the SQL *Plus or from command line?

   (a) **QUIT** (b) **END**
   (c) **EXIT** (d) All the above

10. Which of the following is the default date format of the DATE data type?

   (a) DD-MM-YY (b) DD-MON_YYYY
   (c) DD-MON-YY (d) None of these

## *Exercise*

### Exercise 1

Create a table with the name **Employee** having the following columns **Empid**, **Ename**, **Designation**, **Salary**, **Commission**, **Deptno**. Also, declare a primary key constraint for the **Empid** column.

### Answers to Self-Evaluation Test

**1. GET**, **2. DESC**, **3.** SQL buffer, **4.** T, **5.** F, **6.** F, **7**. T, **8.** c, **9.** c, **10.** c