Practical Workbook
CS-222
Database Management Systems

Name : ______________________________

Year : ______________________________

Batch : ______________________________

Roll No : ______________________________

Department: ______________________________

Department of Computer & Information Systems Engineering
NED University of Engineering & Technology

Practical Workbook
CS-222
Database Management Systems

Prepared by:

Muhammad Ali Akhtar, Hameeza Ahmed, Kashif Asrar

Revised in:

February 2021
Revision No.2

Department of Computer & Information Systems Engineering
NED University of Engineering & Technology

INTRODUCTION

This workbook has been compiled to assist the conduct of practical classes for CS-222 Database Management Systems. Practical work relevant to this course aims at teaching the basic concepts as well as advanced techniques in the management of databases of an organization. The creation and manipulation of database objects requires programming in Structured Query Language (SQL). SQL is a nonprocedural language with capabilities of creating and manipulating data in tables and views. SQL is referred to as a query language because it has wide range of facilities to retrieve data from a database.

The Course Profile of CS-222 Database Management Systems lays down the following Course Learning Outcome:

"**Demonstrate** the use of modern querying tools for database management (C3, PLO-5)"

All lab sessions of this workbook have been designed to assist the achievement of the above CLO. A rubric to evaluate student performance has been provided at the end of the workbook.

Lab sessions 1 & 2 gives an overview for a modern tool SQL developer. Lab session 3 covers the basic data retrieval operations in SQL. Lab session 4 covers the join operations in SQL. Lab session 5 demonstrates single and multiple row functions in SQL. Lab session 6 deals with sub queries and compound queries in SQL. Lab session 7 elaborates the basic data manipulation operations in SQL. Lab session 8 explains the creation and management of tables and views in SQL. Lab session 9 presents sequences, indexes, and synonyms. Lab session 10 covers basics of PL/SQL programming. Lab session 11 deals with control structures and exception handling in PL/SQL. Lab session 12 demonstrates trigger in database. Lab session 13 deals with stored procedures and functions in database. Lab session 14 discusses the complex engineering activity.

CONTENTS

Lab Session 01

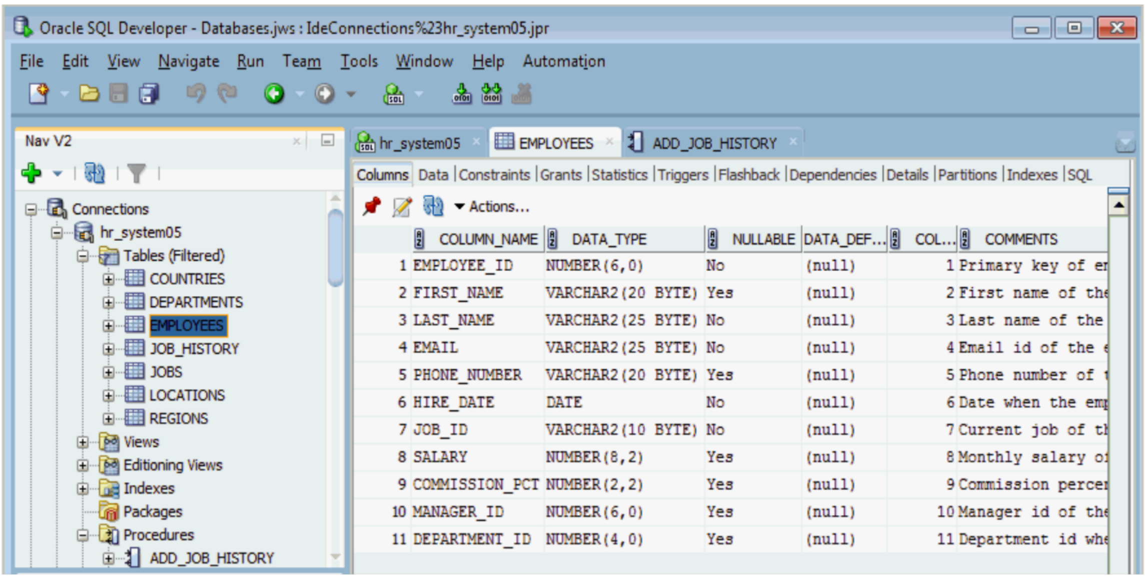
Explore SQL Developer Tool

INTRODUCTION

Oracle SQL Developer is a graphical version of SQL*Plus that gives database developers a convenient way to perform basic tasks. It allows browsing, creation, editing, and deletion. The developer can run SQL statements & scripts, edit & debug PL/SQL code, manipulate & export (unload) data, and view & create reports. It can connect to any target Oracle database schema using standard Oracle database authentication. Once connected, it can perform operations on objects in the database. Also, it can connect to schemas for MySQL and selected third-party (non-Oracle) databases, such as Microsoft SQL Server, Sybase Adaptive Server, and IBM DB2, and view metadata and data in these databases along with allowing migration of these databases to Oracle database.

SQL Developer User Interface

The SQL Developer window generally uses the left side for navigation to find and select objects, and the right side to display information about selected objects.



	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEF...	COL...	COMMENTS
1	EMPLOYEE_ID	NUMBER(6,0)	No	(null)	1	Primary key of em
2	FIRST_NAME	VARCHAR2(20 BYTE)	Yes	(null)	2	First name of the
3	LAST_NAME	VARCHAR2(25 BYTE)	No	(null)	3	Last name of the
4	EMAIL	VARCHAR2(25 BYTE)	No	(null)	4	Email id of the e
5	PHONE_NUMBER	VARCHAR2(20 BYTE)	Yes	(null)	5	Phone number of t
6	HIRE_DATE	DATE	No	(null)	6	Date when the emp
7	JOB_ID	VARCHAR2(10 BYTE)	No	(null)	7	Current job of th
8	SALARY	NUMBER(8,2)	Yes	(null)	8	Monthly salary of
9	COMMISSION_PCT	NUMBER(2,2)	Yes	(null)	9	Commission percen
10	MANAGER_ID	NUMBER(6,0)	Yes	(null)	10	Manager id of the
11	DEPARTMENT_ID	NUMBER(4,0)	Yes	(null)	11	Department id whe

Figure 1.1: SQL Developer Main Window

The menus at the top contain standard entries, plus entries for features specific to SQL Developer as shown in the following figure.

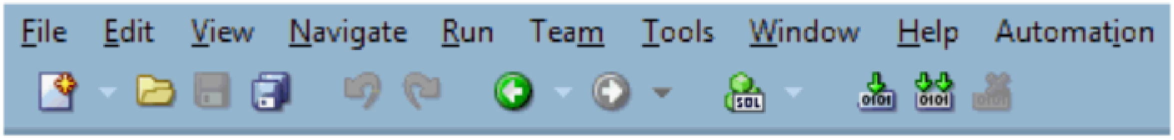


Figure 1.2: SQL Developer Main Menus

Shortcut keys can be used to access menus and menu items: for example **Alt+F** for the File menu and **Alt+E** for the Edit menu; or **Alt+H**, then **Alt+S** for Help, then Search. Also, it allows displaying the File menu by pressing the **F10** key (except in the SQL Worksheet, where F10 is the shortcut for Explain Plan). To close a window that has focus (such as the SQL Developer main window, a wizard or dialog box, or the Help Center) and any of its dependent windows, Alt+F4 can be pressed. The main toolbar (under the menus) contains icons to perform various actions, which by default include the following:

- *New* creates a new database object (see Create New Object).
- *Open* opens a file (see Open File).
- *Save* saves any changes to the currently selected object.
- *Save All* saves any changes to all open objects.
- *Back* moves to the pane that has been most recently visited. (Or use the drop-down arrow to specify a tab view.)
- *Forward* moves to the pane after the current one in the list of visited panes. (Or use the drop-down arrow to specify a tab view.)
- *Open SQL Worksheet* opens the SQL Worksheet (see using the SQL Worksheet). If the drop-down arrow is not used to specify the database connection, it will ask to select a connection.

The main toolbar icons can be added and removed by clicking View, Show Toolbars, Customize Main Toolbar, then choosing desired options. In order to undo any main toolbar customization, Reset Toolbars is selected. The left side of the SQL Developer window has panes for the Connections and Reports navigators (any other navigators that you might open), icons for performing actions, and a hierarchical tree display for the currently selected navigator, as shown in the following figure.

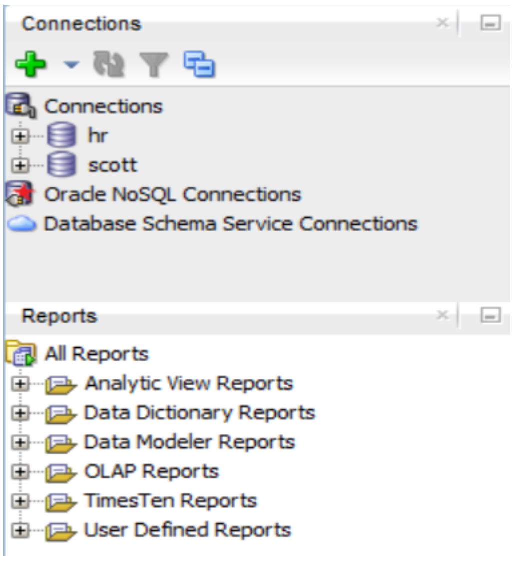


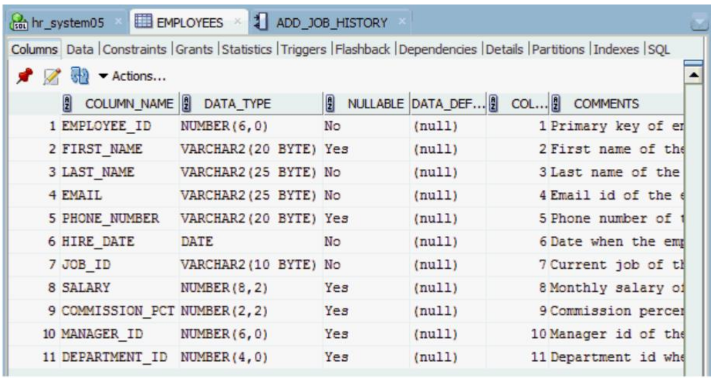
Figure 1.3: SQL Developer Connections and Reports

The ***Connections navigator*** lists database connections that have been created. To create a new database connection, import, export or edit current connections, right-click the Connections node and select the appropriate menu item.

The ***Files navigator*** (marked by a folder icon; not shown in the preceding figure) displays the local file system using a standard hierarchy of folders and files. The double-click or drag and drop can be used to open, edit, and save files. For example, if a .sql file is opened, it is displayed in a SQL Worksheet window. The Files navigator is especially useful if versioning is used with SQL Developer.

The ***Reports navigator*** (not shown in the preceding figure) lists informative reports provided by SQL Developer, such as a list of tables without primary keys for each database connection, as well as any user-defined reports. (For more information, see SQL Developer Reports.)

The right side of the SQL Developer window has tabs and panes for objects that can be selected or opened, as shown in the following figure, which displays information about a table named EMPLOYEES.



	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEF...	COL...	COMMENTS
1	EMPLOYEE_ID	NUMBER(6,0)	No	(null)	1	Primary key of e
2	FIRST_NAME	VARCHAR2(20 BYTE)	Yes	(null)	2	First name of th
3	LAST_NAME	VARCHAR2(25 BYTE)	No	(null)	3	Last name of the
4	EMAIL	VARCHAR2(25 BYTE)	No	(null)	4	Email id of the
5	PHONE_NUMBER	VARCHAR2(20 BYTE)	Yes	(null)	5	Phone number of
6	HIRE_DATE	DATE	No	(null)	6	Date when the em
7	JOB_ID	VARCHAR2(10 BYTE)	No	(null)	7	Current job of t
8	SALARY	NUMBER(8,2)	Yes	(null)	8	Monthly salary o
9	COMMISSION_PCT	NUMBER(2,2)	Yes	(null)	9	Commission perce
10	MANAGER_ID	NUMBER(6,0)	Yes	(null)	10	Manager id of th
11	DEPARTMENT_ID	NUMBER(4,0)	Yes	(null)	11	Department id wh

Figure 1.4: SQL Developer selected Object Information

EXERCISES

1. Create a connection of your name and seat number. Attach screenshot of every step.

2. Write down short keys of all operations performed in lab session 01. For example to insert row in a table we use (CTRL + I).

3. After following all steps given below install Oracle SQL Developer in your system. Attach the screenshots of each step.

Installing and Getting Started with SQL Developer

A. Unzip the SQL Developer kit into a folder (directory) of your choice, which will be referred to as <sqldeveloper_install>. Unzipping the SQL Developer kit causes a folder named sqldeveloper to be created under the <sqldeveloper_install> folder.

For example, if you unzip the kit into C:\, the folder C:\sqldeveloper is created, along with several subfolders under it.
If Oracle Database (Release 11 or later) is also installed, a version of SQL Developer is also included and is accessible through the menu system under Oracle. This version of SQL Developer is separate from any SQL Developer kit that you download and unzip on your own, so do not confuse the two, and do not unzip a kit over the SQL Developer files that are included with Oracle Database. Suggestion: Create a shortcut for the SQL Developer executable file that you install, and always use it to start SQL Developer.

B. To start SQL Developer, go to the sqldeveloper directory under the <sqldeveloper_install> directory, and do one of the following:

- On Linux and Mac OS X systems, run sh sqldeveloper.sh.
- On Windows systems, double-click sqldeveloper.exe.
- If you are asked to enter the full pathname for the JDK, click Browse and find it. For example, on a Windows system the path might have a name similar to C:\Program Files\Java\jdk1.7.0_51. (If you cannot start SQL Developer, it could be due to an error in specifying or configuring the JDK. See Java Development Kit (JDK) Location for more information.)

C. Create at least one database connection (or import some previously exported connections), so that you can view and work with database objects, use the SQL Worksheet, and use other features.

4. By default HR account is disabled in oracle. Unlock the HR account and reset its password after taking help from Oracle Docs link given below:
https://docs.oracle.com/cd/E11882_01/appdev.112/e10766/tdddg_connecting.htm#TDDDG12510
Connecting to the oracle database as user **"HR"** and explore all the tables present in the schema. Attach screenshot of all the tables.

Lab Session 2

Apply basic (crude) SQL operations on SQL Developer

Creating New Connection

A new database connection can be created for a Predefined user "**Scott**" with Password "**tiger**" using the following steps

1) Click on Plus sign in Connection Tab

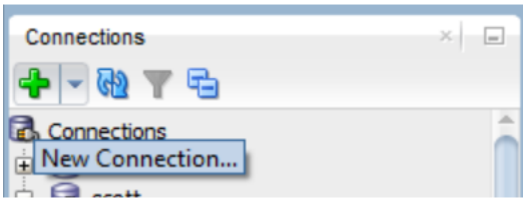
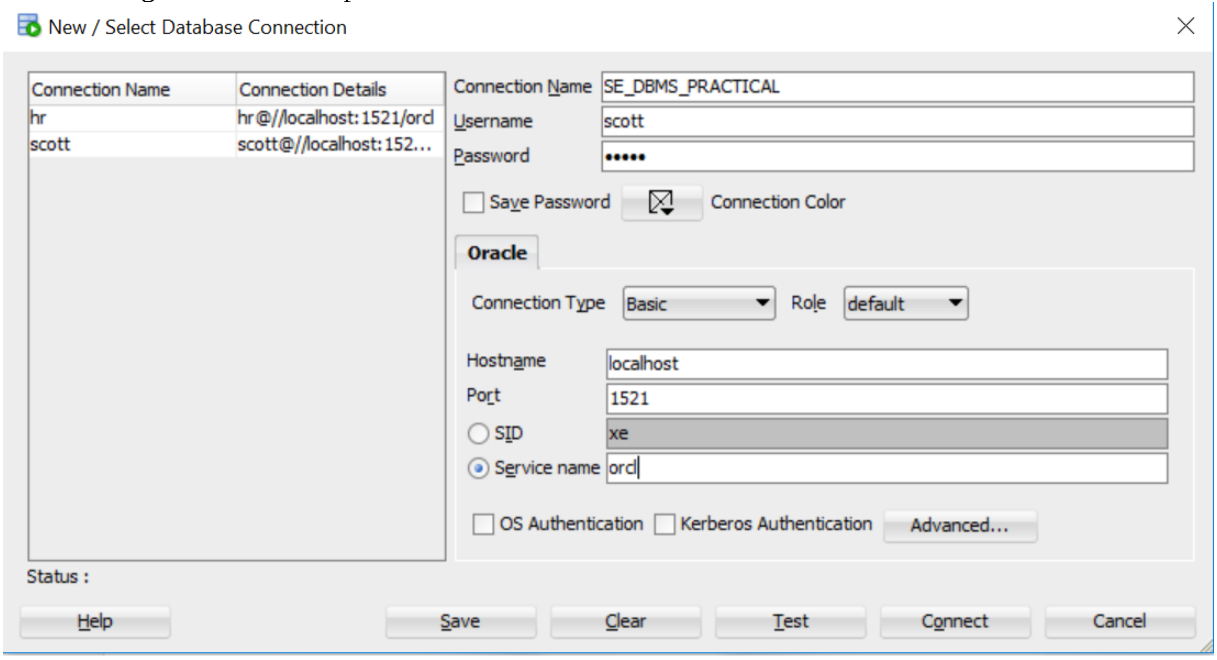


Figure 2.1: Connections

2) Fill Connection name of choice under predefined Username and password
3) Select hostname as a server where database is being installed. As the testing is done on local machines so write Localhost there.
4) Service for running Oracle in Our Operating systems is ORCL so ORCL service is selected after selecting Service name option as shown below.



New / Select Database Connection

Connection Name	Connection Details
hr	hr@//localhost:1521/orcl
scott	scott@//localhost:152...

Connection Name: SE_DBMS_PRACTICAL
Username: scott
Password: •••••
□ Save Password
Connection Color

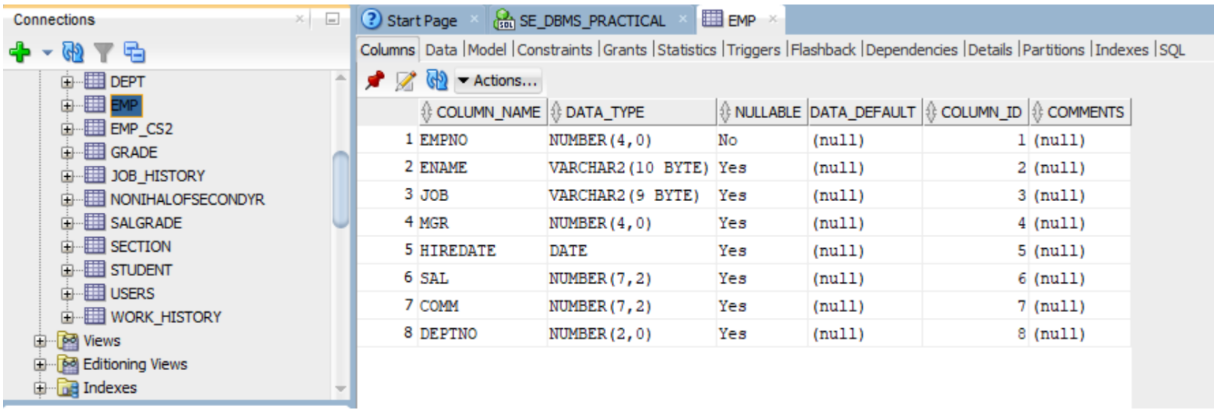
Oracle
Connection Type: Basic
Role: default
Hostname: localhost
Port: 1521
○ SID: xe
● Service name: orcl
□ OS Authentication □ Kerberos Authentication
Status :

Figure 2.2: Creating Connection

5) Finally click on Test to check the connection with database.
6) On success status above the help button in figure ***Status: Success*** is shown.
7) In your connection Tab you will find New Connection with your defined name.
8) Now you can access all tables and can perform all allowable operations.

Accessing Table and its Records

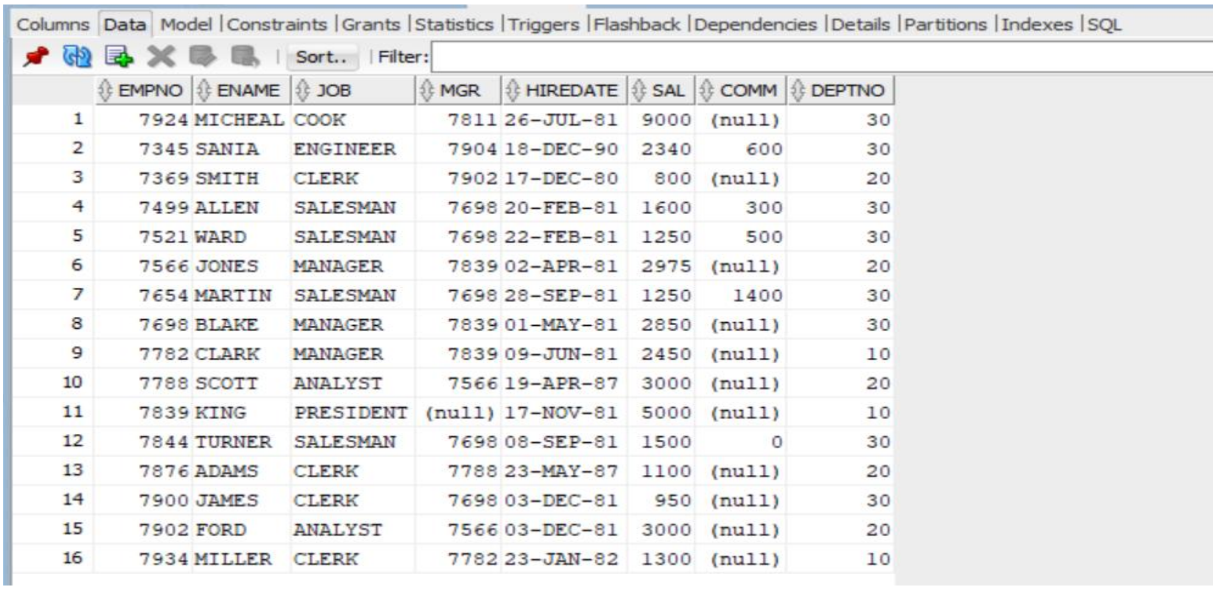
1) Click on table name under connection (employee Table).



	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	EMPNO	NUMBER(4,0)	No	(null)	1	(null)
2	ENAME	VARCHAR2(10 BYTE)	Yes	(null)	2	(null)
3	JOB	VARCHAR2(9 BYTE)	Yes	(null)	3	(null)
4	MGR	NUMBER(4,0)	Yes	(null)	4	(null)
5	HIREDATE	DATE	Yes	(null)	5	(null)
6	SAL	NUMBER(7,2)	Yes	(null)	6	(null)
7	COMM	NUMBER(7,2)	Yes	(null)	7	(null)
8	DEPTNO	NUMBER(2,0)	Yes	(null)	8	(null)

Figure 2.3: Employee Table Structure

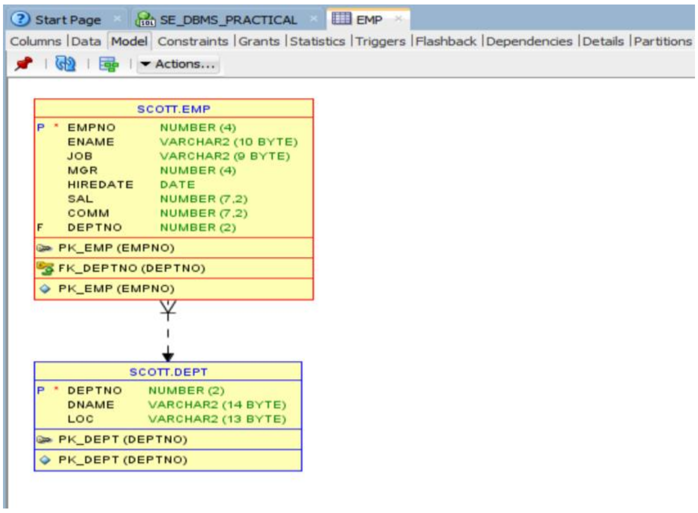
2) To view Records in that table Data tab is clicked as shown below.



	EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
1	7924	MICHEAL	COOK	7811	26-JUL-81	9000	(null)	30
2	7345	SANIA	ENGINEER	7904	18-DEC-90	2340	600	30
3	7369	SMITH	CLERK	7902	17-DEC-80	800	(null)	20
4	7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
5	7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
6	7566	JONES	MANAGER	7839	02-APR-81	2975	(null)	20
7	7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
8	7698	BLAKE	MANAGER	7839	01-MAY-81	2850	(null)	30
9	7782	CLARK	MANAGER	7839	09-JUN-81	2450	(null)	10
10	7788	SCOTT	ANALYST	7566	19-APR-87	3000	(null)	20
11	7839	KING	PRESIDENT	(null)	17-NOV-81	5000	(null)	10
12	7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
13	7876	ADAMS	CLERK	7788	23-MAY-87	1100	(null)	20
14	7900	JAMES	CLERK	7698	03-DEC-81	950	(null)	30
15	7902	FORD	ANALYST	7566	03-DEC-81	3000	(null)	20
16	7934	MILLER	CLERK	7782	23-JAN-82	1300	(null)	10

Figure 2.4: Employee Table Data

3) The relations of the table in database with other tables is depicted via model tab.



SCOTT.EMP

P	*	EMPNO	NUMBER (4)
		ENAME	VARCHAR2 (10 BYTE)
		JOB	VARCHAR2 (9 BYTE)
		MGR	NUMBER (4)
		HIREDATE	DATE
		SAL	NUMBER (7,2)
		COMM	NUMBER (7,2)
F		DEPTNO	NUMBER (2)

PK_EMP (EMPNO)
FK_DEPTNO (DEPTNO)
PK_EMP (EMPNO)

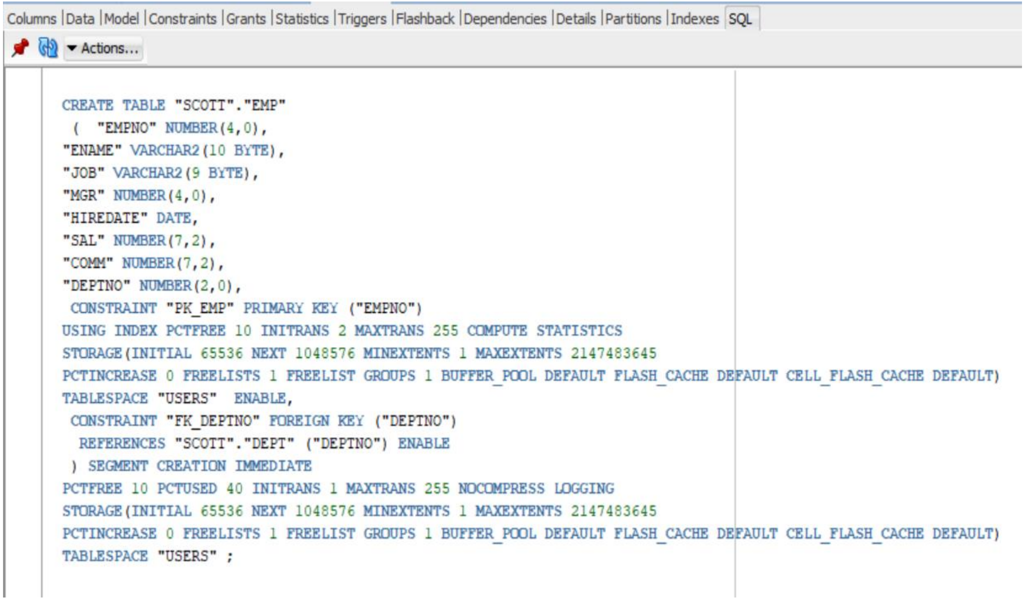
SCOTT.DEPT

P	*	DEPTNO	NUMBER (2)
		DNAME	VARCHAR2 (14 BYTE)
		LOC	VARCHAR2 (13 BYTE)

PK_DEPT (DEPTNO)
PK_DEPT (DEPTNO)

Figure 2.5: Employee Table Model

4) SQL query for create table or any other functions or triggered applied on that table can be seen from SQL tab.



```
CREATE TABLE "SCOTT"."EMP"
 (  "EMPNO" NUMBER(4,0),
"ENAME" VARCHAR2(10 BYTE),
"JOB" VARCHAR2(9 BYTE),
"MGR" NUMBER(4,0),
"HIREDATE" DATE,
"SAL" NUMBER(7,2),
"COMM" NUMBER(7,2),
"DEPTNO" NUMBER(2,0),
 CONSTRAINT "PK_EMP" PRIMARY KEY ("EMPNO")
USING INDEX PCTFREE 10 INITRANS 2 MAXTRANS 255 COMPUTE STATISTICS
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT)
TABLESPACE "USERS"  ENABLE,
 CONSTRAINT "FK_DEPTNO" FOREIGN KEY ("DEPTNO")
  REFERENCES "SCOTT"."DEPT" ("DEPTNO") ENABLE
 ) SEGMENT CREATION IMMEDIATE
PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255 NOCOMPRESS LOGGING
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT)
TABLESPACE "USERS" ;
```

Figure 2.6: Employee Table Query

Creating New Table

1. In the Connections navigation hierarchy, click the plus sign (+) next to newly created connection (SE_DBMS_PRACTICAL) to expand the list of schema objects.
2. Right-click Tables.
3. Select New Table.

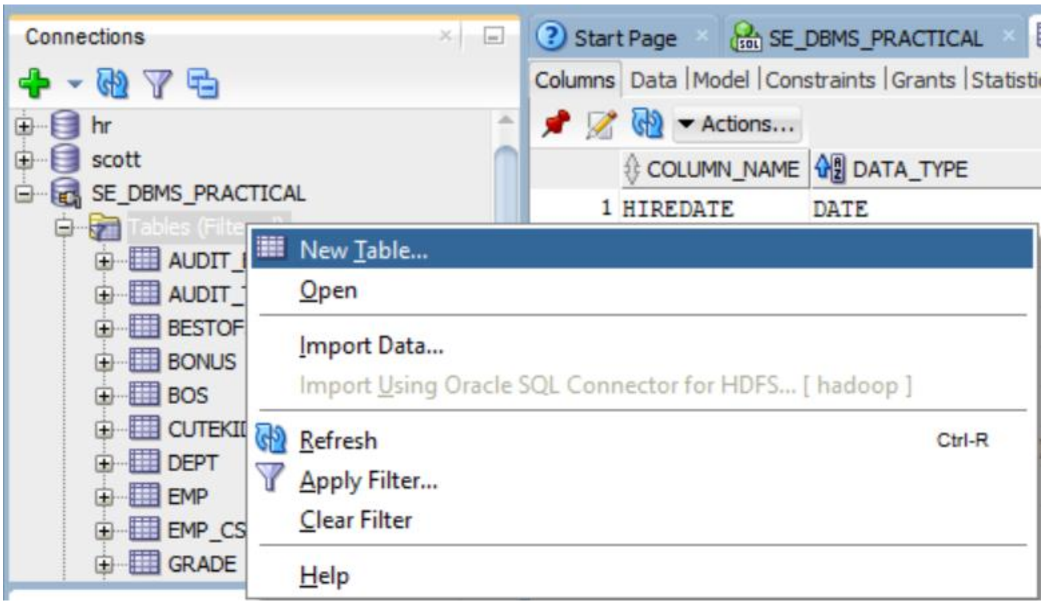
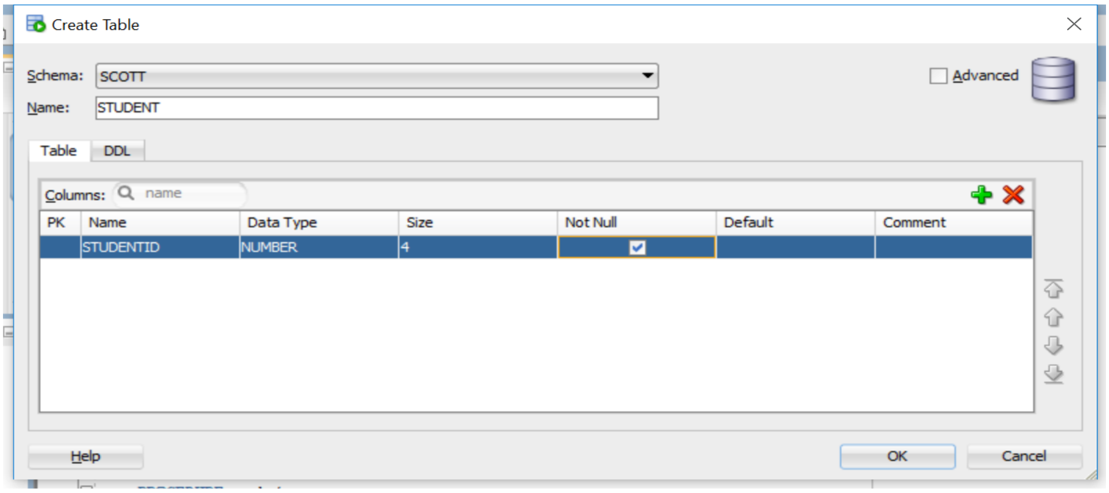


Figure 2.7: Creating a new Table

After selecting, a new window is appeared for creating the table. Here the schema and table name can be set and the DDL can be updated accordingly.



Schema: SCOTT

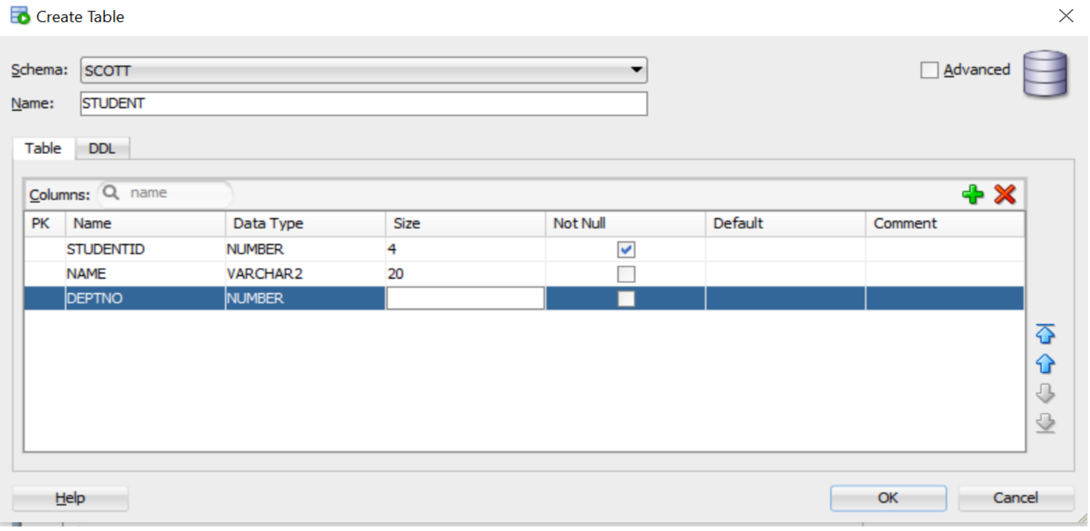
Name: STUDENT

Columns: name

PK	Name	Data Type	Size	Not Null	Default	Comment
	STUDENTID	NUMBER	4	☑		

Figure 2.8

4. Plus sign adds up new columns in the table
5. Press Ok to finalize table otherwise you can also copy or change DDL script via DDL tab here.



Schema: SCOTT

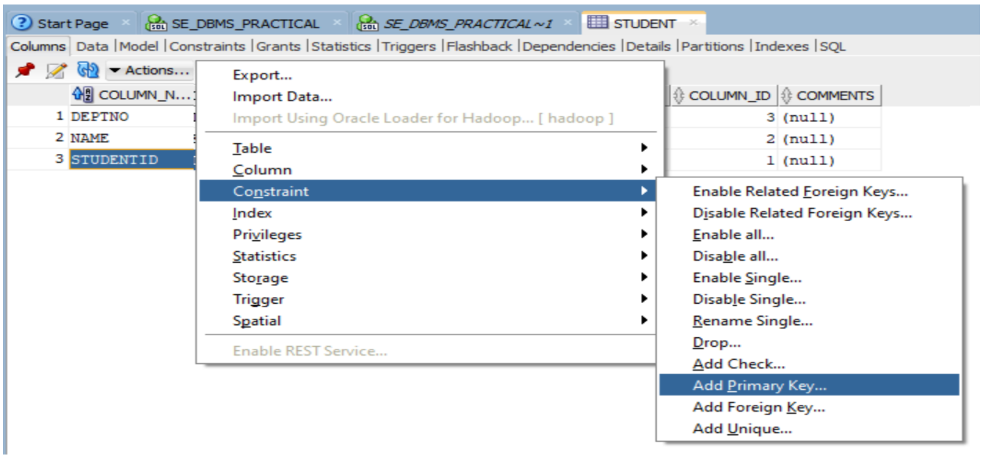
Name: STUDENT

Columns: name

PK	Name	Data Type	Size	Not Null	Default	Comment
	STUDENTID	NUMBER	4	☑		
	NAME	VARCHAR2	20	□		
	DEPTNO	NUMBER		□		

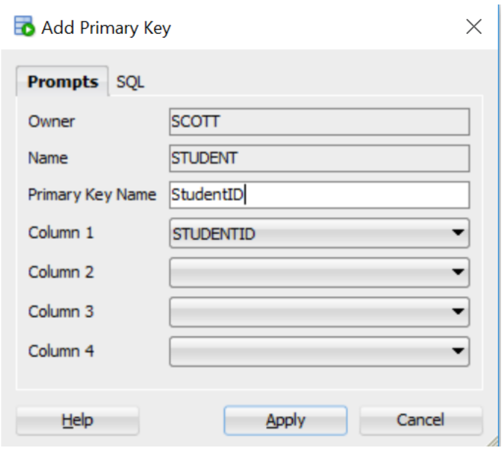
Figure 2.9

6. To add primary key/Foreign key in a table click on action then constraints and then add primary key/Foreign key.



	COLUMN_N...	COLUMN_ID	COMMENTS
1	DEPTNO	3	(null)
2	NAME	2	(null)
3	STUDENTID	1	(null)

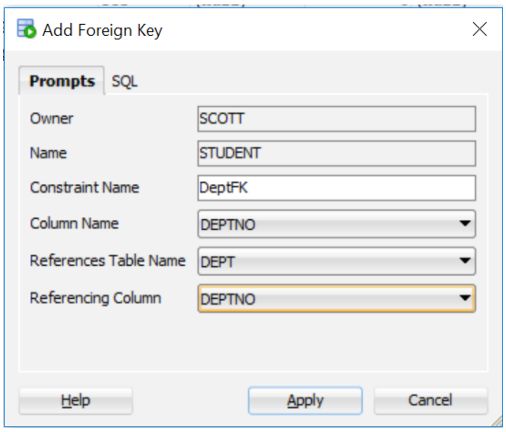
Figure 2.10



Owner: SCOTT
Name: STUDENT
Primary Key Name: StudentID
Column 1: STUDENTID
Column 2:
Column 3:
Column 4:

Figure 2.11

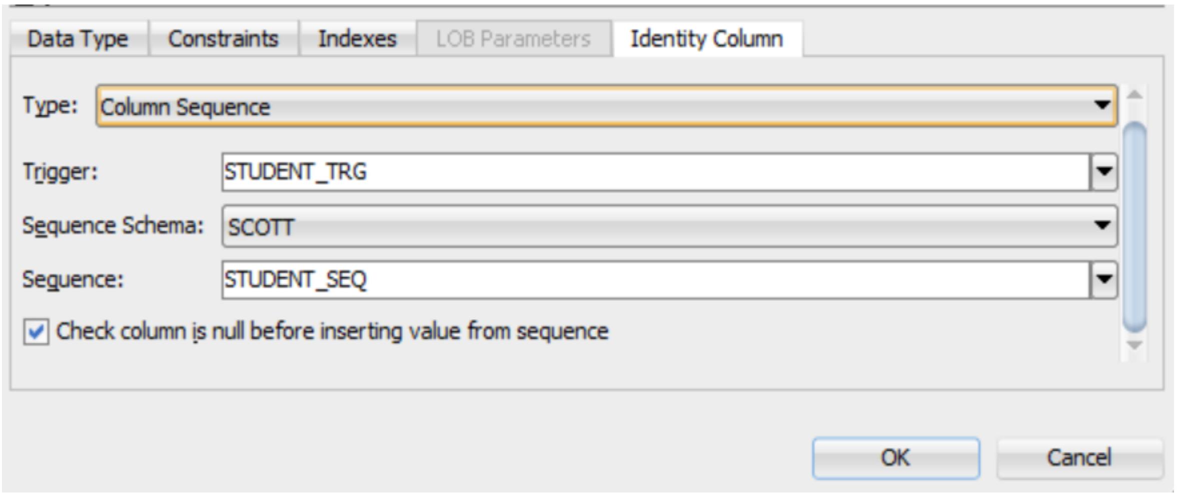
7. Similarly foreign key for this *student* table can be set with Deptno as a primary key for *department* table as:



Owner: SCOTT
Name: STUDENT
Constraint Name: DeptFK
Column Name: DEPTNO
References Table Name: DEPT
Referencing Column: DEPTNO

Figure 2.12

8. The table structure can be edited by clicking on Edit and then Add or remove or edit columns and DDL can also be created for the updated structure.
9. In order to make PK *auto increment*, the ID column property is set for that primary key by right clicking on the table and selecting "Edit".
10. In "Edit" Table window, select "columns", and then select PK column.
11. Go to ID Column tab and select Column Sequence as Type. This will create a trigger and a sequence, and associate the sequence to primary key. See the picture below for better understanding.



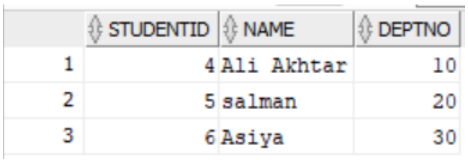
Type: Column Sequence
Trigger: STUDENT_TRG
Sequence Schema: SCOTT
Sequence: STUDENT_SEQ
☑ Check column is null before inserting value from sequence

Figure 2.13

Add Record in Table

1. Click on Data Tab and then click on plus sign for adding a new row.
2. As you have created sequence so no need to write StudentID (auto generated) while fill others column and then commit it . If you don't want your update to be reflected on database you can also Rollback .

```
INSERT INTO table (column_1, column_2)
    VALUES (value_1, value_2);
```



	STUDENTID	NAME	DEPTNO
1	4	Ali Akhtar	10
2	5	salman	20
3	6	Asiya	30

Figure 2.14

Delete Record from Table

1. Right click on a row you need to delete and click on *Delete Selected Row .*
2. Commit it to finalize your deletion.

Update Record in Table

1. Double click on a cell you need to update and as soon as you will change the record commit/Rollback option will appear.

Creating Backup using Import/Export Utilities

SQL Developer provides convenient wizards for exporting and importing metadata and data.

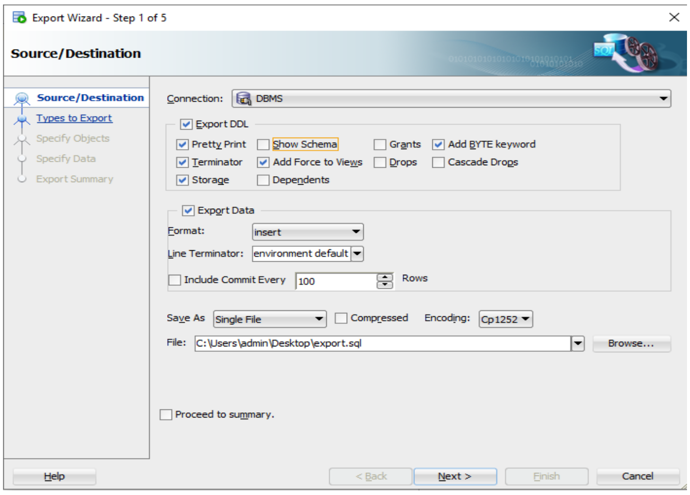
- To export metadata or data, or both, use the Export Wizard: click **Tools**, then **Database Export**.
- To import metadata or data, or both, use an appropriate method depending on how the material to be imported was created, or the format of the data to be imported. This method might be running a script file, or using the Data Import Wizard to import from a data file (such as a .csv file or a Microsoft Excel .xlsx file).

Exporting Metadata and Data for a Table

Assume that you want to export the **REGIONS** table, which is part of the **HR** sample schema, so that it can be created, along with its data, in another schema (either in the same Oracle database or another Oracle database).

To unload the **REGIONS** table:

1. In SQL Developer, click Tools, then Database Export as shown below.



Connection: DBMS

☑ Export DDL

☑ Pretty Print ☐ Show Schema ☐ Grants ☑ Add BYTE keyword

☑ Terminator ☑ Add Force to Views ☐ Drops ☐ Cascade Drops

☑ Storage ☐ Dependents

☑ Export Data

Format: insert

Line Terminator: environment default

☐ Include Commit Every 100 Rows

Save As Single File ☐ Compressed Encoding: Cp1252

File: C:\Users\admin\Desktop\export.sql Browse...

☐ Proceed to summary.

Figure 2.15 Export Wizard: Source/Destination

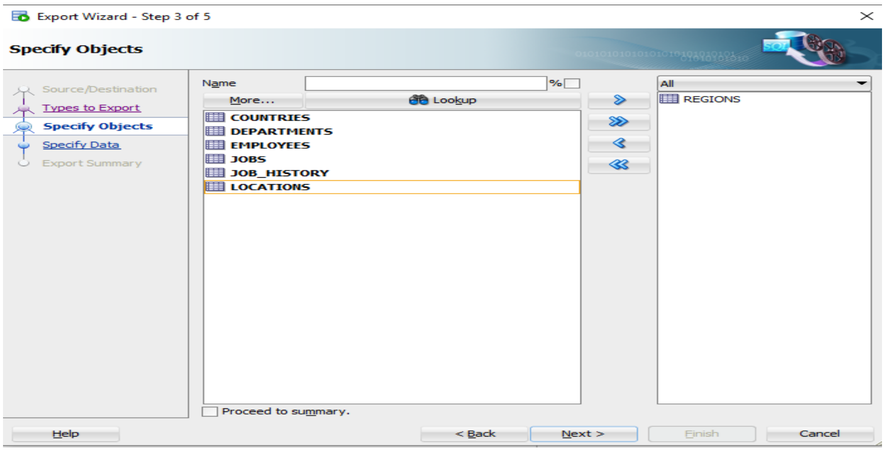
2. Accept the default values for the Source/Destination page options, except as follows:

Connection: Select **DBMS**.

Show Schema: ***Deselect*** (uncheck) this option, so that the **HR** schema name is not included in CREATE and INSERT statements in the .sql script file that will be created. (This enables you to re-create the table in a schema with any name, such as one not named HR.)

Save As location: Enter or browse to a desired folder on your local hard drive, and specify the file name for the script file. (In the figure, this file is **C:\users\admin\Desktop\export.sql**.) The script file containing CREATE and INSERT statements will be created in this location.

3. Click **Next**.
4. On the Types to Export page, deselect Toggle All, then select *only* **Tables** (because you only want to export a table).
5. Click Next.
6. On the Specify Objects page, click Lookup, then double-click the **REGIONS** table on the left to move it to the right-hand column as shown below.



COUNTRIES
DEPARTMENTS
EMPLOYEES
JOBS
JOB_HISTORY
LOCATIONS

REGIONS

☐ Proceed to summary.

Figure 2.16 Export Wizard: Specify Objects

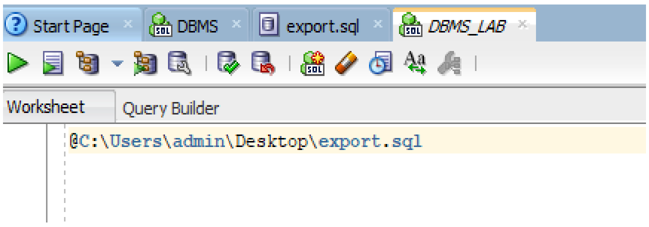
7. Click **Next**.
8. On the Specify Data page, accept the defaults and click **Next**.
   By default, all data from the specified table or tables is exported; however, if you want to limit the data to be exported, you can specify one or more "**WHERE** clauses" in the bottom part of this page.
9. On the Summary page, review the information; and if it is what you want, click **Finish**. This causes the export script to be created as **c:\users\admin\Desktop\export.sql**.

Importing Metadata and Data using a Script File

Assume that you wanted to re-create the **REGIONS** table that has been exported, but in a different schema. This other schema can be an existing one or one that you create.
For example, assume that you have created a user named **DBMS_LAB**. To re-create the **REGIONS** table in the schema of user **DBMS_LAB** by invoking the script in **c:\users\admin\Desktop\export.sql** follow these steps using SQL Developer:

1. Open the **DBMS_LAB** connection.
2. In the SQL Worksheet for the **DBMS_LAB** connection, type the following:
3. @c:\users\admin\Desktop\export.sql.
4. Click the Run Script icon.

The Script Output pane shows that the **REGIONS** table has been created and four rows have been inserted.



```
@C:\Users\admin\Desktop\export.sql
```

Figure 2.17

5. In the Connections navigator, expand the Tables node under the **DBMS_LAB** connection. You now see the **REGIONS** table.
6. Optionally, click the **REGIONS** table in the Connections navigator, and examine the information under the **Columns** and **Data** tabs in the main display area.

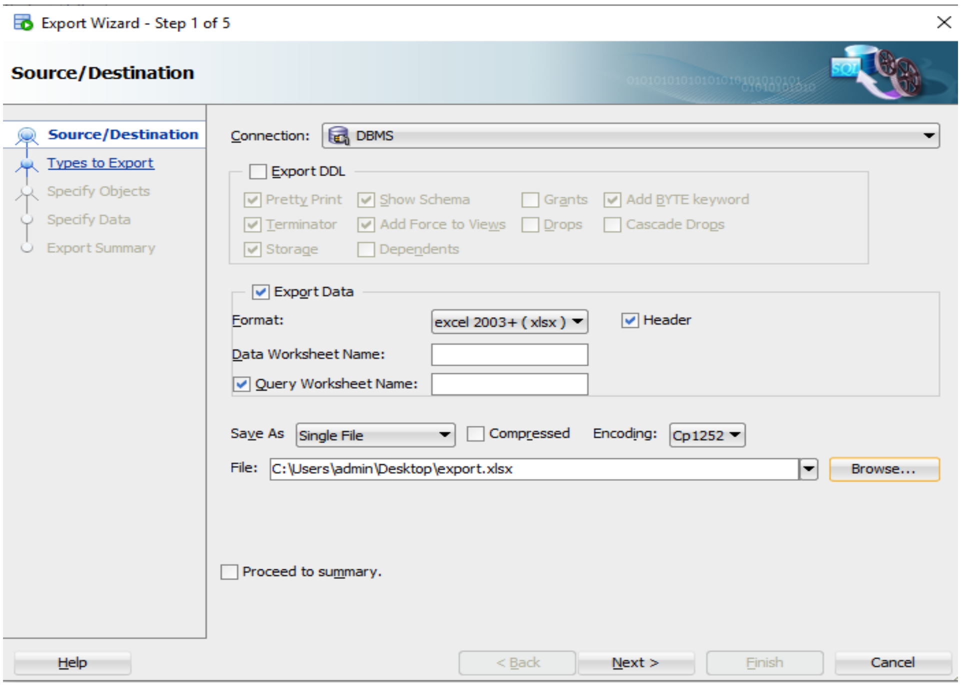
Exporting Data to a Microsoft Excel File

Assume that you want to export only the data from the **REGIONS** table, which is part of the **HR** sample schema, so that the data can be imported into a table with the same column definitions. This might be a **REGIONS** table in another schema (either in the same Oracle database or another Oracle database).

You use the same Database Export wizard, but export only the data, and not the DDL (Data Definition Language statements for creating database objects).

To export the data the **REGIONS** table:

1. In SQL Developer, click Tools, then Database Export as shown below.



Export Wizard - Step 1 of 5

Source/Destination

Source/Destination
Types to Export
Specify Objects
Specify Data
Export Summary

Connection: DBMS

☐ Export DDL

☑ Pretty Print ☑ Show Schema ☐ Grants ☑ Add BYTE keyword
☑ Terminator ☑ Add Force to Views ☐ Drops ☐ Cascade Drops
☑ Storage ☐ Dependents

☑ Export Data

Format: excel 2003+ (xlsx) ☑ Header

Data Worksheet Name:

☑ Query Worksheet Name:

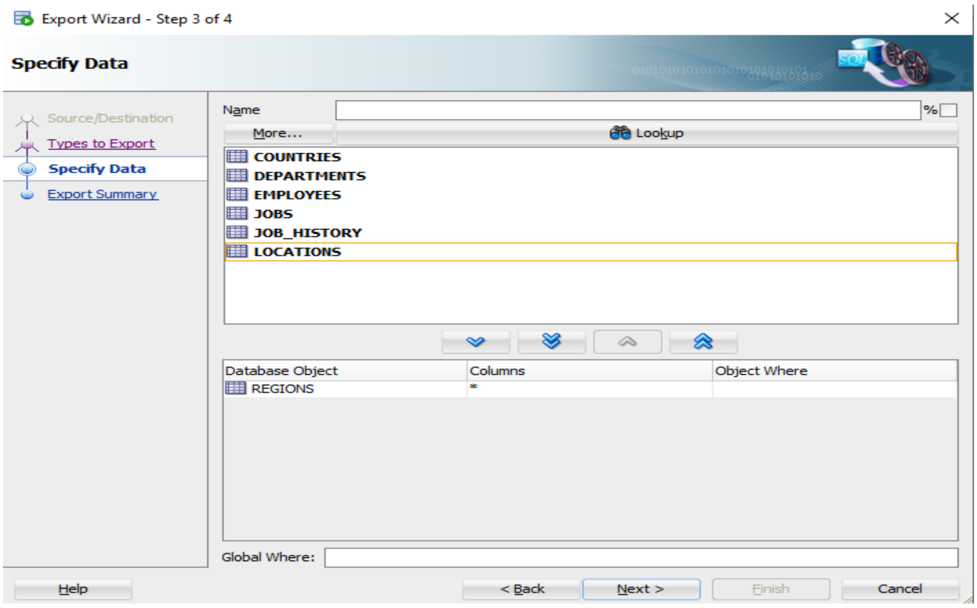
Save As Single File ☐ Compressed Encoding: Cp1252

File: C:\Users\admin\Desktop\export.xlsx Browse...

☐ Proceed to summary.

Figure 2.18 Export Wizard: Source/Destination Specifying Data Export Only

2. Accept the default values for the Source/Destination page options, except as follows:
   Connection: Select **DBMS_LAB**.
   Export DDL: ***Deselect*** (uncheck) this option. If a .sql script file is generated (which will not happen in this example), it will not contain any **CREATE** statements, but only **INSERT** statements.
   Format: Select xls to have the data saved to a Microsoft Excel .xlsx file.
   Save As location: Enter or browse to a desired folder on your local hard drive, and specify the file name for the .xlsx file. (In the figure, this file is **C:\Users\admin\Desktop\export.xlsx**.)
3. Click **Next**.
4. On the Types to Export page, deselect Toggle All, then select *only* **Tables** (because you only want to export data for a table).
5. Click Next.
6. On the Specify Objects page, click **Lookup**, then double-click the **REGIONS** table on the left to have it appear in a row in the bottom part of the page as shown below.



Export Wizard - Step 3 of 4

Specify Data

Source/Destination
Types to Export
Specify Data
Export Summary

Name %☐

COUNTRIES
DEPARTMENTS
EMPLOYEES
JOBS
JOB_HISTORY
LOCATIONS

Database Object	Columns	Object Where
REGIONS	*	

Global Where:

Figure 2.19 Export Wizard: Specify Objects for Exporting Data

By default, all data from the specified table or tables is exported; however, if you want to limit the data to be exported, you can specify one or more "**WHERE** clauses" in the bottom part of this page.

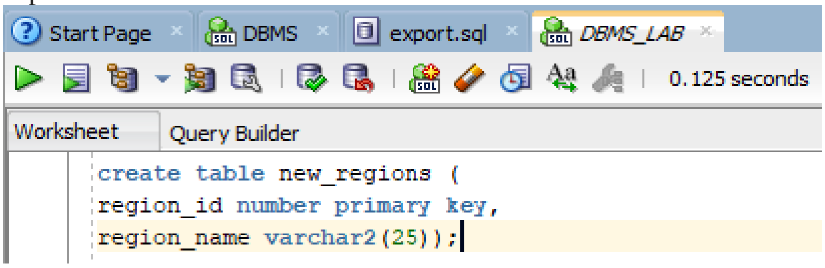
7. Click **Next**.
8. On the Summary page, review the information; and if it is what you want, click **Finish**. This causes the data in the **REGIONS** table to be exported to the file **C:\Users\admin\Desktop\export.xlsx**.

Importing Data from a Microsoft Excel File

Assume that you wanted to import the data that has been exported into a new table that has the same column definitions as the original (**REGIONS**) table.
For example, assume that you created a user named **DBMS_LAB**. This user wants to take the exported data, add one row in the Excel file, and import it into a new table that has the same column definitions as the **REGIONS** table.

To accomplish these goals, follow these steps:

1. Open the **DBMS_LAB** connection.
2. In the SQL Worksheet for the **DBMS_LAB** connection, type the following:
3. create table new_regions (region_id number primary key, region_name varchar2(25));
4. Click the Run Script icon.

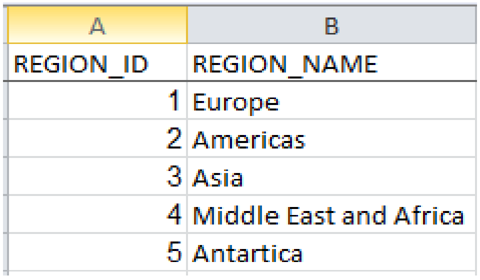


```
create table new_regions (
region_id number primary key,
region_name varchar2(25));
```

Figure 2.20

The Script Output pane shows that the **NEW_REGIONS** table has been created.

5. In the Connections navigator, expand the Tables node under the **DBMS_LAB** connection. You now see the **NEW_REGIONS** table.
   If you do not see the **NEW_REGIONS** table, disconnect from **DBMS_LAB** (right-click **DBMS_LAB** in the Connections navigator and select Disconnect) and connect again, and expand the Tables node.
6. Using Microsoft Excel, open the file containing the exported data (for example, **C:\Users\admin\Desktop**\export.xlsx), and optionally add one or more rows. Following figure shows the original file with one row added for the **Antarctica** region.

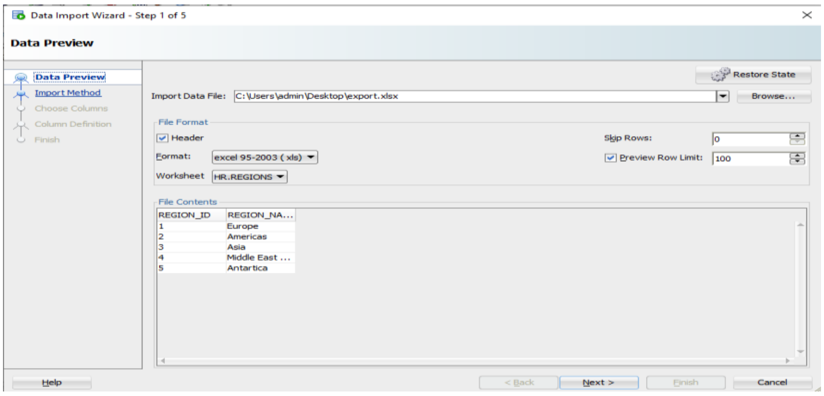


A	B
REGION_ID	REGION_NAME
1	Europe
2	Americas
3	Asia
4	Middle East and Africa
5	Antartica

Figure 2.21 Microsoft Excel File with Exported Data (Modified)

7. Save and close the Microsoft Excel .xlsx file.

8. In SQL Developer, in the Connections navigator display for **DBMS_LAB**, right-click the **NEW_REGIONS** table and select **Import Data**.
9. In the dialog box that is displayed, navigate to the **C:\Users\admin\Desktop** folder, select **export.xlsx**, and click **Open**.
10. In the Data Import Wizard, accept all the defaults; click **Next** on each page until Summary, and click **Finish** there.



Import Data File: C:\Users\admin\Desktop\export.xlsx

☑ Header

Format: excel 95-2003 (xls)

Worksheet HR.REGIONS

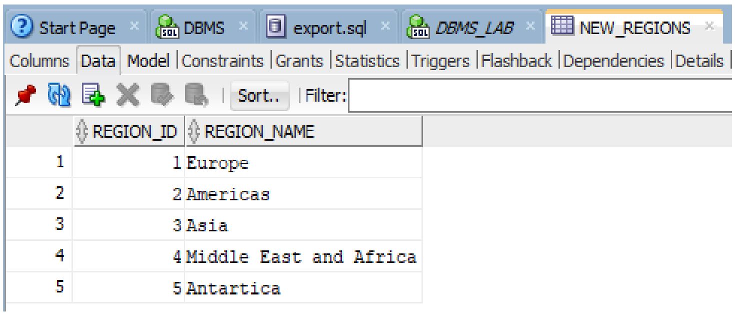
Skip Rows: 0

☑ Preview Row Limit: 100

REGION_ID	REGION_NA...
1	Europe
2	Americas
3	Asia
4	Middle East ...
5	Antartica

Figure 2.22

The data from the **.xlsx** file is loaded into the **NEW_REGIONS** table and is committed as shown below.



	REGION_ID	REGION_NAME
1	1	Europe
2	2	Americas
3	3	Asia
4	4	Middle East and Africa
5	5	Antartica

Figure 2.23

EXERCISES

1. Create the following tables in database. Attach the screenshots of created tables.

Books

Column Name	Type	Size	Other Information and Notes
book_id	VARCHAR2	20	Primary Key (Automatically checks Not Null; an index is also created on the primary key column. This is the Dewey code or other book identifier.)
Title	VARCHAR2	50	Not Null
author_last_name	VARCHAR2	30	Not Null
author_first_name	VARCHAR2	30	
Rating	NUMBER		(Librarian's personal rating of the book, from 1 (poor) to 10 (great))

Patrons

Column Name	Type	Size	Other Information and Notes
patron_id	NUMBER		Primary Key. (Unique patron ID number, with values to be created using a sequence that you will create)
last_name	VARCHAR2	30	Not Null
first_name	VARCHAR2	30	
street_address	VARCHAR2	30	
city_state_zip	VARCHAR2	30	

Transactions

Column Name	Type	Size	Other Information and Notes
transaction_id	NUMBER		Primary Key. (Unique transaction ID number, with values to be created using a trigger and sequence that will be created automatically)
patron_id	NUMBER		(Foreign key; must match a patron_id value in the PATRONS table)
book_id	VARCHAR2	20	(Foreign key; must match a book_id value in the BOOKS table)
transaction_date	DATE		(Date and time of the transaction)
transaction_type	NUMBER		(Numeric code indicating the type of transaction, such as 1 for checking out a book)

2. Insert data in above provided tables as given below. Attach the screenshots of tables containing inserted data.

Books

BOOK_ID	TITLE	AUTHOR_LAST_NAME	AUTHOR_FIRST_NAME	RATING
A1111	Silence on the Moon	Akhtar	Ali	10
A2222	Get Rich Really Fast	Asrar	Kashif	1
A3333	Finding Inner Peace	Ahmed	Hameeza	0
A4444	Great Mystery Stories	Hussain	Nadir	5
A5555	Software Wizardry	Iftikhar	Umer	10

Patrons

PATRON_ID	LAST_NAME	FIRST_NAME	STREET_ADDRESS	CITY_STATE_ZIP	LOCATION
100	Smith	Jane	123 Main Street	Mytown, MA 01234	
101	Chen	William	16 S. Maple Road	Mytown, MA 01234	
102	Fernandez	Maria	502 Harrison Blvd.	Sometown, NH 03078	
103	Murphy	Sam	57 Main Street	Mytown, MA 01234	

Transactions

TRANSACTION_ID	PATRON_ID	BOOK_ID	TRANSACTION_DATE	TRANSACTION_TYPE
1	100	A1111	24-FEB-19	1
2	100	A2222	25-FEB-19	2
3	101	A3333	08-FEB-19	3
4	101	A2222	20-FEB-19	1
5	102	A3333	20-FEB-19	1
6	103	A4444	04-FEB-19	2
7	100	A4444	04-FEB-19	1
8	102	A2222	24-FEB-19	2
9	102	A5555	14-FEB-19	1
10	101	A2222	09-FEB-19	1

3. Export the created tables **Books**, **Patrons**, **Transactions** using a script file. Attach the screenshots of generated script.

4. Create excel files for the tables (**Books**, **Patrons**, **Transactions)** using following data entries. Import data into created tables from the excel files. Attach the screenshots of tables containing inserted data.

Books

BOOK_ID	TITLE	AUTHOR_LAST_NAME	AUTHOR_FIRST_NAME	RATING
BOOK1	Database System Concepts	Silberschatz	Abraham	10
BOOK2	MongoDB: The Definite Guide	Chodorow	Kristina	1

Patrons

PATRON_ID	LAST_NAME	FIRST_NAME	STREET_ADDRESS	CITY_STATE_ZIP	LOCATION
200	Sam	William	123 Main Street	Mytown, MA 01234	
201	Chen	Fernandez	16 S. Maple Road	Mytown, MA 01234	

Transactions

TRANSACTION_ID	PATRON_ID	BOOK_ID	TRANSACTION_DATE	TRANSACTION_TYPE
1	200	BOOK1	24-FEB-19	1
2	201	BOOK2	25-FEB-19	2
3	200	BOOK1	08-FEB-19	3
4	201	BOOK2	20-FEB-19	1

Lab Session 03

*Apply Data Retrieval Operations of SQL and SQL*Plus*

INTRODUCTION

Oracle 7 is a relational database management system whereas *Oracle 8* and *Oracle 9i* are object relational database management systems. There are two products *Oracle 9i Application Server* and *Oracle 9i Database* that provide a complete and simple infrastructure for internet applications. The Oracle 9i application server (Oracle *9iAS*) runs all your applications. The Oracle 9i database stores all your data. In the lab sessions, **Oracle 11g** is used.

The table 3.1 below shows the contents of the EMP table or relation that stores data about employees presently working in an organization.

- The table has eight columns namely EMPNO, ENAME. JOB, MGR, HIREDATE, SAL, COMM, DEPTNO storing the different attributes of an employee.
- The table has fourteen rows each representing all data that is required for a particular employee. Each row in a table should be identified by a ***primary key***,
- A ***foreign key*** is a column or a set of columns that refers to a primary key or a unique key in the same table or another table. In EMP table, DEPTNO is the foreign key.
- A field can be found at the intersection of a row and a column. There can be only one value or no value in a cell. No value is called *null value*. In the EMP table, only *salesman* has a value in the COMM (Commission) field.

EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT		17-NOV-81	5000		10
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	0	30
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7788	SCOTT	ANALYST	7566	09-DEC-82	3000		20
7876	ADAMS	CLERK	7788	12-JAN-83	1100		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

Table 3.1

DEPT

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

Table 3.2

SALGRADE

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

Table 3.3

JOB_HISTORY

EMPNO	JOB	START_DATE	END_DATE
7698	ASSISTANT	04-MAR-80	30-APR-81
7654	RECEPTIONIST	13-JAN-80	09-SEP-80
7654	SALESMAN	10-SEP-80	20-SEP-81
7788	PROGRAMMER	13-FEB-80	03-DEC-82
7876	TYPIST	12-APR-80	13-NOV-81
7876	OPERATOR	15-NOV-81	11-JAN-83
7839	ANALYST	13-JUN-78	10-OCT-81

Table 3.4

Guidelines for Primary and Foreign Keys

- No duplicate values are allowed in a primary key.
- Primary keys generally cannot be changed.
- Foreign keys are based on data values and are purely logical, not physical pointers.
- A foreign key value must match an existing primary key value or unique key value, or else be null.

SQL (Structured Query Language)

Most commercial database management systems support a query language, SQL, which is the most influential commercially marketed product. SQL is a nonprocedural language: you specify *what* information you require, rather than *how* to get it. In other words, SQL does not require you to specify the access methods to the data and provides abstraction. As a result, it doesn't provide traditional programming structures.

Data-definition language (DDL): The SQL DDL provides commands for defining relation schemas, deleting relations, creating indices, and modifying relation schemas.

Interactive data-manipulation language (DML): It includes commands to insert tuples into, delete tuples from, and to modify tuples in the database.

Basic Data Retrieval

The basic structure of an SQL query consists of three clauses: SELECT, FROM and WHERE.

```
SELECT  * | {[DISTINCT] column | expression [alias], …}
FROM  table
[WHERE  condition];
```

Note: The keywords and clauses enclosed in square brackets are *optional*.

Examples

- Selecting all columns from a table

```
SELECT  *
FROM DEPT;
```

- To select names of all jobs in a department, use

```
SELECT DISTINCT JOB
FROM EMP;
```

  Note: The DISTINCT clause before a column name suppresses duplicate values

- To select all employees whose salary is greater than 2200.

```
SELECT *
FROM EMP
WHERE SAL > 2200;
```

- To display the name and department number of employees who were hired before 12th May, 1981.

```
SELECT ENAME, DEPTNO
FROM EMP
WHERE HIREDATE < '12-MAY-1981';
```

- To display the name and job of employees using literal character strings and concatenation operators.

```
SELECT ENAME || ' is a ' || JOB
AS "Employee Details"
FROM EMP;
```

- Retrieving data from multiple tables: To select employee name, job and department name,

```
SELECT E.ENAME, E.JOB, D.DNAME
FROM EMP E, DEPT D
WHERE E.DEPTNO = D.DEPTNO;
```

SELECT Statement

To extract data from the database, the SQL SELECT statement is used.

Capabilities of SELECT statement

Following are the various operations that can be performed using SELECT:-

1. **Selection**: The selection capability can be used to choose rows in a table depending on the criteria to selectively restrict the rows.

Examples

- Selecting all employees whose salary is between 3500 and 5000 and who were hired after 31st July, 1981.

```
SELECT  *
FROM  EMP
WHERE (SAL BETWEEN 3500 AND 5000) AND HIREDATE > TO_DATE('31-JUL-1981', 'DD-MON-YYYY');
```

- Selecting all employees whose job is either clerk or analyst and were hired between 23rd July, 1981 and 14th May, 1982.

```
SELECT  *  FROM EMP
WHERE (JOB = 'CLERK' OR JOB = 'ANALYST') AND HIREDATE BETWEEN TO_DATE('23-JUL-1981', 'DD-MON-YYYY') AND TO_DATE('14-MAY-1982', 'DD-MON-YYYY');
```

2. **Projection**: It refers to choosing the columns in a table that are to be returned by a query. We can choose as few or as many columns of the table as we require.

Examples

- Selecting employee number, name and their job

```
SELECT EMPNO, ENAME, JOB
```

FROM EMP;

- Selecting employee number, name and their salary who do not earn commission
  SELECT EMPNO, ENAME, SAL
  FROM EMP
  WHERE COMM IS NULL;

3. **Join**: To bring together data that is stored in different tables by creating a link through a column that both the tables share.

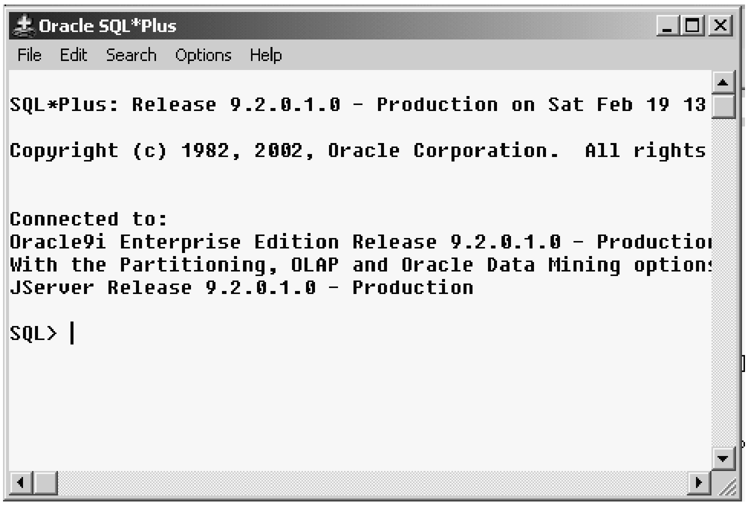
Example

To retrieve the employee name, their job and department name, we need to extract data from two tables, EMP and DEPT. This type of join is called *equijoin*-that is, values in the DEPTNO column on both tables must be equal. Equijoin is also called *simple join* or *inner join*. The output is shown in figure 2.2.

- SELECT E.ENAME, E.JOB, D.DNAME
  FROM EMP E, DEPT D
  WHERE E.DEPTNO = D.DEPTNO;

SQL*Plus

SQL*Plus is an Oracle tool that recognizes and submits SQL and PL/SQL statements to the server for execution, and contains its own command language. It accepts *ad hoc* query statements through the editor as well as SQL input from files. The SQL*Plus environment is shown below in figure 3.1.



```
SQL*Plus: Release 9.2.0.1.0 - Production on Sat Feb 19 13

Copyright (c) 1982, 2002, Oracle Corporation.  All rights


Connected to:
Oracle9i Enterprise Edition Release 9.2.0.1.0 - Productio
With the Partitioning, OLAP and Oracle Data Mining option
JServer Release 9.2.0.1.0 - Production

SQL> |
```

Figure 3.1

In contrast to SQL statements, SQL*Plus commands do not allow the manipulation of values in the database and are not stored in the SQL buffer. Commands are entered one line at a time and have a dash (-) as a continuation character if the command is longer than one line. It uses commands to format data retrieved through SQL statements. SQL*Plus commands can be abbreviated whereas SQL statements cannot. A few commands of SQL*Plus are as follows:-

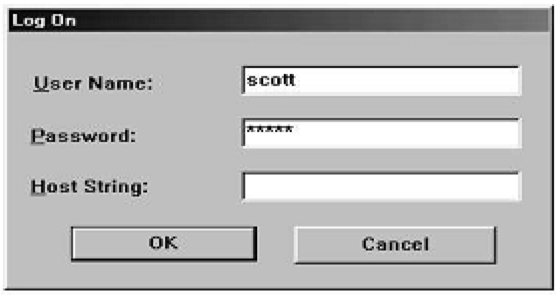
DESC[RIBE]: To display the structure of a table e.g. SQL> DESC EMP
SAV[E] *filename*[.ext]: Saves current contents of SQL buffer to a file e.g. SQL>SAVE D:\DATA\FINDSAL
GET *filename*[.ext]: Writes the contents of a previously saved file to the SQL buffer. The default extension for the file is .sql. e.g. SQL> GET D:\DATA\FINDSAL
@: Runs a previously saved command file e.g. SQL>@ *filename*
SPO[OL]: Stores query results in a file e.g. SQL>SPOOL *filename.ext*

SPOOL OFF: Closes the spool file
SPOOL OUT: Closes the spool file and sends the file results to the system printer
ED[IT]: Invokes the editor and saves the buffer contents to a file named afiedt.buf
ED[IT] [*filename*[.ext]**]:** Invokes editor to edit contents of a saved file
EXIT: Leaves SQL*Plus

Logging in to SQL*Plus

To log in through a windows environment:

- Click Start/Programs/Oracle for Windows/SQL*Plus
- Fill in username, password, and database in the window shown in figure 3.2.



User Name: scott
Password: *****
Host String:

Figure 3.2

Operators and their meanings

Comparison Operators

Comparison operators are used in conditions that compare one expression to another. They are used in the WHERE or HAVING clause of the SELECT statement.

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to

Table 3.5

Besides basic comparison operators (>, <, >=, <=, =, <>), Oracle SQL also supports following comparison operators.

Operator	Meaning
BETWEEN … AND …	Between two values (inclusive)
IN (list)	Match any of a list of values
LIKE	Match a character pattern
IS NULL	Is a null value

Table 3.6

Examples

- To display record of employees who are not managers.
  SELECT * FROM EMP
  WHERE JOB <> ‘MANAGER’;
- To display the employee number, name, salary and the manager’s employee number of all the employees whose manager’s employee number is 7902, 7566, or 7788.
  SELECT EMPNO, ENAME, SAL, MGR
  FROM EMP

```
WHERE MGR IN (7902, 7566, 7788);
```

- To display the names of all employees with names starting with **S,**

```
SELECT ENAME
FROM EMP
WHERE ENAME LIKE ‘S%’;
```

Note: Above query performs wildcard searches using LIKE operator. Here % symbol represents any sequence of zero or more characters.

- To display the names of all employees with second character of name as **A,**

```
SELECT ENAME
FROM EMP
WHERE ENAME LIKE ‘_A%’;
```

Note: Here _ character represents any single character

Logical Operators

A logical operator combines the result of two component conditions to produce a single result based on them or to invert the result of a single condition. Three logical operators are available in SQL as shown below.

Operator	Meaning
AND	Returns TRUE if both component conditions are TRUE
OR	Returns TRUE if either component condition is TRUE
NOT	Returns TRUE if the following condition is FALSE

Table 3.7

Examples

- To display record of all clerks who earn more than 1100

```
SELECT empno, ename, job, sal
FROM emp
WHERE sal >= 1100
AND job = ‘CLERK’;
```

- To display record of all employees who are either clerks or earn more than 1100.

```
SELECT empno, ename, job, sal
FROM emp
WHERE sal >= 1100
OR job = ‘CLERK’;
```

- To display name and job title of all the employees whose are not CLERK, MANAGER, or ANALYST.

```
SELECT ename, job
FROM emp
WHERE job NOT IN (‘CLERK’, ‘MANAGER’, ‘ANALYST’);
```

Rules of Precedence

Order Evaluated	Operator
1	All comparison operators
2	NOT
3	AND
4	OR

Table 3.8

For example, consider the following statement:

-
```
SELECT ename, job, sal FROM emp4
WHERE job = ‘SALESMAN’
OR job = ‘PRESIDENT’
```

AND sal > 1500;

In the above example, there are two conditions:

- The first condition is that job is SALESMAN.
- The second condition is that job is CLERK and salary is greater than 1000.

Therefore the SELECT statement reads as follows:-
Select the row if an employee is a SALESMAN or an employee is a CLERK and earns more than 1000.

In order to force the OR operator to be evaluated before AND, use parentheses as follows:

```
SELECT ename, job, sal FROM emp
WHERE (job = ‘SALESMAN’
OR job = ‘PRESIDENT’)
AND sal > 1500;
```

Ordering Data

The order of rows returned in a query result is undefined. The ORDER BY clause can be used to sort the rows. This clause comes last in the SELECT statement. ASC at the end of the ORDER BY clause specifies ascending order whereas DESC specifies descending order. ASC is the default order.

Examples

- To select data in the increasing order of hiredate,
  ```
  SELECT ENAME, JOB, DEPTNO, HIREDATE
  FROM EMP
  ORDER BY HIREDATE;
  ```
- To select data in the decreasing order of hiredate,
  ```
  SELECT ENAME, JOB, DEPTNO, HIREDATE
  FROM EMP
  ORDER BY HIREDATE DESC;
  ```
- To sort by column alias,
  ```
  SELECT EMPNO, ENAME, SAL*12 ANNSAL
  FROM EMP
  ORDER BY ANNSAL
  ```
- To sort by multiple columns,
  ```
  SELECT ENAME, DEPTNO, SAL
  FROM EMP
  ORDER BY DEPTNO, SAL DESC;
  ```
  Note: The DESC applies only to SAL column. The DEPTNO appears in ascending order.
- To select list of names and jobs of all employees hired in 1987 in the alphabetical order of name
  ```
  SELECT UPPER(ENAME) “EMP NAME”, JOB
  FROM EMP
  WHERE TO_CHAR(HIREDATE, ‘YYYY’) = 1987
  ORDER BY ENAME;
  ```
- To print employee number, name, job, annual salary of all managers and clerks whose monthly salary is between 3000 and 5500 in descending order of annual salary.
  ```
  SELECT EMPNO, ENAME, JOB, 12*SAL + NVL(COMM, 0)  ANNUAL_SALARY
  FROM EMP
  WHERE JOB = ‘MANAGER’ OR JOB = ‘CLERK’
  AND SAL BETWEEN 3000 AND 5500
  ORDER BY ANNUAL_SALARY DESC;
  ```

EXERCISES

1. Write a query to display the employee number, name and salary of all managers.

2. Write a query to display the name and department number of all employees who were hired after 1982.

3. Write a query to display the name of all managers in department 20.

4. Display the one month salary of employees written as:
   KING: 1 Month salary = 5000
   Hint: use literal character string

5. To display the name and department number of employee with number 7566.

6. To display the name and department number of all employees in departments 10 and 30 in alphabetical order by name.

7. To display the name, department number and hire date of all employees who were hired in *1982*.

8. To display the name of all employees who have two consecutive Ls in their name and are in department 30 or their manager is 7782

9. To display the name of all clerks of department 10 and 20 hired before 1983.

10. Display the name and salary for all employees whose salary is not in range of $1500 and $2850.

11. Display the name, salary and commission for all employees whose commission amount in greater than their salary increased by 10%.

Lab Session 04

Apply Data retrieval operations in SQL using join operations

INTRODUCTION

In previous lab session, we learned different ways to retrieve data from a single table. However, we frequently need data from more than one table. For example, suppose we need a report that displays employee id, name, job and department name. The first three attributes are present in EMP table where as the last one is in DEPT table (see previous lab session).

Cartesian Product

A Cartesian Product results when all rows in the first table are joined to all rows in the second table. A Cartesian product is formed under following conditions:

i. When a join condition is omitted
ii. When a join condition is invalid

Consider the following example:

```
SELECT *
FROM EMP, DEPT;
```

In the above example, if EMP table has 14 rows and DEPT table has 4 rows, then their Cartesian product would generate 14 x 4 = 56 rows.
In fact, the ISO standard provides a special format of the SELECT statement for the Cartesian product:-

```
SELECT *
FROM EMP CROSS JOIN DEPT;
```

A Cartesian product tends to generate a large number of rows and its result is rarely useful. It is always necessary to include a valid join condition in a WHERE clause. Hence a join is always a subset of a Cartesian product.

Types of Joins

i. **Inner-Join/Equi-Join:** If the join contains an equality condition, it is called equi-join.

Examples

- To retrieve the employee name, their job and department name, we need to extract data from two tables, EMP and DEPT. This type of join is called *equijoin*-that is, values in the DEPTNO column on both tables must be equal. Equijoin is also called *simple join* or *inner join*.

```
SELECT E.ENAME, E.JOB, D.DNAME
FROM EMP E, DEPT D
WHERE E.DEPTNO = D.DEPTNO;
```

The SQL-1999 standard provides the following alternative ways to specify this join:

```
SELECT ENAME, JOB, DNAME
FROM EMP NATURAL JOIN DEPT;
```

ii. **Outer-Join:** A join between two tables that returns the results of the inner join as well as unmatched rows in the left or right tables is a left or right outer join respectively. A full outer join is a join between two tables that returns the results of a left and right join.

Left Outer Join

SELECT E.ENAME, D.DEPTNO, D.DNAME
FROM EMP E, DEPT D
WHERE E.DEPTNO = D.DEPTNO(+);

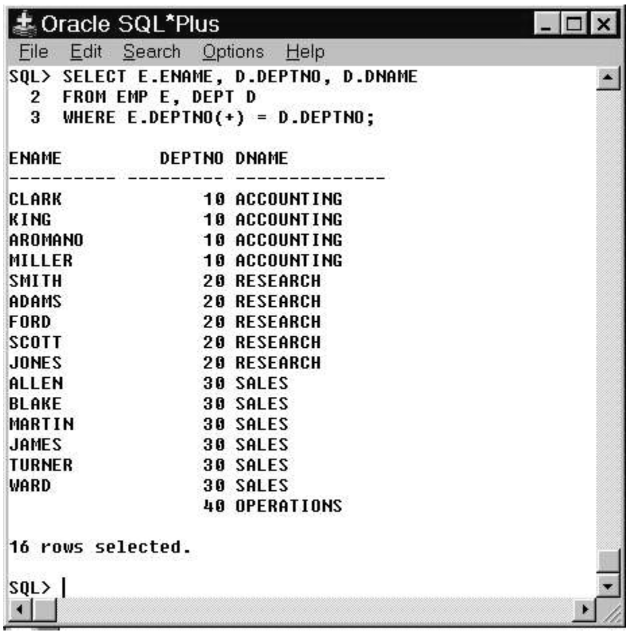
NOTE: The outer join operator appears on only that side that has information missing.

The SQL-1999 standard provides the following alternative way to specify this join:-
SELECT E.ENAME, D.DEPTNO, D.DNAME
FROM EMP E LEFT OUTER JOIN DEPT D
ON (E.DEPTNO = D.DEPTNO);

Right Outer Join

SELECT E.ENAME, D.DEPTNO, D.DNAME
FROM EMP E, DEPT D
WHERE E.DEPTNO(+) = D.DEPTNO;
The SQL-1999 standard provides the following alternative way to specify this join:-
SELECT E.ENAME, D.DEPTNO, D.DNAME
FROM EMP E RIGHT OUTER JOIN DEPT D
ON (E.DEPTNO = D.DEPTNO);

NOTE: In the equi-join condition of EMP and DEPT tables, department OPERATIONS does not appear because no one works in that department. In the outer join condition, the OPERATIONS department also appears.
The output is shown in figure 4.1.



```
SQL> SELECT E.ENAME, D.DEPTNO, D.DNAME
  2  FROM EMP E, DEPT D
  3  WHERE E.DEPTNO(+) = D.DEPTNO;

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

16 rows selected.

SQL> |
```

Figure 4.1: Joining tables using right outer-join

Full Outer Join

The SQL-1999 standard provides the following way to specify this join:-
SELECT E.ENAME, D.DEPTNO, D.DNAME
FROM EMP E FULL OUTER JOIN DEPT D
ON (E.DEPTNO = D.DEPTNO);

iii. Non-Equijoin: If the join contains inequality condition, it is called non-equijoin. E.g. to retrieve employee name, salary and their grades using *non-equijoins*, we need to extract data from two tables, EMP and SALGRADE.

- SELECT E.ENAME, E.SAL, S.GRADE
  FROM EMP E, SALGRADE S
  WHERE E.SAL
  BETWEEN S.LOSAL AND S.HISAL;

iv. **Self Join:** To find the name of each employee's manager, we need to join the EMP table to itself, or perform a *self join*.

- SELECT WORKER.ENAME || ' works for ' || MANAGER.ENAME
  FROM EMP WORKER, EMP MANAGER
  WHERE WORKER.MGR = MANAGER.EMPNO;

EXERCISES

i. To display the employee name, department name, and location of all employees who earn a commission.

ii. To display all the employee's name (including KING who has no manager) and their manager name.

iii. To display the name of all employees whose manager is ***KING***.

iv. Create a unique listing of all jobs that in department 30. Include the location of department 30 in the Output.

v. Write a query to display the name, job, department number and department name for all employees who work in DALLAS.

vi. Display the employee name and employee number along with their manager's name Manager Number. Label the columns Employee, Emp#, Manager, and Manager#, respectively.

Lab Session 05

Apply single-row and multiple-row functions in SQL

SQL Functions

Functions are a very powerful feature of SQL and can be used to do the following tasks:

- Perform calculations on data
- Modify individual data items
- Manipulate output for groups of rows
- Format dates and numbers for display
- Convert column data types

SQL functions may accept arguments and always return a value as illustrated in figure 5.1.

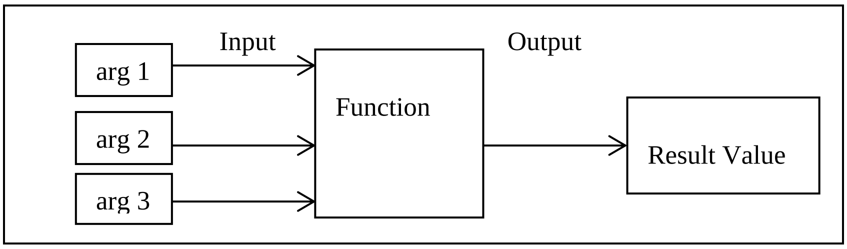


Figure 5.1: Functions accept arguments and return a value

Types of SQL Functions

There are two distinct types of functions:

- Single-row
- Multiple-row

Single-Row Functions: These functions operate on single rows only and return one result per row. There are different types of single-row functions. This session covers the following ones:

- Character
- Number
- Date
- Conversion

Multiple-Row Functions: These functions manipulate groups of rows to give one result per group of rows.

Single-Row Functions
Character Functions: Single-row character functions accept character data as input and can return both character and number values. Character functions can be divided into the following:

- Case conversion functions
- Character manipulation functions

Case Conversion Functions: Convert case for character strings.

Function	Result
LOWER(‘SQL Course’)	sql course
UPPER(‘SQL Course’)	SQL COURSE
INITCAP(‘SQL Course’)	Sql Course

Table 5.1

Examples

- To print an employee name (first letter capital) and job title (lower case)
  SELECT ‘The job title for ‘ || INITCAP(ename) || ‘ is ‘ || LOWER(job) AS “EMPLOYEE DETAILS” FROM emp;
- To display the employee number, name (in upper case) and department number for employee *Blake*.
  SELECT empno, UPPER(ename), deptno
  FROM emp
  WHERE LOWER(ename) = ‘blake’;
  Note: Since the actual case of the letters in the employee name column may not be known, so it is necessary for comparison to convert the name to either uppercase or lowercase.

Character manipulation functions: Manipulate character strings

Function	Result
CONCAT(‘Good’, ‘String’)	GoodString
SUBSTR(‘String’, 2, 4)	trin
LENGTH(‘String’)	6
INSTR(‘String’, ‘r’)	3
LPAD(sal, 10, ‘*’)	******5000

Table 5.2

Example

- To display employee name and job joined together, length of employee name, and the numeric position of letter A in the employee name, for all employees who are in *sales*.
  SELECT empno, CONCAT(ename, job), LENGTH(ename), INSTR(ename, ‘A’)
  FROM emp
  WHERE SUBSTR(job, 1, 5) = ‘SALES’;

Number Functions: Number functions accept numeric input and return numeric values.

ROUND(column|expression, n) Rounds the column, expression or value to **n** decimal places or if **n** is omitted, no decimal places (If n is negative, numbers to left of decimal point are rounded).

TRUNC(column|expression, n) Truncates the column, expression or value to *n* decimal places or if *n* is omitted, no decimal places (If *n* is negative, numbers to left of decimal point are truncated).

MOD(m, n) Returns the remainder of *m* divided by *n*.

Function	Result
ROUND(45.927, 2)	45.93
ROUND(45.927)	46
ROUND(45.927, -1)	50
TRUNC(45.927, 2)	45.92
TRUNC(45.927)	45
MOD(20, 3)	2

Table 5.3

Example

- SELECT ROUND(45.923, 2), ROUND(45.923, 0), ROUND(45.923, -1)
  FROM DUAL;
  The DUAL is a dummy table with one column and one row.
- SELECT TRUNC(45.923, 2), TRUNC(45.923), TRUNC(45.923, -1)
  FROM DUAL;
- To calculate the remainder of the ratio of salary to commission for all employees whose job title is salesman.
  SELECT ename, sal, comm., MOD(sal, comm.)
  FROM emp
  WHERE UPPER(job) = 'SALESMAN';

Date Functions: SYSDATE is a date function that returns the current date and time. The current date can be displayed by selecting SYSDATE from a table. It is customary to select SYSDATE from a dummy table called DUAL.
The DUAL table is owned by the user SYS and can be accessed by all users. It contains one column, DUMMY, and one row with the value X. It is useful for returning a value once only – for instance, the value of a constant, pseudocolumn, or expression that is not derived from a table with user data.

For example, to display the current date using the DUAL table as
SELECT SYSDATE
FROM DUAL;

Arithmetic with Dates: We can add or subtract a number to or from a date for a resultant date value. For example, to display the name and the number of weeks employed for all employees in department 10.
SELECT ename, (SYSDATE – HIREDATE) / 7 "Number of Weeks"
FROM emp
WHERE deptno = 10;
Date functions operate on Oracle dates. All date functions return a value of DATE datatype except MONTHS_BETWEEN, which returns a numeric value.

Function	Result	Description
MONTHS_BETWEEN('01-SEP-95', '11-JAN-94')	19.6774194	Number of months between two dates
ADD_MONTHS('11-JAN-94', 6)	'11-JUL-94'	Add calendar months to dates
NEXT_DAY('01-SEP-95', 'FRIDAY')	'08-SEP-95'	Next day of the date specified
LAST_DAY('01-SEP-95')	'30-SEP-95'	Last day of the month
ROUND(TO_DATE('25-JUL-95', 'DD-MON-YY'), 'MONTH')	01-AUG-95	Round date
ROUND(TO_DATE('25-JUL-95', 'DD-MON-YY'), 'YEAR')	01-JAN-96	Round date
TRUNC(TO_DATE('25-JUL-95', 'DD-MON-YY'), 'MONTH')	01-JUL-95	Truncate date
TRUNC(TO_DATE('25-JUL-95', 'DD-MON-YY'), 'YEAR')	01-JAN-95	Truncate date

Table 5.4

Examples

- For all employees employed for fewer than 200 months, display the employee number, hiredate, number of months employed, six-month review date, first Friday after hiredate and last day of the month hired.
  SELECT empno, hiredate, MONTHS_BETWEEN(SYSDATE, hiredate) TENURE,

ADD_MONTHS(hiredate, 6) REVIEW, NEXT_DAY(hiredate, ‘FRIDAY’), LAST_DAY(hiredate) FROM emp
WHERE MONTHS_BETWEEN(SYSDATE, hiredate) < 200;

- Comparing the hire dates for all employees who started in 1982, display the employee number, hiredate, and month started using the ROUND and TRUNC functions.
  SELECT empno, hiredate, ROUND(hiredate, ‘MONTH’),TRUNC(hiredate, ‘MONTH’) FROM emp WHERE hiredate like ‘%82’;

Conversion Functions: SQL provides three functions to convert a value from one data type to another.
TO_CHAR
TO_NUMBER
TO_DATE

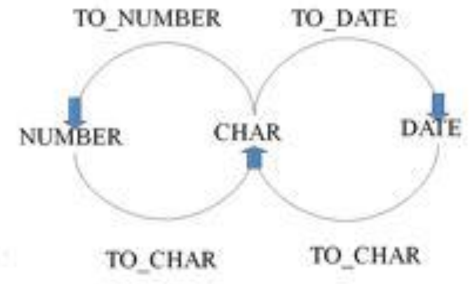


Figure 5.2

TO_CHAR function with Dates

- To display the employee number, the month number and year of hiring
  SELECT empno, TO_CHAR(hiredate, ‘MM/YY’) Month_Hired
  FROM emp
  WHERE ename = ‘BLAKE’;
  The second argument of TO_CHAR is called *format model*, is in single quotation marks and is case sensitive.
- To display the employee name and hiredate for all employees. The hiredate appears as 17 November, 1981.
  SELECT ename, TO_CHAR(hiredate, 'fmDD Month YYYY') HIREDATE
  FROM emp;
  The *fm* element is used to remove padded blanks or suppress leading zeros.
- To print the employee name and time of joining in format HH:MI:SS (Assuming that hiredate column were used for storing joining time)
  SELECT ename, TO_CHAR(hiredate, ‘HH:MI:SS’) HIREDATE
  FROM emp;

TO_CHAR function with Numbers

It is used to display a number value as a character string. This technique is especially useful for concatenating a numeric value to a character string.

i. To display the salary of employee SCOTT with $ sign preceded
   SELECT TO_CHAR(sal, ‘$99,999’) SALARY
   FROM emp
   WHERE ename = ‘SCOTT’;

The oracle server displays a string of pound signs (#) in place of a whole number whose digits exceed the number of digits provided in the format model. The oracle server rounds the stored decimal value to the number of decimal places provided in the format model.

TO_NUMBER function: Convert a character string to a number format using the TO_NUMBER function.

TO_DATE function: Converts a character string to a date format using the TO_DATE function.

- To display the names and hire dates of all the employees who joined on February 22, 1981

```
SELECT ename, hiredate
FROM emp
WHERE hiredate = TO_DATE('February 22, 1981', 'Month dd, YYYY');
```

DECODE Function: Facilitates conditional inquiries by doing the work of a CASE or IF-THEN-ELSE statement. It decodes an expression in a way similar to the IF-THEN-ELSE logic used in various languages. The DECODE function decodes *expression* after comparing it to each *search* value. If the expression is the same as *search, result* is returned. If the default value is omitted, a null value is returned where a search value does not match any of the result values.
DECODE (*col/expression, search1, result1* [,*search2, result2*, ….,] [, *default*]);

<u>Examples</u>

- To print job, salary and revised salary depending on the job.

```
SELECT job, sal, DECODE (job, 'ANALYST', SAL*1.1, 'CLERK', SAL*1.15, 'MANAGER', SAL*1.20, SAL) REVISED_SALARY FROM emp;
```

The same statement can be written as an IF-THEN-ELSE statement:

```
IF job = 'ANALYST' THEN       sal = sal * 1.1
IF job = 'CLERK' THEN   sal = sal * 1.15
IF job = 'MANAGER' THEN       sal = sal * 1.20
ELSE sal = sal
```

- To display the applicable tax rate for each employee in department 30

```
SELECT ename, sal, DECODE(TRUNC(sal/1000, 0), 0, 0.00, 1, 0.09, 2, 0.20, 3, 0.30, 4, 0.40, 5, 0.42, 6, 0.44, 0.45) TAX_RATE
FROM emp
WHERE deptno = 30;
```

Nesting Functions

Single-row functions can be nested to any level. For example, the following query displays the head of a company who has no manager.

- SELECT ENAME, NVL(TO_CHAR(MGR), 'No Manager')

```
FROM EMP
WHERE MGR IS NULL;
```

Multiple_Row Functions: Group functions operate on sets of rows to give one result per group. The following table identifies the options that can be used in the syntax:-

Following is the syntax of using group functions:-

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

Applying multiple-row functions to all rows in a table

<u>Examples</u>

- To show the average salary, minimum salary, maximum salary and count of employees in the organization

```
SELECT AVG (SAL), MIN(SAL), MAX(SAL), COUNT(*)
FROM EMP;
```

- To show the minimum and maximum hiredate for employees

  ```
  SELECT MIN (hiredate), MAX(hiredate)
  FROM emp;
  ```

- To return the number of rows in a table

  ```
  SELECT COUNT(*)
  FROM emp
  WHERE deptno = 30;
  ```

- To return the number of non null rows in a table

  ```
  SELECT COUNT(comm)
  FROM emp
  WHERE deptno = 30;
  ```

- The group function like AVG do not include null rows. The NVL function forces group functions to include null values.

  ```
  SELECT AVG(NVL(comm, 0))
  FROM emp;
  ```

Applying Multiple-row functions to groups of rows in a table

Examples

- To show the department-wise average salary,

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

  Note that all columns in the SELECT list that are not in group functions must be in the GROUP BY clause.

- To show the job-wise total salary for each department

  ```
  SELECT deptno, job, sum(sal)
  FROM emp
  GROUP BY deptno, job;
  ```

Excluding groups result: In the same way that we use the WHERE clause to restrict the rows that we select, the HAVING clause is used to restrict groups. First the group function is applied and the groups matching the HAVING clause are displayed. The syntax of the SELECT statement showing the HAVING clause along with the GROUP BY clause is shown below:-

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

The HAVING clause can precede the GROUP BY clause but it is recommended that the GROUP BY clause come first because it is more logical.

Examples

- To show the department-wise average and maximum salary, in the descending order of average salary, for all departments having average salary higher than 4500.

  ```
  SELECT DEPTNO, AVG(SAL), MAX(SAL)
  FROM EMP
  GROUP BY DEPTNO
  ```

```
HAVING AVG(SAL) > 2000
ORDER BY AVG(SAL)
```

- To display the job title and total monthly salary for each job title with a total payroll exceeding 5000.

```
SELECT JOB, SUM(SAL) PAYROLL
FROM EMP
WHERE JOB NOT LIKE 'SALES%'
GROUP BY JOB
HAVING SUM(SAL) > 5000
ORDER BY SUM(SAL);
```

Nesting Group Functions

- To display the maximum average salary by nesting group functions

```
SELECT max(avg(sal))
FROM emp
GROUP BY deptno;
```

EXERCISES

1. Write down the result of the function calls in the first column to the second column.

Function Call	Result
SUBSTR(CONCAT('HIGH', 'SALARY'), 4, 6)	
CONCAT(SUBSTR('INFORMATION', 3, 4), 'TECH')	
INSTR(CONCAT('GET', 'ING'), 'TIN')	
ROUND(69.476, 1)	
TRUNC('13-MAR-90', 'MONTH')	
TRUNC('13-MAR-90', 'YEAR')	
MOD(90, LENGTH('SEVENTY'))	
MONTHS_BETWEEN('14-AUG-96', '23-MAR-95')	

2. Write down SQL queries to perform following functions:-

i. To show the current date. Label the column *Current Date*.

ii. To display the employee number, name, salary, salary increase by 15% expressed as a whole number (labeled as *New Salary*), the difference between old salary and new salary (labeled as *Increment*).

iii. To display the employee name and calculate the number of months between today and the date the employee was hired (Labeled as *Months_Worked*). Order the results by the number of months employed and round the number of months up to the closest whole number.

iv. Write a query that produces the following for each employee:
<*employee name*> earns <*salary*> monthly

v. To display the employee's name (labeled *name*) with the first letter capitalized and all other letters lowercase and the length of their name (labeled *length*), for all employees whose name starts with J, A or M.

vi. To list the name, hiredate, and day of the week (labeled *DAY*) on which job was started. Order the result by day of week starting with Monday.

vii. To display the job-wise count of employees in each department as follows:-

DEPTNO	**JOB**	**NUM_EMP**

viii. To display the department name, location name, number of employees and the average salary for all employees in that department. Label the columns DNAME, LOC, NUMBER OF PEOPLE and SALARY, respectively. Round the average salary to two decimal places.

ix. To display the employee name, department number and job title for all employees whose department location is *Dallas*.

x. To display the difference between the highest and lowest salaries (Labeled as *DIFFERENCE*)

xi. To show the manager name, MANAGER, and the number of employees, NUM, working under him.

Lab Session 06

Apply subqueries and compound queries in SQL

Why use subqueries?

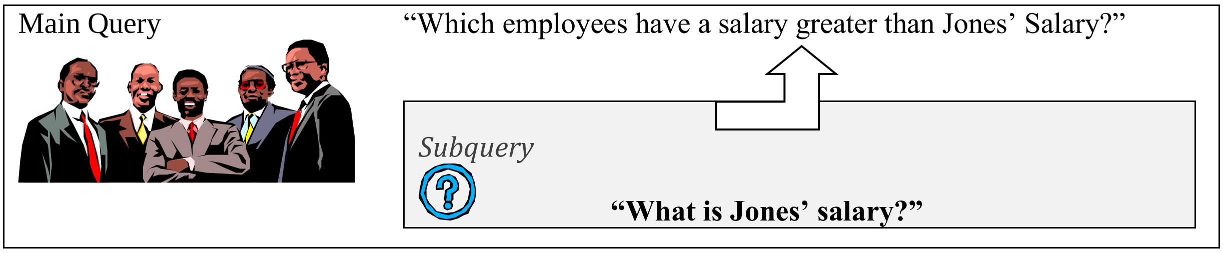


Figure 6.1 Subquery

The inner query or the *subquery* returns a value that is used by the outer query or the main query. Using a subquery is equivalent to performing two sequential queries and using the result of the first query as the *search value* in the second query.

The subquery can be placed in a number of SQL clauses:

- WHERE clause
- HAVING clause
- FROM clause

The syntax of SELECT statement using subqueries is

```
SELECT          select_list
FROM  table
WHERE expr operator
(SELECT         select_list
FROM            table);
```

Note: In the syntax, operator means comparison operator. Comparison operators fall into two clauses: single-row operators (>, =, >=, <, <>, <=) and multiple-row operators (IN, ANY, ALL).
For example, to display the names of all employees who earn more than employee with number 7566.

```
SELECT ename
FROM emp
WHERE sal >
(SELECT sal
FROM emp
WHERE empno = 7566);
```

Types of Subqueries

Single-row subquery: Query that returns only one row from the inner SELECT statement.
Multiple-row subquery: Query that returns more than one row form the inner SELECT statement.
Multiple-column subquery: Query that returns more than one column from the inner SELECT statement.

Single-Row Subqueries

Examples

- To display the employees whose job title is the same as that of employee 7369.

```
SELECT ename, job
FROM emp
```

```
WHERE job =
(SELECT job
FROM emp
WHERE empno = 7369);
```

- To display employees whose job title is the same as that of employee 7369 and whose salary is greater than that of employee 7876.

```
SELECT  ename, job
FROM  emp
WHERE  job =
(SELECT  job
FROM  emp
WHERE  empno = 7369)
AND  sal  >
(SELECT  sal
FROM  emp
WHERE  empno = 7876);
```

- We can display data from a main query by using a group function in a subquery to return a single row. e.g. to display the employee name, job title and salary of all employees whose salary is equal to the minimum salary.

```
SELECT ename, job, sal
FROM emp
WHERE sal = (SELECT MIN(sal) FROM emp);
```

- We can use subqueries not only in the WHERE clause, but also in the HAVING clause. The Oracle server executes the subquery and the results are returned into the HAVING clause of the main query. E.g. to display all departments that have a minimum salary greater than that of department 20.

```
SELECT  deptno, MIN(sal)
FROM  emp
GROUP BY  deptno
HAVING MIN(sal) >
(SELECT MIN(sal)
FROM emp
WHERE  deptno = 20);
```

Multiple-Row Subqueries

Multiple-row subqueries return more than one row. We use multiple-row operator, instead of a single-row operator, with a multiple-row subquery. The multiple-row operator expects one or more values. Following table illustrates multiple row operators.

Operator	Meaning
IN	Equal to any member in the list
ANY	Compare value to each value returned by the subquery
ALL	Compare value to every value returned by the subquery

Table 6.1

Note: The **NOT** operator can be used with IN, ANY, and ALL operators.

Examples

- Find the employees who earn the same salary as the minimum salary for departments.

```
SELECT  ename, sal, deptno
FROM  emp
```

```
WHERE  sal  IN  (SELECT   MIN(sal)
FROM  emp
GROUP  BY  deptno);
```

- To display employees whose salary is less than any clerk and who are not clerks.

```
SELECT  empno, ename, job
FROM  emp
WHERE     sal < ANY
(SELECT  sal
FROM  emp
WHERE   job = 'CLERK') AND JOB <> 'CLERK';
```

- To display employees whose salary is greater than the average salary of all the departments.

```
SELECT  empno, ename, job
FROM  emp
WHERE  sal > ALL
(SELECT  avg(sal)
FROM  emp
GROUP BY deptno);
```

Multiple-Column Subqueries

If we want to compare two or more columns, we must write a compound WHERE clause using logical operators. Multiple column subqueries enable us to combine duplicate WHERE conditions into a single WHERE clause. For example, to display the name of all employees who have done their present job somewhere before in their career.

```
SELECT ENAME
FROM EMP
WHERE (EMPNO, JOB)
IN
(SELECT EMPNO, JOB
FROM JOB_HISTORY)
```

COMPOUND QUERIES

In SQL, we can use the normal set operators of Union, Intersection and Set Difference to combine the results of two or more component queries into a single result table. Queries containing SET operators are called *compound* queries. The following table shows the different set operators provided in Oracle SQL.

Operator	Returns
UNION	All distinct rows selected by either query
UNION ALL	All rows selected by either query including all duplicates
INTERSECT	All distinct rows selected by both queries
MINUS	All distinct rows that are selected by the first SELECT statement and that are not selected in the second SELECT statement

Table 6.2

Restrictions on using set Operators

There are restrictions on the tables that can be combined using the set operations, the most important one being that the two tables have to be union-compatible; that is, they have the same structure. This implies that the two tables must contain the same number of columns, and that their corresponding columns contain the same data types and lengths. It is the user's responsibility to ensure that values in corresponding columns come from the same domain. For example, it would not be sensible to combine a column containing the age of staff with the number of rooms in a property, even though both columns may have the same data type i.e. NUMBER.

The UNION Operator

The UNION operator returns rows from both queries after eliminating duplicates. By default, the output is sorted in ascending order of the first column of the SELECT clause.
For example to display all the jobs that each employee has performed, the following query will be given. (NOTE: If an employee has performed a job multiple times, it will be shown only once)

```
SELECT EMPNO, JOB
FROM JOB_HISTORY
UNION
SELECT EMPNO, JOB
FROM EMP;
```

The UNION ALL Operator

The UNION ALL operator returns rows from both queries including all duplicates. For example to display the current and previous jobs of all employees, the following query will be given. (NOTE: If an employee has performed a job multiple times, it will be shown separately).

```
SELECT EMPNO, JOB
FROM JOB_HISTORY
UNION ALL
SELECT EMPNO, JOB
FROM EMP;
```

The INTERSECT Operator

The INTERSECT operator returns all rows that are common to both queries. For example, to display all employees and their jobs those have already performed their present job somewhere else in the past.

```
SELECT EMPNO, JOB
FROM JOB_HISTORY
INTERSECT
SELECT EMPNO, JOB
FROM EMP;
```

The MINUS Operator

The MINUS operator returns rows from the first query that is not present in the second query. For example to display the ID of those employees whose present job is the first one in their career.

```
SELECT EMPNO, JOB
FROM EMP
MINUS
SELECT EMPNO, JOB
FROM JOB_HISTORY;
```

EXERCISE

1. Write down SQL queries to perform following functions:-

   i. To display the employee number and name for all employees who earn more than the average salary. Sort the results in descending order of salary.

ii. To display the employee name and salary of all employees who report to *king*.

iii. To display the department number, name and job for all employees in the *Sales* department.

iv. To display the name, hiredate and salary for all employees who have both the same salary and commission as *scott*.

v. To display the employee name, department number and job title for all employees whose location is *Dallas*.

vi. List the id of all employees who have not performed the job of *analyst* anywhere in their career. (Note: Use set operators)

vii. Write a query to display the employee name and hiredate for all employees in the same department as Blake. Exclude Blake.

viii. Display the employee number, name and salary for all employees who earn more than the average salary and who work in department with any employee with a T in their name.

Lab Session 07

Apply Data manipulation operations in SQL

Data-Manipulation Language

Data manipulation language is a core part of SQL. When we want to add, update or delete data in the database, we execute a DML statement. A collection of DML statements that form a logical unit of work is called a *transaction*.
Consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction might consist of three separate operations: decrease the savings account, increase the checking account, and record the transaction in the transaction journal. The transaction control must be in operation then.

The SQL DML includes statements to perform following operations:-

Statement	Description
INSERT	Enter new rows into tables
UPDATE	To change existing rows
DELETE	To delete existing rows

Table 8.1

Adding a new row to a table
We can add new rows to a table by using the INSERT statement. The syntax is
INSERT INTO table [(column [, column ...])]
VALUES (value [, value ...]);

<u>Examples</u>

- Inserting a new row in the dept table
  INSERT INTO dept (deptno, dname, loc)
  VALUES (50, ‘DEVELOPMENT’, ‘DETROIT’);
  Note: If the column list is not included, the values must be listed according to the default order of the columns in the table. The order can be seen using the DESCRIBE command in SQL*PLUS (See lab session 1)
- Inserting rows with Null values
  - *Implicit Method*: Omit the column from the column list.
    INSERT INTO dept (deptno, dname)
    VALUES (60, ‘MIS’);
  - *Explicit Method:* Specify the NULL keyword
    INSERT INTO dept
    VALUES (70, ‘FINANCE’, NULL);

  Note: The oracle server automatically enforces all datatypes, data ranges and data integrity constraints. Any column that is not listed explicitly obtains a null value in the new row.
- Using special values, for example, SYSDATE function, to obtain data for a column when inserting a row in a table
  INSERT INTO emp (empno, ename, job, mgr, hiredate, sal, comm, deptno)
  VALUES (7196, ‘GREEN’, ‘SALESMAN’, 7782, SYSDATE, 2000, NULL, 10);
  Similarly we can also use the USER function when inserting rows in a table. The USER function records the current username.
- Adding a new employee by inserting specific date values
  INSERT INTO emp
  VALUES (2296, ‘AROMANO’, ‘SALESMAN’, 7782, TO_DATE(‘FEB 3, 97’, ‘MON DD, YY’), 1300, NULL, 10);

- We can produce an INSERT statement that allows the user to add values interactively by using SQL*Plus substitution variables.

```
INSERT INTO dept (deptno, dname, loc)
VALUES (&department_id, ‘&department_name’, ‘&location’);
```

```
Enter value for department_id: 80
Enter value for department_name: EDUCATION
Enter value for location: ATLANTA

1 row created
```

- Copying rows from another table
  We can use the INSERT statement to add rows to a table where the values are derived from some other existing table. In place of the VALUES clause, we use a subquery. e.g. to insert rows from EMP table to EMP10 table,

```
INSERT INTO EMP10
SELECT * FROM EMP
WHERE DEPTNO = 10;
```

Changing data in a table

We can modify existing rows in a table with the UPDATE statement. The syntax is
UPDATE table
SET column = value [, column = value, ...]
[WHERE condition];
As shown in the above syntax, we can update more than one row at a time depending on a condition.

<u>Examples</u>

- To transfer an employee with number 7782 to department 20.

```
UPDATE  emp
SET  deptno = 20
WHERE empno = 7782;
```

- All rows in the table are modified if the WHERE clause is omitted.

```
UPDATE  emp
SET  deptno = 20;
```

- Updating with multiple column subquery: Update employee 7698’s job and department to match that of employee 7499.

```
UPDATE emp
SET (job, deptno) =
(SELECT job, deptno
FROM emp
WHERE  empno = 7499)
WHERE empno = 7698;
```

Removing a row from a table

We can remove existing rows from a table by using the DELETE statement. The syntax is
DELETE [FROM] table
[WHERE condition];

<u>Examples</u>

- Specific rows are deleted from a table by specifying the WHERE clause.

```
DELETE FROM department
```

```
        WHERE  dname = ‘DEVELOPMENT’;
```

- All rows in the table are deleted if we omit the WHERE clause.

```
        DELETE FROM department;
```

- Remove all employees who started after January 1, 1997.

```
        DELETE FROM employee
        WHERE hiredate > TO_DATE(’01.01.97’, ‘DD.MM.YY’);
```

- Deleting rows based on another table by using subqueries in DELETE statements.

```
        DELETE from employee
        WHERE       deptno =
                            (SELECT  deptno
                            FROM  dept
                            WHERE  dname = ‘SALES’);
        Delete record of employees in department 30
        DELETE FROM employee
        WHERE DEPTNO = 30;
```

Database Transactions

The oracle server ensures data consistency based on transactions. Transactions consist of DML statements that makeup one consistent change to the data. For example, a transfer of funds between two accounts should include the debit to one account and a credit to another account in the same amount. Both actions should either fail or succeed together. The credit should not be committed without the debit.

Transaction Types

Type	Description
Data Manipulation language (DML)	Consists of any number of DML statements that the Oracle Server treats as a single entity or a logical unit of work
Data Definition language (DDL)	Consists of only one DDL statement
Data Control language (DCL)	Consists of only one DCL statement

Table 8.2

A transaction begins when the first executable SQL statement is encountered and terminates when one of the following occurs:

- A COMMIT or ROLLBACK statement is issued
- A DDL statement, such as CREATE, is issued
- A DCL statement is issued
- The user exits SQL*Plus
- A machine fails or the system crashes

After one transaction ends, the next executable SQL statement automatically starts the next transaction. A DDL or DCL statement is automatically committed and therefore implicitly ends a transaction.

Transaction Control

COMMIT: Ends the current transaction by making all pending data changes permanent.
ROLLBACK: Ends the current transaction by discarding all pending data changes.
SAVEPOINT: Marks a savepoint within the current transaction.

Example

To create a new advertising department with at least one employee and make the data changes permanent.

- INSERT INTO dept (deptno, dname, loc)
  VALUES (50, ‘ADVERTISING’, ‘ATLANTA’);
  UPDATE EMP
  SET DEPTNO = 50
  WHERE EMPNO = 7566;
  COMMIT;

EXERCISES

1. Write a transaction to insert following rows in EMP table.

EMPNO	**ENAME**	**JOB**	**MGR**	**HIREDATE**	**SAL**	**COMM**	**DEPTNO**
7123	RALPH	DESIGNER	7566	21-APR-85	2300		50
7890	GEORGE	CLERK	7566	03-MAY-85	1235		50
7629	BOB	SALESMAN	7698	06-MAR-86	1800	1000	30

2. Write down SQL statements to perform following functions:-

i. Increase the salary by 250 of all clerks with a salary less than 900.

ii. Transfer the employee with number 7890 to department 20 and increase his salary by 15%.

iii. Increase the salary of employee with number 7369 by 10% of the salary of employee with number 7499.

iv. Assign to employee 7876 the same manager as the employee 7900.

v. Remove all employees who were hired before 1981.

Lab Session 08

Explore management of tables and views

Database Objects

An Oracle database can contain multiple data structures. Each structure should be outlined in the database design so that it can be created during the build stage of database development.

Table: Stores data
View: Subset of data from one or more tables
Sequence: Generates primary key values
Index: Improves the performance of some queries
Synonym: Gives alternative names to objects

Oracle Table Structures

- Tables can be created at any time, even while users are using the database.
- We do not need to specify the size of any table. The size is ultimately defined by the amount of space allocated to the database as a whole.
- Table structure can be modified online.

Naming Conventions

- Name database tables and columns according to the standard rules for naming any Oracle database object.
- Table names and column names must begin with a letter and can be 1-30 characters long.
- Names must contain only the characters A-Z, a-z, 0-9, _*(underscore*), $, and # (legal characters, but their use is discouraged).
- Names must not duplicate the name of another object owned by the same Oracle Server user.
- Names must not be an Oracle Server reserved word.

Creating and Altering Tables

The CREATE TABLE statement

To create a table, a user must have the CREATE TABLE privilege and a storage area in which to create objects. The database administrator uses data control language (DCL) statements, covered in a later session, to grant privileges to users.

The syntax is as follows:-
CREATE TABLE [*schema* .] *table*
(*column datatype* [DEFAULT *expr*] [, …]);

Referencing another user’s tables

A *schema* is a collection of objects. *Schema objects* are the logical structures that directly refer to the data in a database. Schema objects include tables, views, synonyms, sequences, stored procedures, indexes, clusters, and database links.
If a table does not belong to the user, the owner’s name must be prefixed to the table.

The DEFAULT option

A column can be given a default value by using the DEFAULT option. This option prevents null values from entering the columns if a row is inserted without a value for the column. The default value can be a literal, an expression, or a SQL function, such as SYSDATE and USER, but the value cannot be the name of another column or a pseudocolumn, such as NEXTVAL, or CURRVAL. The default expression must match the datatype of the column.

For example,
… hiredate DATE DEFAULT SYSDATE, …

Example

The following example creates the DEPT table mentioned in the lab. session 01.

```
CREATE TABLE dept
(deptno  NUMBER(2),
dname   VARCHAR2(14),
loc        VARCHAR2(13));
```

Since creating a table is a DDL statement, an automatic commit takes place when this statement is executed. In order to confirm the creation of the table, issue the DESCRIBE command as discussed in lab session 01.

```
DESCRIBE DEPT
```

SQL Data Types

Datatype	Description
VARCHAR2(*size*)	Variable-length character data (A maximum *size* must be specified. Default and minimum *size* is 1; maximum *size* is 4000)
CHAR(*size*)	Fixed-length character data of length *size* bytes (Default and minimum *size* is 1; maximum *size* is 2000)
NUMBER(*p*, *s*)	Number having precision p and scale s (The precision is the total number of decimal digits and the scale is the number of digits to the right of the decimal point. The precision can range from 1 to 38 and the scale can range from -84 to 127.)
DATE	Date and time values between January 1, 4712 B.C. and December 31, 9999 A.D.
LONG	Variable length character data up to 2 gigabytes
CLOB	Single-byte character data up to 2 gigabytes
RAW(*size*)	Raw binary data of length *size* (A maximum size must be specified. Maximum *size* is 2000.)
LONG RAW	Raw binary data of variable length up to 2 gigabytes
BLOB	Binary data up to 4 gigabytes
BFILE	Binary data stored in an external file; up to 4 gigabytes

Table 9.1

CLOB, BLOB and BFILE are the large object data types and can store blocks of unstructured data (such as text, graphics images, video clips and sound wave forms up to 4 gigabytes in size.) LOBs also support random access to data.

Creating a table by using a Subquery

The following example creates a table, DEPT30, that contains details of all employees working in department 30

```
CREATE TABLE  dept30
AS SELECT  empno, ename, sal * 12 ANNSAL, hiredate
FROM  emp
```

```
WHERE  deptno = 30;
```

Altering table structure

The ALTER TABLE statement is used to

- Add a new column
- Modify an existing column
- Define a default value for the new column

The following example adds a new column to the DEPT30 table:-

```
ALTER TABLE dept30
ADD    (job  VARCHAR2(9));
```

To modify an existing column, use

```
ALTER TABLE dept30
MODIFY         (ename  VARCHAR2(15));
```

Dropping a Table

The DROP TABLE statement removes the definition of an Oracle table. The database loses all the data in the table and all the indexes associated with it.
The DROP TABLE statement, once executed, is irreversible. The Oracle Server does not question the action when the statement is issued and the table is immediately dropped. All DDL statements issue a commit, therefore, making the transaction permanent.

To drop the table DEPT30,

```
DROP TABLE DEPT30;
```

Changing the name of an object

To change the name of a table, view, sequence, or synonym, execute the RENAME statement:-

```
RENAME dept TO department;
```

What are constraints?

The Oracle Server uses constraints to prevent invalid data entry into tables.
Constraints are used for the following purposes:-

- Enforce rules at the table level whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed.
- Prevent the deletion of a table if there are dependencies from other tables.
- Provide rules for Oracle tools, such as Oracle Developer.

The following constraint types valid in Oracle:-

Constraint	Description
NOT NULL	Specifies that this column may not contain a null value
UNIQUE	Specifies a column or combination of columns whose values must be unique for all rows in the table
PRIMARY KEY	Uniquely identifies each row of the table
FOREIGN KEY	Establishes and enforces a foreign key relationship between the column and a column of the referenced table
CHECK	Specifies a condition that must be true

Table 9.2

Constraint Guidelines

- All constraints are stored in the data dictionary
- Name a constraint or the Oracle server will generate a name by using the SYS_Cn format
- Create a constraint
  - At the same time as the table is created
  - After the table has been created

The EMP table is being created specifying various constraints:-

```
CREATE TABLE DEPT (
DEPTNO         NUMBER(2) constraint DEPT_DEPTNO_PK PRIMARY KEY,
DNAME          VARCHAR2(14),
LOC            VARCHAR2(13),
CONSTRAINT DEPT_DNAME_UK            UNIQUE(DNAME));

CREATE TABLE EMP (
EMPNO          NUMBER(4) CONSTRAINT  EMP_EMPNO_PK  PRIMARY KEY,
ENAME          VARCHAR2(10) NOT NULL,
JOB            VARCHAR2(9),
MGR            NUMBER(4),
HIREDATE   DATE     DEFAULT      SYSDATE,
SAL            NUMBER(7, 2),
COMM           NUMBER(7, 2),
DEPTNO         NUMBER(7)    NOT NULL,
CONSTRAINT          EMP_DEPTNO_CK     CHECK (DEPTNO BETWEEN 1 AND 50),
CONSTRAINT EMP_DEPTNO_FK     FOREIGN KEY (DEPTNO)
REFERENCES DEPT(DEPTNO));
```

Composite primary keys are defined at the table level.

Creating Views

What is a View?

A view is a logical table based on other tables or another view

Simple and Complex Views

There are two classifications for views: simple and complex. The basic difference is related to the DML (insert, update and delete) operations.

Feature	Simple Views	Complex Views
Number of tables	One	One or more
Contain functions	No	Yes
Contain groups of data	No	Yes
DML through view	Yes	Not always

Table 9.3

Creating a View

We can create a view by embedding a subquery within the CREATE VIEW statement. The syntax is as follows:-

```
CREATE [OR REPLACE] VIEW view
AS subquery;
```

For example, to create a view, EMPVU10, that contains the employee number, name and job title for all the employees in department 10.

```
CREATE VIEW empvu10
AS SELECT empno, ename, job
FROM emp
WHERE deptno = 10;
```

We can display the structure of the view by using the SQL*Plus DESCRIBE command as follows:-

```
DESCRIBE empvu10
```

We can also create views by using column aliases in the subquery.

```
CREATE VIEW salvu30
AS SELECT empno EMPLOYEE_NUMBER, ename NAME, sal SALARY
FROM emp
WHERE deptno = 30;
```

Now select the columns from this view by the given alias names.
The data from the view would be retrieved as follows:-

```
SELECT *
FROM salvu30;
```

Views in the data dictionary

Once a view has been created, we can query the data dictionary table called USER_VIEWS to see the name of the view and the view definition. The text of the SELECT statement that constitutes the view is stored in a LONG column.

Creating a Complex View

A complex view contains columns from multiple tables and may also include group functions.

- To create a complex view to show employee number, employee name and department name, we would have to join EMP and DEPT tables as follows:-

  ```
  CREATE VIEW EMP_DEPT
  AS
  SELECT EMPNO, ENAME, DNAME
  FROM EMP, DEPT
  WHERE EMP.DEPTNO = DEPT.DEPTNO;
  ```

- To create a complex view that contains group functions to display values from two tables.

  ```
  CREATE VIEW dept_sum_vu (name, minsal, maxsal, avgsal)
  AS SELECT d.dname, MIN(e.sal), MAX(e.sal), AVG(e.sal)
  FROM EMP e, DEPT d
  WHERE e.DEPTNO = d.DEPTNO
  GROUP BY d.dname;
  ```

Removing a View

We can remove a view without losing data because a view is based on underlying tables in the database.

The syntax is

```
DROP VIEW view;
```

For example to drop the empvu10 view,

```
DROP VIEW empvu10;
```

EXERCISES

Consider the following schema, in the form of normalized relations, to represent information about *employees*, *grades*, *training* and *projects* in an organization.

EMPLOYEE
Empno (eg 6712)
Name
Designation (e.g. *Database Developer*)
Qualification
Joindate

GRADE
Designation
Grade (1-20)
TotalPosts
PostsAvailable (<= TotalPosts)

PROJECT
PID (eg P812)
Title
Client
Duration (in weeks)
Status (New, In Progress, Complete)

TRAINING
Tcode (eg T902)
Title
StartDate
EndDate

EMP_PROJECT
Empno
PID
Performance (Excellent, Good, Fair, Bad, Poor)

EMP_TRAINING
Empno
Tcode
Attendance (%)

1. Develop a script file **EMPLOYEE.SQL** to create tables for the above schema. Implement all necessary *integrity constraints* including primary and foreign keys. (NOTE: All ***check*** constraints should be at table level)

2. Write SQL statements to add
   - *Gender* column to **EMP** table. The only possible values are *Male* and *Female.*
   - *Instructor_Name* column to **TRAINING** table.
   - *Salary* column to **GRADE** table.

3. Write down a transaction to insert following data in GRADE table. The data should be finally saved in the database.

S.N.	Designation	Grade	TotalPosts	PostsAvailable
1.	CLERK	12	8	8
2.	PROGRAMMAR	17	6	6
3.	DATABASE DEVELOPER	17	4	4
4.	SENIOR SYSTEM ANALYST	18	2	2

4. Write down a transaction to insert following data in EMPLOYEE table. The data should be finally saved in the database.

S.N.	Empno	Name	Designation	Qualification	JoinDate
1.	3400	ASIF	Database Developer	B.E.	25th Nov, 2000
2.	3108	MEHMOOD	Senior System Analyst	M.S.	16th May, 1996
3.	3345	FARHAN	Database Developer	B.E.	23rd April, 2001
4.	3315	NAVEED	Clerk	B.A.	11th Jan, 1997
5.	3300	NAVEEN	Clerk	B.A.	24th Feb, 1998

5. Write down a transaction to insert following data in TRAINING table. The data should be finally saved in the database.

S.N.	Tcode	Title	StartDate	EndDate
1.	TR-03	Developer 6i	23rd May, 2002	13th July, 2002
2.	TR-13	Java	14th Jan, 2001	15th March, 2001
3.	TR-17	Typing/Short Hand	22nd June, 2001	22nd July, 2001

6. Write down a transaction to insert following data in PROJECT table. The data should be finally saved in the database.

S.N.	**PID**	**Title**	**Client**	**Duration (in weeks)**	**Status**
1.	P023	Payroll	Superior Bank	12	New
2.	P321	Accounts	ABC Leasing	16	In Progress
3.	P178	Taxation	Farhan Motors	4	Complete
4.	P315	Payroll	ABC Leasing	8	New
5.	P300	Inventory	Khurran Textile Mills	12	In Progress

7. Write down a transaction to insert data in EMP_TRAINING table. The data should be finally saved in the database.
   i. Employee 3400 gets *Developer 6i* training and his attendance is 87%
   ii. Employee 3300 gets *Typing/shorthand* training and her attendance is 95%

8. Write down a transaction to insert data in EMP_PROJECT table. The data should be finally saved in the database.
   i. Employee 3400 has ***good** performance* in project P023.
   ii. Employee 3300 has ***Excellent** performance* in project P023.
   iii. Employee 3345 has ***Bad** performance* in project P321.

9. Create views for following purposes:-
   i. To display each designation and number of employees with that particular designation.

   ii. To display employee number, employee name, project title and employee performance in that project.

   iii. To display employee number, employee name and number of projects in which employee performance is *excellent*.

Lab Session 09

Apply sequences, indexes and synonyms

Sequences

A Sequence generator can be used to automatically generate sequence numbers for rows in tables. A sequence is a database object created by a user and can be shared by multiple users. A typical usage for sequences is to create a primary key value, which must be unique for each row. The sequence is generated and incremented (or decremented) by an internal Oracle routine. Sequence numbers are stored and generated independently of tables. Therefore, the same sequence can be used for multiple tables.

Creating Sequences

Following is the syntax of SQL statement to create sequences:-

```
CREATE SEQUENCE sequence
[INCREMENT BY n]
[START WITH n];
```

For example, creating a sequence named DEPT_DEPTNO to be used for the primary key of the DEPT table.

```
CREATE SEQUENCE dept_deptno
INCREMENT BY 1
START WITH 50;
```

NEXTVAL and CURRVAL Pseudocolumns

The NEXTVAL pseudocolumn is used to extract successive sequence numbers from a specified sequence. We must qualify NEXTVAL with the sequence name. When we reference *sequence*. NEXTVAL, a new sequence number is generated and the current sequence number is placed in CURRVAL. NEXTVAL returns the next available sequence value. It returns a unique value every time it is referenced, even for different users. CURRVAL obtains the current sequence value. NEXTVAL must be issued for that sequence before CURRVAL contains a value.

Using a Sequence

Insert a new department named MARKETING in San Diego

```
INSERT INTO dept (deptno, dname, loc)
VALUES (dept_deptno.NEXTVAL, ‘MARKETING’, ‘SAN DIEGO’);
```

In order to view the current value for the DEPT_DEPTNO sequence

```
SELECT dept_deptno.CURRVAL
FORM dual;
```

Removing a sequence

A sequence can be removed by using the DROP SEQUENCE statement. Once removed, the sequence can no longer be referenced.

```
DROP SEQUENCE dept_deptno;
```

Indexes

An Oracle Server index is a schema object that can speed up the retrieval of rows by using a pointer. Indexes can be created explicitly or automatically. An index provides direct and fast access to rows in a table. Its purpose is to reduce the necessity of disk I/O by using an indexed path to locate data quickly. The index is used and maintained automatically by the Oracle Server. Once an index is created, no direct activity is required by the user. Indexes are logically and physically independent of the table they index. Therefore, they can be created or dropped at any time and have no effect on the base tables or other indexes.

Types of indexes
Oracle maintains the indexes automatically: when new rows are added to the table, updated, or deleted, Oracle updates the corresponding indexes. We can create the following indexes:-

Bitmap index
A bitmap index does not repeatedly store the index column values. Each value is treated as a key, and for the corresponding ROWIDs a bit is set. Bitmap indexes are suitable for columns with low cardinality, such as the GENDER column in the EMP table, where the possible values are M or F. The cardinality is the number of distinct column values in a column. In the EMP table column, the cardinality of the SEX column is 2.

B-tree index
This is the default. The index is created using the b-tree algorithm. The b-tree includes nodes with the index column values and the ROWID of the row. The ROWIDs are used to identify the rows in the table.

The following are the types of b-tree indexes:-

- Unique Index: The Oracle server automatically creates this index when a column in a table is defined to be a PRIMARY KEY or UNIQUE key contraint.
- NonUnique Index: Users can create nonunique indexes on columns to speed up access time to the rows. For example, we can create a FOREIGN KEY column index for a join in a query to improve retrieval speed.
- Function-based index: The function-based index can be created on columns with expressions. For example, creating an index on the SUBSTR(EMPID, 1, 2) can speed up the queries using the SUBSTR(EMPID, 1, 2) in the WHERE clause.

Creating an Index

- To create an index (b-tree) on ENAME column in the EMP table.
  ```
  CREATE INDEX emp_ename_idx
  ON emp(ename);
  ```
- To create an index (b-tree) on first 5 characters of JOB column in the EMP table.
  ```
  CREATE INDEX emp_job5_idx
  ON emp(SUBSTR(JOB, 1, 5));
  ```
- To create a bitmap index, we must specify the keyword BITMAP immediately after CREATE. Bitmap indexes cannot be unique. For example:
  ```
  CREATE BITMAP INDEX IND_PROJ_STAT
  ON PROJECT (STATUS);
  ```

Confirming Indexes

We can confirm the existence of indexes from the USER_INDEXES data dictionary view. It contains the name of the index and its uniqueness.

```
SELECT INDEX_NAME, TABLE_NAME, TABLE_OWNER, UNIQUENESS
FROM USER_INDEXES;
```

Removing an Index

It is not possible to modify an index. To change it, we must drop it first and then re-create it. Remove an index definition from the data dictionary by issuing the DROP INDEX statement. To drop an index, one must be the owner of the index or have the DROP ANY INDEX privilege.
DROP INDEX *index*;

For example, remove the EMP_ENAME_IDX index from the data dictionary.
DROP INDEX emp_ename_idx;

Synonyms

In order to refer to a table owned by another user, it is necessary to prefix the table name with the name of the user who created it followed by a period. Creating a synonym eliminates the need to qualify the object name with the schema and provides with an alternative name for a table, view, sequence, procedure, or other object. This method can be especially useful with lengthy object names, such as views.
The syntax is
CREATE [PUBLIC] SYNONYM *synonym*
FOR *object*;

To create a shortened name for the DEPT_SUM_VU view,
CREATE SYNONYM d_sum
FOR dept_sum_vu;

The DBA can create a public synonym accessible to all users. e.g. to create a public synonym named DEPT for SCOTT's DEPT table:
CREATE PUBLIC SYNONYM DEPT
FOR SCOTT.DEPT;

To drop a synonym,
DROP SYNONYM DEPT;

EXERCISES

Consider the schema of the previous lab session that represents information about *employees, grades, training* and *projects* in an organization and answer the following questions.

1. Create a sequence to generate the primary key column EMPNO of EMPLOYEE table in the lab session 06. The sequence should start with 1, increment by 1 and have maximum value of 10000.

2. Create **B-Tree** indexes on i) **Name** column of EMP table ii) **Designation** column of EMP table iii) First 10 characters of **Title** in TRAINING table

3. Create **bitmapped** indexes on i) **Gender** column of EMP table ii) **Performance** column of EMP_PROJECT table.

Lab Session 10

Explore PL/SQL Programming with executable statements

INTRODUCTION

Procedural Language/SQL (PL/SQL) is Oracle Corporation's procedural language extension to SQL, the standard data access language for relational databases. It allows the data manipulation and query statements of SQL to be included in block-structured and procedural units of code, making PL/SQL a powerful *transaction processing* language.

PL/SQL offers modern software engineering features such as data encapsulation, exception handling, information hiding, and object orientation, and so brings state-of-the-art programming to the Oracle Server and Toolset.

PL/SQL BLOCK STRUCTURE

A PL/SQL block consists of up to three sections: ***declarative*** (optional), ***executable*** (required), and ***exception*** handling (optional). Only BEGIN and END keywords are required. The following illustrates the PL/SQL block structure:-

- **DECLARE** - Optional
  - Variables, cursors, user-defined exceptions
- **BEGIN** – mandatory
  - SQL statements
  - PL/SQL statements
- **EXCEPTION** – optional
  - Actions to perform when errors occur
- **END**; - Mandatory

Table 11.1

Block Types

Every unit of PL/SQL comprises one or more blocks. These blocks can be entirely separate or nested one within another. The basic units (procedures and functions, also known as subprograms, and anonymous blocks) that make up a PL/SQL program are logical blocks, which can contain any number of nested sub-blocks.

Anonymous Blocks

Anonymous blocks are unnamed blocks. They are declared at the point in an application where they are to be executed and are passed to the PL/SQL engine for execution at runtime. You can embed an anonymous block within a precompiler program and within SQL*Plus or Server Manager. Triggers in Oracle Developer components consist of such blocks.

Subprograms

Subprograms are named PL/SQL blocks that can take parameters and can be invoked. We can declare them either as *procedures* or as *functions*. Generally we use a procedure to perform an action and a function to compute a value.

We can store subprograms at the server or application level. Using Oracle Developer components (Forms, Reports and Graphics), we can declare procedures and functions as part of the application (a Form or report) and call them from other procedures, functions and triggers within the same application whenever necessary.

Note: A function is similar to a procedure, except that a function must return a value.

Program Constructs

The following table outlines a variety of different PL/SQL program constructs that use the basic PL/SQL block. They are available based on the environment in which they are executed.

Program Construct	Description	Availability
Anonymous block	Unnamed PL/SQL block that is embedded within an application or is issued interactively	All PL/SQL environments
Stored procedure or function	Named PL/SQL block stored in the Oracle Server that can accept parameters and can be invoked repeatedly by name.	Oracle Server
Application procedure or function	Named PL/SQL block stored in an Oracle Developer application or shared library that can accept parameters and can be invoked repeatedly by name	Oracle Developer components – for example, forms
Package	Named PL/SQL module that groups related procedures, functions and identifiers	Oracle Server and Oracle Developer components – for example, Forms
Database trigger	PL/SQL block that is associated with a database table and is fired automatically when triggered by DML statements	Oracle Server
Application trigger	PL/SQL block that is associated with an application event and is fired automatically	Oracle Developer components – for example, Forms

Table 11.2

Variables in PL/SQL

With PL/SQL we can declare variables and then use them in SQL and procedural statements anywhere an expression can be used.

Uses of Variables

Following are various uses of variables:-

- **Temporary storage of data**
  Data can be temporarily stored in one or more variables for use when validating data input for processing later in the data flow process.

- **Manipulation of stored values**
  Variables can be used for calculations and other data manipulations without accessing the database.

- **Reusability**
  Once declared, variables can be used repeatedly in an application simply be referencing them in other statements, including other declarative statements.

- **Ease of maintenance**
  When using %TYPE and %ROWTYPE, we declare variables basing the declarations on the definitions of database columns. PL/SQL variables or cursor variables previously declared in the current scope may also use the %TYPE and %ROWTYPE attributes as datatype specifiers. If an underlying definition changes, the variable declaration changes accordingly at runtime. This provides data independence, reduces maintenance costs, and allows programs to adapt as the database changes to meet new business needs.

Types of Variables

All PL/SQL variables have a datatype, which specifies a storage format, constraints and valid range of values. PL/SQL supports four datatype categories – scalar, composite, reference, and LOB (large object) – that can be used to declare variables, constants and pointers.

- **PL/SQL variables**
  - **Scalar**: Holds a single value and has no internal components. Scalar data types can be classified into four categories: *number*, *character*, *date* and *Boolean*. Character and number datatypes have subtypes that associate a base type to a constraint. For example, INTEGER and POSITIVE are subtypes of the NUMBER base type.
  - **Composite**: For example, records that allows groups of fields to be defined and manipulated in PL/SQL blocks.
  - **Reference**: Holds values, called pointers that designate other program items.
  - **LOB (large objects)**: Holds values, called locators that specify the location of large objects (graphic images for example) that are stored out of line.

- **Non-PL/SQL variables**
  - Host language variables declared in precompiler programs
  - Screen fields in Forms applications
  - SQL*Plus host variables: PL/SQL does not have input/output capability of its own. In order to pass values into and out of a PL/SQL block, it is necessary to rely on the environment in which PL/SQL is executing is executing. SQL*Plus host (or bind) variables can be used to pass runtime values out of the PL/SQL block back to the SQL*Plus environment. You can reference them in a PL/SQL block with a preceding column.

Declaring PL/SQL variables

It is necessary to declare all PL/SQL identifiers in the declaration section before referencing them in the PL/SQL block. It is optional to assign an initial value. It is not necessary to assign a value to a variable in order to declare it.

Syntax
identifier [CONSTANT] *datatype* [NOT NULL] [:= | DEFAULT expr];

Examples

```
Declare
        v_hiredate      DATE;
        v_deptno        NUMBER(2)    NOT NULL     :=      10;
        v_location      VARCHAR2(13) := ‘Atlanta’;
        c_comm.         CONSTANT     NUMBER       :=      1400;
        v_countBINARY_INTEGER := 0;
        v_orderdate     DATE         :=      SYSDATE  +  7;
```

Guidelines

- Name the identifier according to the same rules used for SQL objects.
- Use naming conventions – for example, v_name to represent a variable, g_name to represent global variables and c_name to represent a constant variable.
- If NOT NULL constraint is used, it is needed to assign a value.
- Declaring only one identifier per line makes code more easily read and maintained.
- Identifiers must not be longer than 30 characters. The first character must be a letter; the remaining characters can be letters, numbers, or special symbols.

Assigning values to variables

Syntax

```
        identifier      :=      expr;
```

Examples

Set a predefined hiredate for new employees

```
        v_hiredate      :=      '31-DEC-98';
```

Set the employee name to 'Maduro'

```
        v_ename         :=      'Maduro';
```

Another way to assign values to variables is to select or fetch database values into it. The following example, computes a 10% bonus on the salary of an employee:-

```
SELECT SAL * 0.10
INTO V_BONUS
FROM EMP
WHERE EMPNO = 7369;
```

Then we can use the variable bonus in another computation or insert its value into a database table.

Variable Initialization and Keywords

Variables are initialized every time a block or subprogram is entered. By default, variables are initialized to NULL.

- Use the assignment operator (:=) for variables that have no typical value.
- V_hiredate := to_date('15-SEP-1999', 'DD-MON-YYYY');
- DEFAULT. We can use the DEFAULT keyword instead of the assignment operator to initialize variables. Use the default for variables that have a typical value.
- g_mgr NUMBER(4) DEFAULT 7839;
- NOT NULL. Impose the NOT NULL constraint when the variable must contain a value. We cannot assign nulls to a variable defined as NOT NULL. The NOT NULL constraint must be followed by an initialization clause.
- v_location VARCHAR2(13) NOT NULL := 'CHICAGO';

Scalar Datatypes

A scalar datatype holds a single value and has no internal components. Scalar datatypes can be classified into four categories: number, character, date, and Boolean. Character and number datatypes have subtypes that associate a base type to a constraint. For example, INTEGER and POSITIVE are subtypes of the NUMBER base type.

Basic Scalar Datatypes

Datatype	Description
VARCHAR2(*maximum length*)	Base type for variable-length character data up to 32,767 bytes. There is no default size for VARCHAR2 variables and constants.
NUMBER [(precision, scale)]	Base type for fixed and floating-point numbers.
DATE	Base type for dates and times. DATE values include the time of day in seconds since midnight. The range for dates is between 4712 B.C. and 9999 A.D.
CHAR [(maximum length)]	Base type for fixed-length character data upto 32, 760 bytes. If the maximum length is not specified, the default length is set to 1.
LONG	Base type for variable-length character data up to 32,760 bytes. The maximum width of a LONG database column is 2,147,483,647 bytes.
LONG RAW	Base type for binary data and byte strings up to 32,760 bytes. LONG RAW data

Datatype	Description
	is not interpreted by PL/SQL.
BOOLEAN	Base type that stores one of three possible values used for logical calculations TRUE, FALSE, or NULL.
BINARY_INTEGER	Base type for integers between –2,147,483,647 and 2,147,483,647.
PLS_INTEGER	Base type for signed integers between –2,147,483,647 and 2,147,483,647. PLS_INTEGER values require less storage and are faster than NUMBER and BINARY_INTEGER values.

Table 11.3

Note: The LONG datatype is similar to VARCHAR2, except that the maximum length of a LONG value is 32,760 bytes.

Scalar Variable Declarations

Following are some examples of variable declarations in PL/SQL.

```
v_job           VARCHAR2(9);
v_countBINARY_INTEGER := 0;
v_total_sal     NUMBER(9, 2) := 0;
c_tax_rate      CONSTANT NUMBER(5,2) := 8.25;
v_valid BOOLEAN NOT NULL := TRUE;
```

The %TYPE Attribute

When we declare PL/SQL variables to hold column values, it is necessary to ensure that the variable is of the correct datatype and precision. Rather than hard coding the datatype and precision of a variable, we can use the %TYPE attribute to declare a variable according to another previously declared variable or database column. The %TYPE attribute is most often used when the value stored in the variable will be derived from a table in the database or if the variable is destined to be written to.

To use the attribute in place of the datatype required in the variable declaration, prefix it with the database table and column name. If referring to a previously declared variable, prefix the variable name to the attribute.

Examples

```
v_ename             emp.ename%TYPE;
v_balance           NUMBER(7,2);
v_min_balance  v_balance%TYPE := 10;
```

Declaring Boolean Variables

Only the values TRUE, FALSE, and NULL can be assigned to a Boolean variable. In PL/SQL, we can compare variables in both SQL and procedural statements. In a SQL statement, we can use Boolean expressions to specify the rows in a table that are affected by the statement. In a procedural statement, Boolean expressions are the basis for conditional control. NULL stands for missing, inapplicable or unknown value.

For example, to declare and initialize a Boolean variable:-

```
v_comm._sal     BOOLEAN      :=       (v_sal1 < v_sal2);
```

Composite Datatypes

Composite datatypes (also known as collections) are TABLE, RECORD, NESTED TABLE, and VARRAY. We use the RECORD datatype to treat related but dissimilar data as a logical unit. The

TABLE data type is used to reference and manipulate collections of data as a whole object. The composite data types are not covered in this workbook.

LOB Datatype Variables

With the LOB (Large Object) Oracle 8 data types, we can store blocks of unstructured data (such as text, graphic images, video clips, and sound wave forms) up to 4 gigabytes in size. LOB data types allow efficient, random, piecewise access to the data and can be attributes of an object type. LOBs also support random access to data.

- The CLOB (character large object) datatype is used to store large blocks of single-byte character data in the database.
- The BLOB (binary large object) datatype is used to store large binary objects in the database in line (inside the row) or out of line (outside the row).
- The BFILE (binary file) datatype is used to store large binary objects in operating system files outside the database.
- The NCLOB (national language character large object) datatype is used to store large blocks of single-byte or fixed-width multi-byte NCHAR data in the database, in line or out of line.

Bind Variables

A bind variable is a variable that is declared in the host environment and then used to pass runtime values, either number or character, into or out of one or more PL/SQL programs, which can use it as they would use any other variable. Variables declared in the host or calling environment can be referenced in PL/SQL statements, unless the statement is in a procedure, function or package. This includes host language variables declared in pre-compiler programs, screen fields in Oracle Developer Forms applications, and SQL*Plus bind variables.

Creating Bind Variables

To declare a bind variable in the SQL*Plus environment, we use the command VARIABLE. For example, to declare a variable of type NUMBER and VARCHAR2 as follow:-

```
VARIABLE     return_code     NUMBER
VARIABLE     return_msg      VARCHAR2 (30);
```

For displaying the bind variables, we use PRINT command.

DBMS_OUTPUT.PUT_LINE

DBMS_OUTPUT is an Oracle-supplied package, and PUT_LINE is a procedure within that package. PUT_LINE procedure is used to display information from a PL/SQL block.
The package must first be enabled on SQL*Plus session. To do this, execute the SQL*Plus command SET SERVEROUTPUT ON.

Example

The following script computes the monthly salary and prints it to the screen, using DBMS_OUTPUT.PUT_LINE.

```
SET SERVEROUTPUT ON
ACCEPT          p_annual_sal     PROMPT          'Please enter the annual salary : '

DECLARE
        v_sal     NUMBER(9, 2) := &p_annual_sal;
BEGIN
        v_sal := v_sal / 12;
        DBMS_OUTPUT.PUT_LINE (‘The monthly salary is ‘ || TO_CHAR(v_sal));
END;
/
PRINT g_monthly_sal
```

Table 11.4

Executable Statements

Identifiers are used to name PL/SQL program items and units, which include constants, variables, exceptions, cursors, cursor variables, subprograms, and packages.

- Identifiers can contain up to 30 characters, but they must start with an alphabetic character.
- Do not choose the same name for the identifier as the name of columns in a table used in the block. If PL/SQL identifiers are in the same SQL statements and have the same name as a column, then Oracle assumes that it is the column that is being referenced.
- Reserved words cannot be used as identifiers unless they are enclosed in double quotation marks (for example, “SELECT”)
- Reserved words should be written in uppercase to promote readability.

A *literal* is an explicit numeric, character, string, or Boolean value not represented by an identifier.

- Character literals include all the printable characters in the PL/SQL character set: letters, numerals, spaces, and special symbols.
- Numeric literals can be represented either by a simple value (for example, -32.5) or by scientific notation (for example, 2E5, meaning 2 * 10 to the power of 5 = 200000).

Commenting Code

Comment code to document each phase and to assist with debugging. Comment the PL/SQL code with two dashes (--) if the comment is on a single line, or enclose the comment between the symbols /* and */ if the comment spans several lines. Comments are strictly informational and do not enforce any conditions or behavior on behavioral logic or data. Well placed comments are extremely valuable for code readability and future code maintenance.

Example

Compute the yearly salary from the monthly salary

```
…
        v_sal     NUMBER(9, 2);
BEGIN
        /* Compute the annual salary based on the monthly
                salary input from the user */
        v_sal := &p_monthly_sal * 12;
END;  -- This is the end of the transaction
```

Table 11.5

SQL Functions in PL/SQL

Most of the functions available in SQL are also valid in PL/SQL expressions:

- Single-row number functions
- Single-row character functions
- Datatype conversion functions
- Date functions
- GREATEST, LEAST
- Miscellaneous functions

The following functions are not available in procedural statements:

- DECODE
- Group functions: AVG, MIN, MAX, COUNT, SUM, STDDEV, and VARIANCE. Group functions apply to groups of rows in a table and therefore we are available only in SQL statements in a PL/SQL block.

Examples

- Build the mailing list for a company

v_mailing_address := v_name || CHR(10) || v_address || CHR(10) || v_state || CHR(10) || v_zip;

- Convert the employees name to lowercase.

v_ename := LOWER(v_ename);

Datatype conversion

PL/SQL attempts to convert datatypes dynamically if they are mixed in a statement. For example, if a NUMBER value is assigned to a CHAR variable, then PL/SQL dynamically translates the number into a character representation, so that it can be stored in the CHAR variable. The reverse situation also applies, providing that the character expression represents a numeric value. Providing that they are comparable, we can also assign characters to DATE variables and vice versa.

Conversion functions

- TO_CHAR
- TO_DATE
- TO_NUMBER

Example 1

```
DECLARE
V_DATE VARCHAR2(15);
BEGIN
SELECT TO_CHAR(HIREDATE, 'MON. DD, YYYY')
INTO V_DATE
FROM EMP
WHERE EMPNO = 7839;
DBMS_OUTPUT.PUT_LINE('HIRE DATE IS ' || V_DATE);
END;
```

Table 11.6

Example 2

v_date := TO_DATE(‘January 13, 1998’, ‘Month DD, YYYY’);

Nested Blocks

One of the advantages that PL/SQL has over SQL is the ability to nest statements. We can nest blocks wherever an executable statement is allowed, thus making the nested block a statement. Therefore, we can break down the executable part of a block into smaller blocks. The exception section can also contain nested blocks.

Variable Scope

The scope of an object is the region of the program that can refer to the object. We can reference the declared variable within the executable section.

Example

```
…
x         BINARY_INTEGER;
BEGIN
                                                   scope of x
          …
          DECLARE
                    Y         NUMBER;
                              scope of y
          BEGIN
                    …
          END;
          …
END;
```

In the nested block shown above, the variable named y can reference the variable named x. Variable x, however, cannot reference variable y. If the variable named y in the nested block is given the same name as the variable named x in the outer block its value is valid only for the duration of the nested block.

Operators in PL/SQL

- Logical
- Arithmetic
- Concatenation
- Parentheses to control order of operations
- Exponential operator (**)

The operations within an expression are done in a particular order depending on their precedence (priority). The following table shows the default order of operations from top to bottom:-

Operator	Operation
**, NOT	Exponentiation, logical navigation
+, -	Identity, negation
*, /	Multiplication, division
+, -, ||	Addition, subtraction, concatenation
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparison
AND	Conjunction
OR	Inclusion

Table 11.7

Using Bind Variables

To reference a bind variable in PL/SQL, prefix its name with a colon (:),

Example

```
VARIABLE            g_salary NUMBER
DECLARE
          v_sal     emp.sal%TYPE
BEGIN
```

```
          SELECT       sal
          INTO         v_sal
          FROM         emp
          WHERE empno = 7369;
          :g_salary     :=      v_sal;
END;
/
```

Printing Bind Variables

In SQL*Plus, we can display the value of the bind variable using the PRINT command

```
SQL> PRINT g_salary

G_SALARY
---------------
   800
```

Interacting with Oracle Server

It is necessary to use SQL to extract information from or apply changes to the database. PL/SQL supports full data manipulation language and transaction control commands within SQL. We can use SELECT statement to populate variables with values queried from a row in a table. However, the DML statements can process multiple rows.

SELECT statements in PL/SQL

Syntax is
SELECT *select_list*
INTO {*variable_name* [, *variable_name*] …}
FROM *table*
WHERE *condition*;
Note: Remember that host variables must be prefixed with a colon.

SQL Cursor

A cursor is a private SQL work area of memory in which the SQL statement is parsed and executed. When the executable part of a block issues a SQL statement, PL/SQL creates an implicit cursor, which has the SQL identifier. PL/SQL manages this cursor automatically. The programmer explicitly declares and names an explicit cursor. There are four attributes available in PL/SQL that can be applied to cursors.

SQL Cursor Attributes

Using SQL cursor attributes, we can test the outcome of our SQL statements.

SQL%ROWCOUNT	Number of rows affected by the most recent SQL statement (an integer value)
SQL%FOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows
SQL%NOTFOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement does not affect any rows
SQL%ISOPEN	Always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed.

Table 11.8

Example

Delete rows that have the specified order number from the ITEM table. Print the number of rows deleted.

```
VARIABLE     rows_deleted     VARCHAR2(30)
DECLARE
        v_ordid          NUMBER        :=        605;
BEGIN
        DELETE           FROM  item
        WHERE                  ordid     = v_ordid;
        :rows_deleted := (SQL % ROWCOUNT || ‘ rows deleted.’);
END;
/
PRINT rows_deleted
```

EXERCISES

1. Write down SQL*Plus commands as well as a complete PL/SQL block to input basic monthly salary and print net salary using DBMS_OUTPUT.PUT_LINE. Assume total allowances are 45% and deductions include taxes are 11%.

____

____

____

____

____

____

____

2. Build a PL/SQL block that computes the total compensation for one year. The annual salary and the annual bonus percentage are passed to the PL/SQL block through SQL*Plus substitution variables, and the bonus needs to be converted from a whole number to a decimal (for example, 15 to .15). If the salary is null, set it to zero before computing the total compensation. Execute the PL/SQL block. Attach screenshots of output. Reminder: Use the NVL function to handle null values.

```
Please enter the salary amount: 50000
Please enter the bonus percentage: 10

PL/SQL procedure successfully completed.

G_TOTAL
-------------
55000
```

3. Write a PL/SQL block to print following message on screen:-
   Number of Employees in department 10 = xxx
   Maximum salary in department 20 = xxx
   Average Salary of all employees = xxx

Lab Session 11

Use control structures and handling exceptions in PL/SQL

INTRODUCTION

The logical flow of statements can be changed within the PL/SQL block with a number of control structures. This session addresses two types of PL/SQL control structures: conditional constructs with the IF statement and LOOP control structures.

Conditional Constructs

There are three forms of IF statements:-

- IF-THEN-END IF
- IF-THEN-ELSE-END IF
- IF-THEN-ELSIF-END IF

Examples

- Set the manager ID to 22 if the employee name is Osborne.

```
IF v_ename = ‘OSBORNE’ THEN
        v_mgr := 22;
END IF;
```

- Set the job title to Salesman, the department number to 35, and the commission to 20% of the current salary if the last name is Miller.

```
…
IF v_ename = ‘MILLER’ THEN
        V_job := ‘SALESMAN’;
        V_deptno := 35;
        V_new_comm. := sal * 0.20;
END IF;
…
```

  Note: Complete the above program by entering employee name and print the three values.

- Determine an employee’s bonus based upon the department

```
…
IF v_deptno = 10 THEN
        v_bonus := 5000;
ELSIF v_deptno = 20 THEN
        v_bonus := 7500;
ELSE
        v_bonus := 2000;
END IF;
…
```

  Note: Complete the above program by entering department number and print bonus amount.

Iterative Control

Loops repeat a statement or sequence of statements multiple times.
There are three types of loops in PL/SQL:-

- *Basic loop* to provide repetitive actions without overall conditions
- *FOR loops* to provide iterative control of actions based on a count

- *WHILE loops* to provide iterative control of actions based on a count

Basic Loop

The basic loop encloses a sequence of statements between the keywords LOOP and END LOOP. Each time the flow of execution reaches the END LOOP statement, control is returned to the corresponding LOOP statement above it. A basic loop allows execution of its statement at least once; even if the condition is already met upon entering the loop. However, we can terminate the loop using the EXIT statement.

Example

Insert the first 10 new line items for order number 601.

```
DECLARE
        v_ordid item.ordid%TYPE := 601;
        v_counter       NUMBER(2) := 1;
BEGIN
        LOOP
                INSERT INTO item(ordid, itemid)
                VALUES (v_ordid, v_counter)
                V_counter := v_counter + 1;
                EXIT WHEN v_counter > 10;
        END LOOP;
END;
```

FOR Loop

Insert the first 10 new line items for order number 601.
Example

```
DECLARE
        V_ordid         item.ordid%TYPE := 601;
BEGIN
        FOR   I   IN   1..10   LOOP
                INSERT INTO item(ordid, itemid)
                VALUES (v_ordid, i);
        END LOOP;
END;
```

WHILE Loop

We use the WHILE loop to repeat a sequence of statements until the controlling condition is no longer TRUE.
Example

```
ACCEPT p_new_order PROMPT ‘Enter the order number: ‘
ACCEPT p_items PROMPT ‘Enter the number of items in this order: ‘
DECLARE
        v_countNUMBER(2) := 1;
BEGIN
        WHILE v_count <= &p_items LOOP
                INSERT INTO item (ordid, itemid)
                VALUES (&p_new_order, v_count);
                v_count := v_count + 1;
        END LOOP;
        COMMIT;
```

END;

Writing explicit cursors in PL/SQL

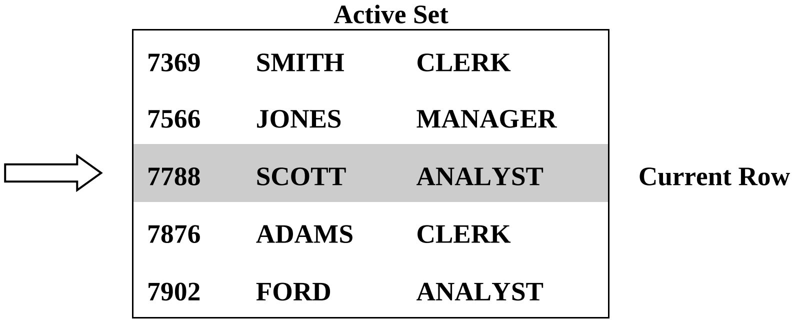
Oracle server uses work areas called *private SQL areas* to execute SQL statements and to store processing information. We can use PL/SQL cursors to name a private SQL area and access its stored information. The cursor directs all phases of processing.

Cursor Type	Description
Implicit	Implicit cursors are declared by PL/SQL implicitly for all DML and PL/SQL SELECT statements, including queries that return only one row.
Explicit	For queries that return more than one row. Explicit cursors are declared and named by the programmer and manipulated through specific statements in the block's executable actions.

Table 12.1

Explicit Cursor Functions

We use explicit cursors to individually process each row returned by a multiple-row SELECT statement. The set of rows returned by a multiple-row query is called the *active set*. Its size is the number of rows that meet the search criteria. The figure 12.1 below show how an explicit cursor points to the *current row* in the active set.



Active Set

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

Current Row

Figure 12.1: Explicit cursor containing the active set

A PL/SQL program opens a cursor, processes rows returned by a query and then closes the cursor. The cursor marks the current position in an active set.

Explicit cursor functions

- Can process beyond the first row returned by the query, row by row.
- Keep track of which row is currently being processed
- Allow the programmer to manually control them in the PL/SQ block

Controlling Explicit Cursors

i. *Declare the cursor* by naming it and defining the structure of the query to be performed within it.

ii. *Open the cursor*. The OPEN statement executes the query and binds any variables that are referenced. Rows identified by the query are called the *active set* and are now available for fetching.

iii. *Fetch data from the cursor*. The FETCH statement loads the current row from the cursor into variables. Each fetch causes the cursor to move its pointer to the next row in the active set. Therefore, each fetch accesses a different row returned by the query. In the flow diagram below, each fetch tests the cursor for any existing rows. If rows are found, the fetch loads the current row into variables; otherwise, it closes the cursor.

iv. *Close the cursor*. The CLOSE statement releases the active set of rows. It is now possible to reopen the cursor to establish a fresh active set.

We use the OPEN, FETCH and CLOSE statements to control a cursor.
The *OPEN* statement executes the query associated with the cursor, identifies the active set, and positions the cursor (pointer) before the first row. The *FETCH* statement retrieves the current row and advances the cursor to the next row. When the last row has been processed, the *CLOSE* statement disables the cursor.

Cursor FOR loop

A cursor FOR loop processes rows in an explicit cursor. It is a shortcut because the cursor is opened, rows are fetched once for each iteration in the loop, and the cursor is closed automatically when all the rows have been processed. The loop itself is terminated automatically at the end of the iteration where the last row was fetched.

Examples

i. To print employee number, name and salary of all employees having salary higher than average salary.

```
SET SERVEROUTPUT ON
DECLARE
  AVGSAL NUMBER;
  CURSOR  emp_cursor IS
     SELECT empno, ename, sal
     FROM emp;
  emp_record   emp_cursor%ROWTYPE;
BEGIN
  SELECT AVG(SAL) INTO AVGSAL FROM EMP;
  DBMS_OUTPUT.PUT_LINE('Average Salary : ' || AVGSAL);
  OPEN emp_cursor;
  LOOP
     FETCH  emp_cursor  INTO emp_record;
     EXIT   WHEN   emp_cursor%NOTFOUND;
     IF emp_record.sal > AVGSAL THEN
       DBMS_OUTPUT.PUT_LINE('Employee Number : ' || emp_record.empno);
       DBMS_OUTPUT.PUT_LINE('Employee Name : ' || emp_record.ename);
       DBMS_OUTPUT.PUT_LINE('Salary : ' || emp_record.sal);
     END IF;
  END LOOP;
  CLOSE emp_cursor; -- closes cursor
END;
```

ii. Retrieves employees one by one and print out a list of those employees currently working in the Sales department using cursor FOR loop.

```
SET SERVEROUTPUT ON
DECLARE
CURSOR emp_cursor IS
  SELECT ename, deptno
  FROM emp;
BEGIN
  FOR emp_record IN emp_cursor LOOP
            -- implicit open and implicit fetch occur
     IF emp_record.deptno = 30 THEN
        DBMS_OUTPUT.PUT_LINE('Employee ' || emp_record.ename ||
' works in the sales Dept.');
     END IF;
END LOOP; -- implicit close occurs
END;
```

iii. To retrieve the first 5 items for an order one by one. As each product is processed for the order, calculate the new total for the order and print it on the screen.

```
SET SERVEROUTPUT ON
DECLARE
  v_prodid   order_data.prodid%TYPE;
  v_item_total  NUMBER(11, 2);
  v_order_total NUMBER(11, 2) := 0;

  CURSOR item_cursor IS
    SELECT prodid, unitprice * quantity
    FROM order_data
    WHERE orderid = &p_ordid;
BEGIN
  OPEN item_cursor;
  LOOP
        FETCH item_cursor INTO v_prodid, v_item_total;
        EXIT WHEN item_cursor%ROWCOUNT > 5 OR
          item_cursor%NOTFOUND;
        v_order_total := v_order_total + v_item_total;
   DBMS_OUTPUT.PUT_LINE('Product number ' || TO_CHAR(v_prodid) ||
        'brings this order to a total of ' || TO_CHAR(v_order_total, '$999,999.99'));
  END LOOP;
CLOSE item_cursor;
END;
```

Handling Exceptions

An exception is an identifier in PL/SQL, raised during the execution of a block that terminates its main body of actions. A block always terminates when PL/SQL raises an exception, but we specify an exception handler to perform final actions.

Methods for raising exception

Two methods for raising an exception are as follows:-

- An Oracle error occurs and the associated exception is raised automatically. For example, if the error ORA-01403 occurs when no rows are retrieved from the database in a SELECT statement, then PL/SQL raises the exception NO_DATA_FOUND.
- An exception is raised explicitly by issuing the RAISE statement within the block. The exception being raised may be either user defined or predefined.

Handling Exceptions

Two methods for handling an exception are as follows:-

- Trap it with a handler
- Propagate it to the calling environment

Trapping an Exception

If the exception is raised in the executable section of the block, processing branches to the corresponding exception handler in the exception section of the block. If PL/SQL successfully handles the exception, then the exception does not propagate to the enclosing block or environment. The PL/SQL block terminates successfully.

Propagating an Exception

If the exception is raised in the executable section of the block and there is no corresponding exception handler, the PL/SQL block terminates with failure and the exception is propagated to the calling environment.

Exception Types

- **Predefined Oracle Server**: One of approximately 20 errors that occur most often in PL/SQL code. Do not declare and allow the Oracle Server to raise them explicitly.
- **Non-predefined Oracle Server error**: Any other standard Oracle Server error. Declare within the declarative section and allow the Oracle Server to raise them implicitly.
- **User-defined error**: A condition that the developer determines is abnormal. Declare within the declarative section and raise explicitly.

This session covers only the predefined exceptions.

Trapping Exceptions

Syntax is

```
EXCEPTION
WHEN exception1 [OR exception2 . . .] THEN
          Statement1;
          Statement2;
          . . .
[WHEN exception1 [OR exception2 . . .] THEN
          Statement1;
          Statement2;
          . . .]
[WHEN OTHERS THEN
          Statement1;
          Statement2;
          . . .]
```

The OTHERS handler traps *all* exceptions not already trapped.

Note: Place the OTHERS clause after all other exception-handling clauses. We can have at most one OTHERS clause. Exceptions cannot appear in assignment statements or SQL statements.

Trapping Predefined Oracle Server Errors

Trap a predefined Oracle Server error by referencing its standard name within the corresponding exception-handling routine. Some commonly used predefined exceptions are as follows:-

Exception Name	Oracle Server Error	Description
DUP_VAL_ON_INDEX	ORA-00001	Attempted to insert a duplicate value
INVALID_NUMBER	ORA-01722	Conversion of character string to number fails
NO_DATA_FOUND	ORA-01403	Single row SELECT returned no data
TIMEOUT_ON_RESOURCE	ORA-00051	Time-out occurred while Oracle is waiting for a resource
TOO_MANY_ROWS	ORA-01422	Single-row SELECT returned more than one row
VALUE_ERROR	ORA-06502	Arithmetic, conversion, truncation, or size-constraint

		error returned
ZERO_DIVIDE	ORA-01476	Attempted to divide by zero

Table 12.2

Note: It is a good idea to always consider the NO_DATA_FOUND and TOO_MANY_ROWS exceptions, which are the most common.

Example 1

To enter an employee number and find his or her name. If no such employee exists, raise an exception NO_DATA_FOUND to generate a message.

```
DECLARE
        v_empno  emp.empno%TYPE := &p_empno;
        v_ename emp.ename%TYPE;
BEGIN
        SELECT ename into v_ename
        FROM EMP
        WHERE empno = v_empno;
        DBMS_OUTPUT.PUT_LINE ('Employee Name : ' || v_ename);
EXCEPTION
        WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE ('No employee with number ' || v_empno || ' Found');
END;
```

Example 2

To enter an employee name and find his or her salary and name. If no such employee exists, raise an exception NO_DATA_FOUND to generate a message. If more than one employee exists with this name, raise an exception TOO_MANY_ROWS to generate a message.

```
DECLARE
        v_empno  emp.empno%TYPE;
        v_ename emp.ename%TYPE := '&p_ename';
BEGIN
        SELECT empno into v_empno
        FROM EMP
        WHERE ename = v_ename;
        DBMS_OUTPUT.PUT_LINE ('Employee Number : ' || v_empno);
EXCEPTION
        WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE ('No employee with name ' || v_ename || ' Found');
        WHEN TOO_MANY_ROWS THEN
                DBMS_OUTPUT.PUT_LINE ('Too many employees with name ' || v_ename || ' Found');
END;
```

EXERCISES

1. Create a PL/SQL block that computes and prints the bonus amount for a given employee based on the employee’s salary.
    a. Accept the employee number as user input with a SQL*Plus substitution variable.
    b. If the employee’s salary is less than 1,000, set the bonus amount for the employee to 10% of the salary.
    c. If the employee’s salary is between 1,000 and 1,500, set the bonus amount for the employee to 15% of the salary.

d. If the employee’s salary exceeds 1,500, set the bonus amount for the employee to 20% of the salary.
e. If the employee’s salary is NULL, set the bonus amount for the employee to 0.

2. Create a PL/SQL block that determines the name of top *n* employees with respect to salaries. Accept a number **n** as user input with a SQL*Plus substitution parameter. Then, in a loop, get the names and salaries of the top n people with respect to salary in the EMP table.

3. Suppose employees are given house rent 45%, conveyance 12% and medical 15% of basic pay (SAL column in EMP table). Taxes are 20% of gross pay. Write down a PL/SQL block to print the employee number, name and net salary of all employees in the EMP table as follows:-

   Employee Number : xxxx
   Employee Name: xxxx xxxxx xxxx
   Net Salary: Rs xxxxx

4. Write a PL/SQL block to select the name of the employee with a given salary value.
    a. If the salary entered returns more than one row, handle the exception with an appropriate exception handler and insert into the MESSAGES table the message “More than one employee with a salary of <*salary*>.”
    b. If the salary entered does not return any rows, handle the exception with an appropriate exception handler and insert into the MESSAGES table the message “No employee with a salary of <*salary*>.”
    c. If the salary entered returns only one row, insert into the MESSAGES table the employee’s name and the salary amount.
    d. Handle any other exception with an appropriate exception handler and insert into the MESSAGES table the message “Some other error occurred.”
    e. Test the block for a variety of test cases.

```
RESULTS
-------------
SMITH – 800
More than one employee with a salary of 3000
No employee with a salary of 6000
```

Lab Session 12

Explore Triggers in Database

Trigger

A *trigger* is a PL/SQL block that executes implicitly whenever a particular event takes place. A trigger can be either a database trigger or an application trigger.
Database triggers execute implicitly when an INSERT, UPDATE, or DELETE statement is issued against the associated table, no matter which user is connected or which application is used.
Application triggers execute implicitly whenever a particular event occurs within an application. An example of an application that uses triggers extensively is one developed with Developer/2000 Form Builder.
Note: Database triggers can be defined only on tables, not on views. However, if a DML operation is issued against a view, triggers on the base table(s) of a view are fired.

Guidelines for designing triggers

- Only use database triggers for centralized, global operations that should be fired for the triggering statement, regardless of which user or application issues the statement.
- Do not define triggers to implement integrity rules that can be done by using declarative constraints.
- The excessive use of triggers can result in complex interdependencies, which may be difficult to maintain in large applications. Only use triggers when necessary, and beware of recursive and cascading effects.

Database Trigger Types

The trigger type determines the number of times the trigger action is to be executed: once for every row affected by the triggering statement (such as a multiple row UPDATE), or once for the triggering statement no matter how many rows it affects.

Statement Trigger

A statement trigger is fired once on behalf of the triggering event, even if no rows are affected at all. Statement triggers are useful if the trigger action does not depend on data of rows that are affected or data provided by the triggering event itself. For example, a trigger that performs a complex security check on the current user.

Row Trigger

A Row trigger fires each time the table is affected by the triggering event. If the triggering event affects no row(s), a row trigger is not executed at all.
Row triggers are useful if the trigger action depends on data of rows that are affected or data provided by the triggering event itself.

Creating Statement Triggers

Syntax for creating Statement Triggers
CREATE [OR REPLACE] TRIGGER *trigger_name*
Timing event1 [OR *event2* OR *event3*]
ON *table_name*

PL/SQL block;

Trigger Components

Before coding the trigger block, decide on the components of the trigger:-
Trigger timing: BEFORE or AFTER
Triggering event: INSERT or UPDATE or DELETE
Table Name: ON table
Trigger Type: Row or Statement
Trigger body: DECLARE
BEGIN
END;

Trigger Timing

Indicates the time when the trigger fires in relation to the triggering event: BEFORE or AFTER.

BEFORE Triggers
This type of trigger is frequently used in the following situations:

- When the trigger action should determine whether that triggering statement should be allowed to complete. This allows to eliminate unnecessary processing of the triggering statement and its eventual rollback in cases where an exception is raised in the triggering action.
- To derive column values before completing a triggering INSERT or UPDATE statement.

AFTER Triggers
This type of trigger is frequently used in the following situations:

- When the triggering statement is to be completed before executing the triggering action.
- If a BEFORE trigger is already present, and an after trigger can perform different actions on the same triggering statement.

Triggering Event

The triggering event or statement can be an INSERT, UPDATE, or DELETE statement on a table.

- When the triggering event is an UPDATE, we can include a column list to identify which column(s) must be changed to fire the trigger. We cannot specify a column list for an INSERT or for a DELETE statement, as they always affect entire rows.
- The triggering event can contain multiple DML statements. In this way, we can differentiate what code to execute depending on the statement that caused the triggers to fire.

Trigger Body

The trigger action defines what needs to be done when the triggering event is issued. It can contain SQL and PL/SQL statements, define PL/SQL constructs such as variables, cursors, exceptions and so on.
Additionally row triggers have access to the old and new column values of the row being processed by the trigger, using correlation names.
The trigger body is defined with an anonymous PL/SQL block.

```
[DECLARE]
BEGIN
[EXCEPTION]
END;
```

Before Statement Trigger: Example

We can create a *BEFORE statement* trigger in order to prevent the triggering operation from succeeding if a certain condition is violated.

For example, create a trigger to restrict inserts into the EMP table to certain business hours on Monday through Friday. If a user attempted to insert a row into the EMP table on Saturday, for example, the user will see the message, the trigger will fail, and the triggering statement will be rolled back.

RAISE_APPLICATION_ERROR is a server-side built-in procedure that prints a message to the user and causes the PL/SQL block to fail. When a database trigger fails, the triggering statement is automatically rolled back by the Oracle Server.

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT ON emp
BEGIN
    IF (TO_CHAR(sysdate, 'DY') IN ('SAT', 'SUN'))
    OR (TO_CHAR(sysdate, 'HH24') NOT BETWEEN '08' AND '18')
    THEN RAISE_APPLICATION_ERROR (-20000, 'You may only insert
            into EMP during normal hours.');
    END IF;
END;
```

Using Conditional Predicates

We can combine several triggering events into one by taking advantage of the special conditional predicates INSERTING, UPDATING, and DELETING within the trigger body.

Use BEFORE statement triggers to initialize global variables or flags, and to validate complex business rules.

```
CREATE OR REPLACE TRIGGER secure_emp
BEFORE INSERT OR UPDATE OR DELETE ON emp
BEGIN
    IF (TO_CHAR(sysdate, 'DY') IN ('SAT', 'SUN'))
    OR (TO_CHAR(sysdate, 'HH24') NOT BETWEEN '08' AND '18')     THEN
            IF DELETING THEN
            RAISE_APPLICATION_ERROR (-20502, 'You may only delete from EMP during normal hours.');
            ELSIF INSERTING THEN
            RAISE_APPLICATION_ERROR (-20500, 'You may only insert into EMP during normal hours.');
            ELSIF UPDATING(‘SAL’) THEN
            RAISE_APPLICATION_ERROR (-20503, 'You may only update SAL during normal hours.');
            ELSE
            RAISE_APPLICATION_ERROR (-20504, 'You may only update EMP during normal hours.');
    END IF;
    END IF;
END;
```

After Statement Trigger: Example

We can create an *AFTER Statement* trigger in order to audit the triggering operation or perform a calculation after an operation has completed.

Suppose we have a user defined audit table that lists users and counts their data manipulation operations. After any user has updated the SAL column in the EMP table, use the audit table to ensure that the number of salary changes does not exceed the maximum permitted for that user.

User Audit Table

USER_ NAME	TABLE_ NAME	COLUMN_ NAME	INS	UPD	DEL	MAX_INS	MAX_UPD	MAX_DEL
SCOTT	EMP		1	1	1	5	5	5
SCOTT	EMP	SAL		1			5	
SCOTT	EMP		0	0	0	5	0	0

Figure 13.1

```
CREATE OR REPLACE TRIGGER check_salary_count
AFTER UPDATE OF sal ON emp
DECLARE
  v_salary_changes  NUMBER;
  v_max_changes     NUMBER;
BEGIN
SELECT upd, max_upd
INTO v_salary_changes, v_max_changes
FROM audit_table
WHERE user_name = user
AND tablename = 'EMP'
AND column_name = 'SAL';
IF v_salary_changes > v_max_changes THEN
        RAISE_APPLICATION_ERROR (-20501, 'You may only make a maximum of '          ||
to_char(v_max_changes) || ' changes to the sal column');
END IF;
END;
```

Creating Row Triggers

Syntax for creating Row Triggers
CREATE [OR REPLACE] TRIGGER *trigger_name*
Timing event1 [OR *event2* OR *event3*]
ON *table_name*
FOR EACH ROW
[WHEN *condition*]
PL/SQL block;

This syntax is identical to the syntax for creating statement triggers except following:-
FOR EACH ROW: Designates the trigger to be a row trigger
WHEN: Specifies the trigger restriction (This conditional predicate is evaluated for each row to determine whether or not the trigger body is executed.)

After Row Trigger: Example

A row trigger can be created to keep a running count of data manipulation operations by different users on database tables. If a trigger routine does not have to take place before the triggering operation, create an AFTER row trigger rather than a BEFORE row trigger.

```
CREATE OR REPLACE TRIGGER audit_emp
AFTER DELETE OR INSERT OR UPDATE ON emp
FOR EACH ROW
BEGIN
IF DELETING THEN
        UPDATE        audit_table        SET     del = del + 1
      WHERE user_name = user AND table_name = ‘EMP’
      AND             column_name IS NULL;
ELSIF INSERTING THEN
      UPDATE          audit_table     SET     ins = ins + 1
      WHERE           user_name = user AND table_name = ‘EMP’
      AND             column_name IS NULL;
ELSIF UPDATING(‘SAL’) THEN
      UPDATE          audit_table     SET     upd = upd + 1
      WHERE           user_name = user AND table_name = ‘EMP’
      AND             column_name = ‘SAL’;
ELSE
      UPDATE          audit_table     SET     upd = upd + 1
      WHERE           user_name = user AND table_name = ‘EMP’
      AND             column_name IS NULL;
END IF;
END;
```

Using Old and New Qualifiers

We can create a trigger on the EMP table to add rows to a user table, AUDIT_EMP_VALUES, logging a user’s activity against the EMP table. The trigger records the values of several columns both before and after the data changes by using the OLD and NEW qualifiers with the respective column name.

```
CREATE OR REPLACE TRIGGER audit_emp_values
AFTER DELETE OR INSERT OR UPDATE ON emp
FOR EACH ROW
BEGIN
INSERT INTO audit_emp_values (user_name, timestamp, id, old_last_name, old_title,
New_title, old_salary, new_salary)
VALUES (USER, SYSDATE, :old.empno, :old.ename, :new.ename, :old.job, :new.job, :old.sal, :new.sal);
END;
```

Before Row Trigger: Example

To restrict the trigger action to those rows that satisfy a certain condition, provide a WHEN clause. Create a trigger on the EMP table to calculate an employee’s commission when a row is added to the EMP table or an employee’s salary is modified.
The NEW qualifier does not need to be prefixed with a colon in the WHEN clause.

```
CREATE OR REPLACE TRIGGER derive_commission_pct
BEFORE INSERT OR UPDATE OF SAL ON emp
FOR EACH ROW
WHEN (new.job = ‘SALESMAN’)
BEGIN
IF INSERTING THEN       :new.comm. := 0;
ELSE                /* UPDATE of salary */
        IF :old.comm. IS NULL THEN
                :new.comm. := 0;
        ELSE
                :new.comm. := :old.comm. + (:new.sal / :old.sal);
        END IF;
END IF;
END;
```

Difference between Triggers and Stored Procedures

Database Triggers	**Stored Procedure**
Use CREATE TRIGGER	Use CREATE PROCEDURE
Data dictionary contains source and p-code	Data dictionary contains source and p-code
Implicitly invoked	Explicitly invoked
COMMIT, ROLLBACK And SAVEPOINT statements are not allowed within the trigger body	COMMIT, ROLLBACK And SAVEPOINT statements are permitted within the procedure body

Figure 13.2

Triggers are fully compiled when the CREATE TRIGGER command is issued, and the p-code is stored in the Data Dictionary. Hence, firing the trigger no longer requires the opening of a shared cursor to run the trigger action. Instead, the trigger is executed directly. If error occurs during the compilation of a trigger, the trigger is still created.

Disable/Enable a database trigger

ALTER TRIGGER *trigger_name DISABLE* | *ENABLE*;

Removing a Trigger

DROP TIGGER *trigger_name*;

EXERCISE

1. What are triggers? Differentiate between database triggers and row triggers.

2. Differentiate between Statement and Row triggers?

3. What is meant by triggering event? Give examples.

4. Give three examples of a situation when a BEFORE statement trigger is needed?

5. Give three examples of a situation when an AFTER statement trigger is needed?

6. A number of business rules apply to the EMP and DEPT tables. Some of which are as follows:-
   i. Sales persons should always receive commission. Employees who are not sales persons should never receive a commission.
   ii. The EMP table should contain exactly one PRESIDENT. Test your answer.
   iii. An employee should never be manager of more than five employees. Test your answer.
   iv. Salaries may only be increased, never decreased. Test your answer.
   v. If a department moves to another location, each employee of that department automatically receives a salary raise of 2%.

Now implement the above business rules with the help of database triggers.

Lab Session 13

Explore stored procedures and stored functions in Database

INTRODUCTION

Stored procedure is a named PL/SQL block that can take parameters and be invoked. Generally speaking, a procedure is used to perform an action. A procedure has a header, a declarative part, an executable part, and an optional exception-handling part. Procedures promote reusability and maintainability. Once validated, they can be used in any number of applications. If the definition changes, only the procedure is affected, this greatly simplifies maintenance.

Syntax for Creating Procedures

We create new procedures with the CREATE PROCEDURE statement, which may declare a list of arguments (sometimes referred to as parameters), and must define the actions to be performed by the standard PL/SQL block.

```
CREATE [OR REPLACE] PROCEDURE procedure_name
          (argument1  [mode1]  datatype1,
           argument2  [mode2]  datatype2,
          …..
IS [AS]
PL/SQL Block;
```

Syntax Definitions

Parameter	**Description**
Procedure_name	Name of the procedure
Argument	Name of a PL/SQL variable whose value is passed to, populated by the calling environment, or both, depending, on the *mode* being used
Mode	Type of argument IN (default) OUT IN OUT
Datatype	Datatype of the argument
PL/SQL block	Procedural body that defines the action performed by the procedure

Table 14.1

- PL/SQL block starts with either BEGIN or the declaration of local variables and end with either END or END procedure name. We cannot reference host or bind variables in the PL/SQL block of a stored procedure.
- The REPLACE option indicates that if the procedure exists, it will be dropped and replaced with the new version created by the statement.

Creating a Stored Procedure using SQL*Plus

- Enter the text of the CREATE PROCEDURE statement in a system editor or word processor and save it as a script file (*.sql extension*)
- From SQL*Plus, run the script file to compile the source code into p-code and store both in the database.
- Invoke the procedure from an oracle server environment to determine whether it executes without error.

Procedural Parameter Modes

We can transfer values to and from the calling environment through parameters. Choose one of the following three modes for each parameter: IN, OUT, or IN OUT. Attempts to change the value of an IN parameter will result in an error.

DATATYPE can only be the %TYPE definition, %ROWTYPE definition, or an explicit datatype with no size specification.

Type of Parameter	Description
IN (default)	Passes a constant value from the calling environment into the procedure
OUT	Passes a value from the procedure to the calling environment
IN OUT	Passes a value from the calling environment into the procedure and a possibly different value from the procedure back to the calling environment using the same parameter.

Table 14.2

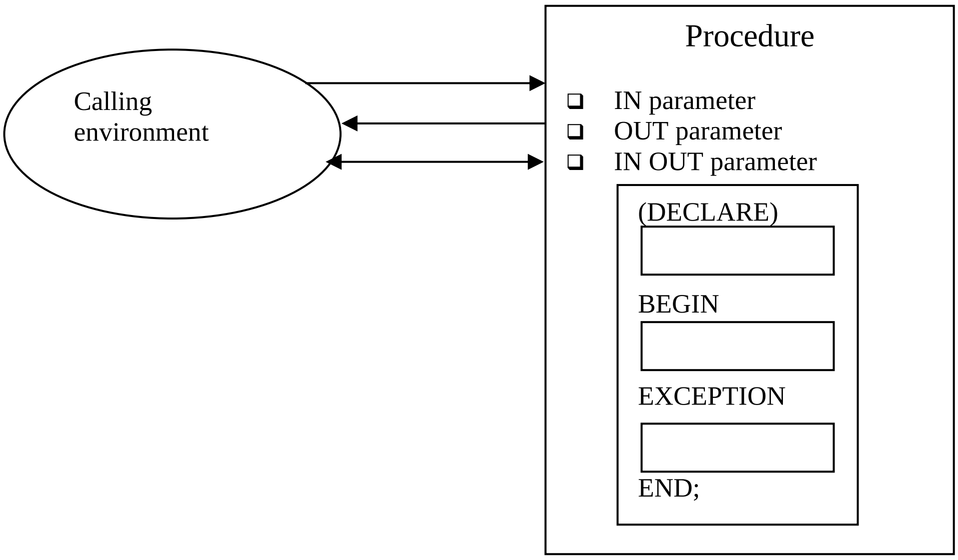


Figure 14.1

Parameter Modes for Formal Parameters

IN	OUT	IN OUT
Default	Must be specified	Must be specified
Value is passed into subprogram	Returned to calling environment	Passed into subprogram; returned to calling environment
Formal parameter acts as a constant	Uninitialized variable	Initialized variable
Actual parameter can be a literal, expression, constant or initialized variable	Must be a variable	Must be a variable

Table 14.3

IN Parameters

The example below shows a procedure with one IN parameter. Running this statement in SQL*Plus creates the RAISE_SALARY procedure. When invoked, RAISE_SALARY takes the parameter for the employee number and updates the employee’s record with a salary increase of 10 percent.

To invoked a procedure in SQL*Plus, use the EXECUTE command.

```
SQL>CREATE OR REPLACE PROCEDURE raise_salary
2         (v_id in emp.empno%TYPE)
3         IS
4         BEGIN
5                   UPDATE        emp
6                   SET           sal = sal *1.10
7                   WHERE         empno = v_id;
8         END       raise_salary;
9         /
Procedure created

SQL> EXECUTE raise_salary (7369)
PL/SQL procedure successfully completed
```

OUT Parameters

The example below shows a procedure, QUERY_EMP, with one IN and three OUT parameters that will return a value to the calling environment. Running this statement in SQL*Plus creates the RAISE_SALARY procedure.

```
SQL>CREATE OR REPLACE PROCEDURE query_emp
2         (v_id IN emp.empno%TYPE,
3         v_name OUT emp.ename%TYPE,
4         v_salary OUT emp.sal%TYPE,
5         v_comm OUT emp.comm%TYPE)
6         IS
7         BEGIN
8         SELECT ename, sal, comm.
9         INTO v_name, v_salary, v_comm.
10        FROM emp
11        WHERE empno = v_id;
10        END       query_emp;
11        /
Procedure created
```

View the value of OUT parameters with SQL*Plus

- Create host variables in SQL*Plus using the VARIABLE syntax.
- Invoke the QUERY_EMP procedure, supplying these host variables as the OUT parameters. Note the use of the colon (:) to reference the host variables in the EXECUTE syntax.
- To view the values passed from the procedure to the calling environment, use the PRINT syntax. Only one variable can be supplied to each PRINT command.

```
SQL> VARIABLE g_name varchar2(15)
SQL> VARIABLE g_salary number
SQL> VARIABLE g_comm number
```

```
SQL> EXECUTE query_emp (7654, :g_name, :g_salary,
:g_comm.)
PL/SQL procedure successfully completed
```

```
SQL> PRINT g_name
G_NAME
--------------------------
MARTIN
```

```
CREATE OR REPLACE PROCEDURE format_phone
(v_phone_no IN OUT VARCHAR2)
IS
BEGIN
   v_phone_no := '(' || SUBSTR(v_phone_no, 1, 3) ||
                 ')' || SUBSTR(v_phone_no, 4, 3) ||
                 '-' || SUBSTR(v_phone_no, 7);
END      format_phone;
/
```

Invoking FORMAT_PHONE from SQL*Plus

```
SQL> VARIABLE g_phone_no varchar2(15)
SQL> BEGIN  :g_phone_no := ‘9125421271’;  END;
2         /
PL/SQL procedure successfully completed.
SQL> EXECUTE format_phone (:g_phone_no)
PL/SQL procedure successfully completed.
SQL> PRINT g_phone_no
G_PHONE_NO
--------------------------
(912)542-1271
```

Viewing IN OUT Parameters with SQL*Plus

1. Create a host variable using the VARIABLE syntax.
2. Populate the host variable with a value, using an anonymous PL/SQL block.
3. Invoke the FORMAT_PHONE procedure supplying the host variable as the IN OUT parameter. Note the use of the colon (:) to reference the host variable in the EXECUTE syntax.
4. To view the value passed back to the calling environment, use the PRINT syntax.

Stored Functions

Stored function is a named PL/SQL block that can take parameters and be invoked. It is used to compute a value. Functions and procedures are structured alike, except that a function must return a value to the calling environment.

Functions promote reusability and maintainability. Once validated, they can be used in any number of applications. If the definition changes, only the function is affected, this greatly simplifies maintenance. Functions can be called as part of a SQL expression or as part of a PL/SQL expression.

Syntax for Creating Functions

We create new functions with the CREATE FUNCTION statement, which may declare a list of arguments (sometimes referred to as parameters), and must define the actions to be performed by the standard PL/SQL block.

```
CREATE [OR REPLACE] FUNCTION  function_name
            (argument1  [mode1]  datatype1,
             argument2  [mode2]  datatype2,
            …..
RETURN datatype
IS | AS
PL/SQL Block;
```

Syntax Definitions

Parameter	**Description**
Function_name	Name of the function
Argument	Name of a PL/SQL variable whose value is passed to, populated by the calling environment, or both, depending, on the *mode* being used
Mode	The type of the parameter; only IN parameters should be declared
Datatype	Datatype of the argument
RETURN datatype	Datatype of the RETURN value that must be output by the function.
PL/SQL block	Procedural body that defines the action performed by the procedure

Table 14.4

Creating a Stored function using SQL*Plus

- Enter the text of the CREATE FUNCTION statement in a system editor or word processor and save it as a script file (*.sql extension*)
- From SQL*Plus, run the script file to compile the source code into p-code and store both in the database.
- Invoke the function from an oracle server environment to determine whether it executes without error.

Example 1

To find the tax rate given the salary of an employee as follows:-

```
CREATE OR REPLACE FUNCTION GET_TAX_RATE (V_SAL  IN   NUMBER)
RETURN NUMBER
IS
        V_TAX_RATE NUMBER;
BEGIN
    IF V_SAL <= 1500 THEN
            V_TAX_RATE := 0;
    ELSIF V_SAL <= 3000 THEN
            V_TAX_RATE := 8;
    ELSIF V_SAL <= 4500 THEN
            V_TAX_RATE := 15;
    ELSE
            V_TAX_RATE := 20;
    END IF;
    RETURN V_TAX_RATE;
END GET_TAX_RATE;
/
SQL> SELECT GET_TAX_RATE(2345)
FROM DUAL;
PL/SQL procedure successfully completed
```

Example 2

To find the net salary given the number of an employee as follows:-

```
CREATE OR REPLACE FUNCTION GET_NET_SAL (V_EMPNO IN NUMBER)
  RETURN NUMBER
  IS
    V_SAL   EMP.SAL%TYPE; -- To store basic salary
    V_ALLOWANCE NUMBER := 45; -- Percentage of allowance
    V_GROSS NUMBER; -- To store gross salary
    V_NET NUMBER; -- To store net salary
  BEGIN
    SELECT SAL INTO V_SAL
    FROM EMP
    WHERE EMPNO = V_EMPNO;
    V_GROSS := V_SAL + V_SAL * V_ALLOWANCE /100;
```

```
   V_NET := V_GROSS - V_GROSS * GET_TAX_RATE(V_SAL)/100;
   RETURN V_NET;
END;
```

Locations to call user-defined functions

- Select list of a SELECT statement.
- Condition of the WHERE and HAVING clauses
- CONNECT BY, START WITH, ORDER BY, and GROUP BY clauses
- VALUES clauses of the INSERT statement
- SET clause of the UPDATE statement

Removing a stored function
The syntax is
DROP FUNCTION function_name;
e.g. SQL> DROP FUNCTION GET_TAX_RATE;

EXERCISES

1. Describe the different types of parameters used in procedures.

____

____

____

____

____

____

2. Write down the different modes for IN and IN OUT parameters.

____

____

____

____

____

____

3. Create a stored procedure to get the project ID and print the project title, client name, duration (in days) and status.

4. Create a stored procedure that takes the employee number and designation. Change the designation of the employee to the new value passed and then print the new grade as follows:-
New Grade: xxx

5. Create a stored procedure that has two arguments *TID* and *duration*. The first argument takes training code for a training program from the calling environment and the second argument should return the duration of training in weeks to the calling environment. Then print the duration in the calling environment.

6. Differentiate between procedures and functions.

____

____

____

____

____

____

____

____

7. What are the advantages of defining stored functions in databases?

____

____

____

____

____

____

8. Suppose employee performance in projects is quantified as Excellent=4, Good=3, Fair=2, Bad=1, Poor=0 (referred to lab session 6). Write a stored function to pass the employee number and get the employee *total-grade* on project performance.

Lab Session 14

Complex Engineering Activity

PROBLEM STATEMENT

COMPLEX PROBLEM SOLVING ATTRIBUTES COVERED

TASK DESCRIPTION

DELIVERABLES

GRADING RUBRIC

SOLUTION

NED University of Engineering & Technology
Department of Computer and Information Systems Engineering

Course Code and Title: CS-222 Database Management Systems

Laboratory Session No. ______________ Date: ___________ __________

Software Use Rubric				
Skill Sets	Extent of Achievement			
	0	1	2	3
To what extent has the student implemented the solution?	The solution has not been implemented.	The solution has syntactic and logical errors.	The solution has syntactic or logical errors.	The solution is syntactically and logically sound for the stated problem parameters.
How efficient is the proposed solution?	The solution does not address the problem adequately.	The solution exhibits redundancy and partially covers the problem.	The solution exhibits redundancy or partially covers the problem.	The solution is free of redundancy and covers all aspects of the problem.
How did the student answer questions relevant to the solution?	The student answered none of the questions.	The student answered less than half of the questions.	The student answered more than half but not all of the questions.	The student answered all the questions.
To what extent is the student familiar with the scripting/ programming interface?	The student is unfamiliar with the interface.	The student is familiar with few features of the interface.	The student is familiar with many features of the interface.	The student is proficient with the interface.

Weighted CLO Score	
Remarks	
Instructor’s Signature with Date	

NED University of Engineering & Technology
Department of Computer and Information Systems Engineering

Course Code and Title: CS-222 Database Management Systems

Laboratory Session No. ______________ Date: _____________________

Software Use Rubric				
Skill Sets	Extent of Achievement			
	0	1	2	3
To what extent has the student implemented the solution?	The solution has not been implemented.	The solution has syntactic and logical errors.	The solution has syntactic or logical errors.	The solution is syntactically and logically sound for the stated problem parameters.
How efficient is the proposed solution?	The solution does not address the problem adequately.	The solution exhibits redundancy and partially covers the problem.	The solution exhibits redundancy or partially covers the problem.	The solution is free of redundancy and covers all aspects of the problem.
How did the student answer questions relevant to the solution?	The student answered none of the questions.	The student answered less than half of the questions.	The student answered more than half but not all of the questions.	The student answered all the questions.
To what extent is the student familiar with the scripting/ programming interface?	The student is unfamiliar with the interface.	The student is familiar with few features of the interface.	The student is familiar with many features of the interface.	The student is proficient with the interface.

Weighted CLO Score	
Remarks	
Instructor’s Signature with Date	

NED University of Engineering & Technology
Department of Computer and Information Systems Engineering

Course Code and Title: CS-222 Database Management Systems

Laboratory Session No. _____________ Date: ___________ __________

Software Use Rubric

Skill Sets	Extent of Achievement			
	0	1	2	3
To what extent has the student implemented the solution?	The solution has not been implemented.	The solution has syntactic and logical errors.	The solution has syntactic or logical errors.	The solution is syntactically and logically sound for the stated problem parameters.
How efficient is the proposed solution?	The solution does not address the problem adequately.	The solution exhibits redundancy and partially covers the problem.	The solution exhibits redundancy or partially covers the problem.	The solution is free of redundancy and covers all aspects of the problem.
How did the student answer questions relevant to the solution?	The student answered none of the questions.	The student answered less than half of the questions.	The student answered more than half but not all of the questions.	The student answered all the questions.
To what extent is the student familiar with the scripting/ programming interface?	The student is unfamiliar with the interface.	The student is familiar with few features of the interface.	The student is familiar with many features of the interface.	The student is proficient with the interface.

Weighted CLO Score	
Remarks	
Instructor’s Signature with Date	