INTRODUCTION

The course *Database Management Systems*, taught in T.E. (Computer and Information Systems Engineering), intends to teach 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. SQL may also be used to implement security controls in a database system by assigning privileges to database users over database objects.

In order to supplement the above-mentioned capabilities of SQL with procedural units of code, control structures and blocks, Oracle provides a Procedural Language extension to SQL called PL/SQL. 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. It allows the data manipulation and query statement of SQL to be included in block-structured and procedural units of code, making PL/SQL a powerful transaction processing language.

This workbook comprehensively covers programming in SQL as well as PL/SQL. This second edition covers more features of SQL programming including joining operations and compound queries. The SQL’s ISO standard has also been covered in more detail. The number of examples and command variations has been increased. A new lab session covering the generation of readable output using SQL*Plus tool of Oracle has been added. More exercises have been included to give students the chance to improve their concepts and skills in SQL programming.

First ten lab sessions cover SQL programming focusing on relational database concepts, retrieval operations, Data Manipulation Language, Data Definition Language & Data Control Language. Remaining portion covers PL/SQL programming focusing data types, operators, block structures, control structures & handling exceptions. Advanced programming concepts such as database triggers and stored procedures/functions are covered in three appendices.

* * * * *
## CONTENTS

<table>
<thead>
<tr>
<th>Lab Session No.</th>
<th>Object</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overview of the features of SQL and SQL*Plus</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Basic data retrieval operations in SQL</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Data retrieval operations in SQL using join operations</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>Single-row and multiple-row functions in SQL</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>Sub-queries and compound queries in SQL</td>
<td>30</td>
</tr>
<tr>
<td>6</td>
<td>Creating readable output with SQL*Plus</td>
<td>37</td>
</tr>
<tr>
<td>7</td>
<td>Data manipulation operations in SQL</td>
<td>42</td>
</tr>
<tr>
<td>8</td>
<td>Creating and managing tables and views</td>
<td>48</td>
</tr>
<tr>
<td>9</td>
<td>Creating sequences, indexes and synonyms</td>
<td>61</td>
</tr>
<tr>
<td>10</td>
<td>Managing profiles and controlling user access</td>
<td>66</td>
</tr>
<tr>
<td>11</td>
<td>Basic features of PL/SQL Programming</td>
<td>76</td>
</tr>
<tr>
<td>12</td>
<td>Writing executable statements in PL/SQL</td>
<td>86</td>
</tr>
<tr>
<td>13</td>
<td>Using control structures in PL/SQL</td>
<td>95</td>
</tr>
<tr>
<td>14</td>
<td>Writing explicit cursors in PL/SQL</td>
<td>99</td>
</tr>
<tr>
<td>15</td>
<td>Handling exceptions in PL/SQL</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td>Appendix A - Database Triggers</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td>Appendix B – Stored Procedures</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td>Appendix C - Stored Function</td>
<td>123</td>
</tr>
</tbody>
</table>
Lab Session 01

OBJECT

Overview of the features of SQL and SQL*Plus

THEORY

What is a database?

Every organization needs information for making effective decisions. A library keeps a list of books, members, due dates and fines. A company needs to save information about employees, departments and salaries. The collection of facts about different entities in an organization is called data.

A database is an organized collection of interrelated data. In order to manage databases, we need Database Management Systems (DBMS). A DBMS is a program that stores, retrieves and modifies data in the database on user’s request as illustrated in figure 1.1 below.

![Figure 1.1: Users access data in a database through DBMS](image)

**Relational Database Concept**

A relational database uses relations or two-dimensional tables to store data. Such a database can be accessed and modified by executing Structured Query Language (SQL) statements. Dr. E. F. Codd proposed the relational model for database systems in 1970. Relational Database Management Systems (RDBMS) soon became very popular, especially for their ease of use and flexibility in structure. In addition, a number of innovative vendors such as Oracle, supplemented the RDBMS with a suite of powerful application development and user products, providing a total solution. A table is the basic storage structure of an RDBMS. A table holds all the data necessary about something in the real world – for example, employees, invoices, or customers.
Oracle 7 is a relational database management system whereas Oracle 8 and Oracle 9i are object relational database management systems. Oracle 9i includes everything needed to develop, deploy and manage internet applications. 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.

**Figure 1.2:** Databases allow compact and efficient storage of data

The table 1.1 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, which allows no duplicate rows. In table 1.1, EMPNO is the primary key where as in table 1.2, DEPTNO is the 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 in it.
- A field may have no value in it. This is called null value. In the EMP table, only employees who have a role of salesman have a value in the COMM (Commission) field.

**EMP**

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

**Table 1.1**
The table DEPT below contains data about different departments in an organization and SALGRADE gives details of salaries for various grades. The table JOB_HISTORY stores data about the job history of employees.

DEPT

<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DNAME</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>NEW YORK</td>
</tr>
<tr>
<td>20</td>
<td>RESEARCH</td>
<td>DALLAS</td>
</tr>
<tr>
<td>30</td>
<td>SALES</td>
<td>CHICAGO</td>
</tr>
<tr>
<td>40</td>
<td>OPERATIONS</td>
<td>BOSTON</td>
</tr>
</tbody>
</table>

Table 1.2

SALGRADE

<table>
<thead>
<tr>
<th>GRADE</th>
<th>LOSAL</th>
<th>HISAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>700</td>
<td>1200</td>
</tr>
<tr>
<td>2</td>
<td>1201</td>
<td>1400</td>
</tr>
<tr>
<td>3</td>
<td>1401</td>
<td>2000</td>
</tr>
<tr>
<td>4</td>
<td>2001</td>
<td>3000</td>
</tr>
<tr>
<td>5</td>
<td>3001</td>
<td>9999</td>
</tr>
</tbody>
</table>

Table 1.3

JOB_HISTORY

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

Table 1.4

Relating Multiple Tables

Each table contains data that describes exactly one entity. For example, the EMP table contains information about employees. Since data about different entities is stored in different tables, it may be needed to combine two or more tables to answer a particular question. For example, someone may want to know the location of the department where an employee works. In this scenario, information is needed from both the EMP and DEPT table. An RDBMS enables to relate the data in one table to the data in another table by using the foreign keys.

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. As a result, it doesn’t provide traditional programming structures. Instead, it is a language for specifying the operations at an unusually high level. The details of the implementation are left to the DBMS.

In Oracle, SQL is a language for communication with the Oracle Server from any tool or application. Oracle SQL has many extensions. Although we refer to the SQL language as a query language, it contains many other capabilities besides querying a database. It includes features for defining the structure of the data, for inserting and modifying data in the database, and for specifying security constraints.

It has following advantages:-
- Efficient
- Easy to learn and use
- Functionally complete (SQL allows to define, retrieve, and manipulate data in the tables.)

**Background of SQL**

The SQL language, originally called SEQUEL, was developed by IBM as part of the extensive work on the relational model in 1970s at their San Jose Research Laboratory, California. In order to test the viability of implementing the relational model in a DBMS, workers at the San Jose Research Laboratory undertook the project of the development of a prototype RDBMS named System R. This project took place from 1974 to 1979. The System R project led to the development of SEQUEL (Structured English Query Language) which was later renamed as SQL. Because System R was well received at the user sites where it was installed, other vendors began developing relational products that used SQL. In the late 1970s, the database management system Oracle was produced by what is now called the Oracle Corporation and was probably the first commercial implementation of a relational DBMS based on SQL. ORACLE is now available in mainframe, client-server and PC-based platforms for many operating systems including DOS, OS/2, various UNIX operating systems, Windows, VAX/VMS and MVS. Numerous RDBMS products now support the SQL language.

In 1986, the American National Standards Institute (ANSI) and the International Standards Organization (ISO) published an SQL standard, called SQL-86. IBM published its own corporate SQL standard, the Systems Application Architecture Database Interface (SAA-SQL) in 1987. An extended standard for SQL, SQL-89, was published in 1989. The ISO and ANSI committees created SQL-92 which was a more extensive expansion of SQL-86. SQL-92 which was in fact a first major revision of the SQL standard is sometimes referred to as SQL2. It was not until 1999 that the next release of the standard was formalized, commonly referred to as SQL3. This release contains additional features that support object-oriented data management.

**Scope**

SQL is used for all types of database activities by all types of users including:
- System administrators
- Database administrators
- Security administrators
- Application programmers
- Decision support system personnel
- Many other types of end users
Language Components

The SQL language has several parts:

**Data Retrieval**: The SQL includes a query language based on both the relational algebra and the tuple relational calculus.

**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.

**Embedded DML**: The embedded form of SQL is designed for use within general purpose programming languages, such as Cobol, Pascal and C.

**View definition**: The SQL DDL includes commands for defining views.

**Authorization**: The SQL DDL includes commands for specifying access rights for relations and views.

**Integrity**: The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.

**Transaction control**: SQL includes commands for specifying the beginning and ending of transaction. Several implementations also allow explicit locking of data for concurrency control.

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**

i. Selecting all columns from a table
```
SELECT  *
FROM DEPT;
```

ii. 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

iii. To select all employees whose salary is greater than 2200.
```
SELECT *
FROM EMP
WHERE SAL > 2200;
```

iv. 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';
```

v. 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;
vi. 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;

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 as well as SQL input from files. It provides a line editor for modifying SQL statements and formats query results into a basic report. The SQL*Plus environment is shown below in figure 1.3.

![SQL*Plus Environment](image)

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 where as SQL statements cannot.

A few commands of SQL*Plus are as follows:-

- **DESCRIBE**: To display the structure of a table e.g. SQL> DESCRIBE EMP
- **SAVE 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
- **SPOOL filename.ext**: 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
- **EDIT**: Invokes the editor and saves the buffer contents to a file named afiedt.buf
- **EDIT [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 1.4.

Figure 1.4

EXERCISES

1. What are relational databases? Describe, with the help of an example, how an RDBMS enables to relate data in one table to the data in another table.

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

2. Why SQL is called a nonprocedural language?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

3. Write down a brief history of SQL and explain why its standardization was necessary.

________________________________________________________________________
________________________________________________________________________
4. Write down the different features of SQL*Plus. How does SQL*Plus commands differ from SQL statements.

5. Describe SQL*Plus commands to move data to SQL buffer from file and move data to file from SQL buffer.
6. Write a query to display the employee number, name and salary of all managers.

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

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

9. Display the one month salary of employees written as:
   
   KING: 1 Month salary = 5000
   
   Hint: use literal character strings
Lab Session 02

OBJECT

Basic data retrieval operations in SQL

THEORY

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:-

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

*Examples*

i. 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');
```

ii. 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');
```

ii. **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*

i. Selecting employee number, name and their job

```
SELECT EMPNO, ENAME, JOB 
FROM EMP;
```

ii. Selecting employee number, name and their salary who do not earn commission

```
SELECT EMPNO, ENAME, SAL
FROM EMP
WHERE COMM IS NULL;
```

iii. **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;
```

![Figure 2.2: Joining tables using equi-join](image)

**NOTE:** Different join operations are discussed in detail in lab session 3.

**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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>

Table 2.1

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BETWEEN … AND …</td>
<td>Between two values (inclusive)</td>
</tr>
<tr>
<td>IN (list)</td>
<td>Match any of a list of values</td>
</tr>
<tr>
<td>LIKE</td>
<td>Match a character pattern</td>
</tr>
<tr>
<td>IS NULL</td>
<td>Is a null value</td>
</tr>
</tbody>
</table>

Table 2.2
Examples

a. To display record of employees who are not managers.
   
   ```sql
   SELECT * FROM EMP
   WHERE JOB <> 'MANAGER';
   ```

b. 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.
   
   ```sql
   SELECT EMPNO, ENAME, SAL, MGR
   FROM EMP
   WHERE MGR IN (7902, 7566, 7788);
   ```

c. To display the names of all employees with names starting with S,
   
   ```sql
   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.

d. To display the names of all employees with second character of name as A,
   
   ```sql
   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:-

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Returns TRUE if both component conditions are TRUE</td>
</tr>
<tr>
<td>OR</td>
<td>Returns TRUE if either component condition is TRUE</td>
</tr>
<tr>
<td>NOT</td>
<td>Returns TRUE if the following condition is FALSE</td>
</tr>
</tbody>
</table>

**Table 2.3**

Examples

- To display record of all clerks who earn more than 1100
  
  ```sql
  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.
  
  ```sql
  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.
  
  ```sql
  SELECT ename, job
  FROM emp
  WHERE job NOT IN ('CLERK', 'MANAGER', 'ANALYST');
  ```
Rules of Precedence

<table>
<thead>
<tr>
<th>Order Evaluated</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>All comparison operators</td>
</tr>
<tr>
<td>2</td>
<td>NOT</td>
</tr>
<tr>
<td>3</td>
<td>AND</td>
</tr>
<tr>
<td>4</td>
<td>OR</td>
</tr>
</tbody>
</table>

Table 2.4

For example, consider the following statement:-
SELECT ename, job, sal FROM emp
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 ename, job, sal FROM emp
WHERE (job = 'SALESMAN' OR job = 'PRESIDENT')
AND sal > 1500;`

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 where as DESC specifies descending order. ASC is the default order.

**Examples**

i. To select data in the increasing order of hiredate,

`SELECT ENAME, JOB, DEPTNO, HIREDATE FROM EMP
ORDER BY HIREDATE;`

ii. To select data in the decreasing order of hiredate,

`SELECT ENAME, JOB, DEPTNO, HIREDATE FROM EMP
ORDER BY HIREDATE DESC;`

iii. To sort by column alias,

`SELECT EMPNO, ENAME, SAL*12 ANNSAL FROM EMP
ORDER BY ANNSAL.``

iv. 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.

v. 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;

vi. 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. Define the different capabilities of SELECT statement? Give an example of each.

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

2. Write down SQL queries to perform following functions:-
   i. To display the name and department number of employee with number 7566.
ii. To display the name and department number of all employees in departments 10 and 30 in alphabetical order by name.

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

iv. 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

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

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

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

* * * * *
Lab Session 03

OBJECT

Data retrieval operations in SQL using join operations

THEORY

In lab session 2, 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 whereas the last one is in DEPT table (see lab session 1). To produce the report, we need to link the EMP and DEPT tables and access data from both of them. This is called join operation. To gain better understanding of join, it would be helpful to first clarify the concept of Cartesian Product.

Figure 3.1: The join operation collects data from multiple sources

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

There are various forms of join operation, each with subtle differences, some more useful than others. The Oracle 9i database offers join syntax that is SQL 1999 compliant. Prior to release 9i, the join syntax was different from the ANSI standard. However, the new syntax does not offer any performance benefits over the Oracle proprietary join syntax that existed in prior releases.
i. **Inner-Join/Equi-Join**

If the join contains an equality condition, it is called equi-join.

**Examples**

i. 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*.

```sql
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:-

```sql
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**

```sql
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:-

```sql
SELECT E.ENAME, D.DEPTNO, D.DNAME
FROM EMP E LEFT OUTER JOIN DEPT D
ON (E.DEPTNO = D.DEPTNO);
```

**Right Outer Join**

```sql
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:-

```sql
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 3.2.
Figure 3.2: Joining tables using right outer-join

**Full Outer Join**

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

```sql
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.

```sql
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*.

```sql
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 04

OBJECT

Single-row and multiple-row functions in SQL

THEORY

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 datatypes

SQL functions may accept arguments and always return a value as illustrated in figure 4.1.

![Figure 4.1: Functions accept arguments and return a value](image)

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOWER(‘SQL Course’)</td>
<td>sql course</td>
</tr>
<tr>
<td>UPPER(‘SQL Course’)</td>
<td>SQL COURSE</td>
</tr>
<tr>
<td>INITCAP(‘SQL Course’)</td>
<td>Sql Course</td>
</tr>
</tbody>
</table>

Table 4.1

Examples
i. 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;

ii. 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCAT(‘Good’, ‘String’)</td>
<td>GoodString</td>
</tr>
<tr>
<td>SUBSTR(‘String’, 2, 4)</td>
<td>trin</td>
</tr>
<tr>
<td>LENGTH(‘String’)</td>
<td>6</td>
</tr>
<tr>
<td>INSTR(‘String’, ‘r’)</td>
<td>3</td>
</tr>
<tr>
<td>LPAD(sal, 10, ‘*’)</td>
<td>******5000</td>
</tr>
</tbody>
</table>

Table 4.2

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUND(45.927, 2)</td>
<td>45.93</td>
</tr>
<tr>
<td>ROUND(45.927)</td>
<td>46</td>
</tr>
<tr>
<td>ROUND(45.927, -1)</td>
<td>50</td>
</tr>
<tr>
<td>TRUNC(45.927, 2)</td>
<td>45.92</td>
</tr>
<tr>
<td>TRUNC(45.927)</td>
<td>45</td>
</tr>
<tr>
<td>MOD(20, 3)</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 4.3

Examples
i.  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.

ii. SELECT TRUNC(45.923, 2), TRUNC(45.923), TRUNC(45.923, -1)
    FROM DUAL;

iii. 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

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONTHS_BETWEEN('01-SEP-95', '11-JAN-94')</td>
</tr>
<tr>
<td>ADD_MONTHS('11-JAN-94', 6)</td>
</tr>
<tr>
<td>NEXT_DAY('01-SEP-95', 'FRIDAY')</td>
</tr>
<tr>
<td>LAST_DAY('01-SEP-95')</td>
</tr>
<tr>
<td>ROUND(TO_DATE('25-JUL-95', 'DD-MON-YY'), 'MONTH')</td>
</tr>
<tr>
<td>ROUND(TO_DATE('25-JUL-95', 'DD-MON-YY'), 'YEAR')</td>
</tr>
<tr>
<td>TRUNC(TO_DATE('25-JUL-95', 'DD-MON-YY'), 'MONTH')</td>
</tr>
<tr>
<td>TRUNC(TO_DATE('25-JUL-95', 'DD-MON-YY'), 'YEAR')</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.6774194</td>
</tr>
<tr>
<td>'11-JUL-94'</td>
</tr>
<tr>
<td>'08-SEP-95'</td>
</tr>
<tr>
<td>'30-SEP-95'</td>
</tr>
<tr>
<td>01-AUG-95</td>
</tr>
<tr>
<td>01-JAN-96</td>
</tr>
<tr>
<td>01-JUL-95</td>
</tr>
<tr>
<td>01-JAN-95</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of months between two dates</td>
</tr>
<tr>
<td>Add calendar months to dates</td>
</tr>
<tr>
<td>Next day of the date specified</td>
</tr>
<tr>
<td>Last day of the month</td>
</tr>
<tr>
<td>Round date</td>
</tr>
<tr>
<td>Round date</td>
</tr>
<tr>
<td>Truncate date</td>
</tr>
<tr>
<td>Truncate date</td>
</tr>
</tbody>
</table>

### Examples

i. 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.

```sql
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;
```

ii. 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.

```sql
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`
TO_CHAR function with Dates

i. 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.

ii. 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.

iii. 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.

i. 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]);

Examples

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

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

ii. 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
Examples
i. 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;

ii. To show the minimum and maximum hiredate for employees
SELECT MIN (hiredate), MAX(hiredate) FROM emp;

iii. To return the number of rows in a table

25
iv. To return the number of nonnull rows in a table
SELECT COUNT(comm)
FROM emp
WHERE deptno = 30;

v. 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
i. 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.

ii. 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
i. 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)
ii. To display the job title and total monthly salary for each job title with a total payroll exceeding 5000.
   ```sql
   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

i. To display the maximum average salary by nesting group functions
   ```sql
   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.

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

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 05

OBJECT

Subqueries and compound queries in SQL

THEORY

SUBQUERIES

Why use subqueries?

Suppose we want to write a query to find out who earns a salary greater than Jones’ salary. To solve this problem, we need two queries: one query to find what Jones earns and a second query to find who earns more than that amount.

The above problem can be solved by combining the two queries, placing one query inside the other query.

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.

<table>
<thead>
<tr>
<th>Main Query</th>
<th>“Which employees have a salary greater than Jones’ Salary?”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subquery</td>
<td>“What is Jones’ salary?”</td>
</tr>
</tbody>
</table>

Figure 5.1 Subquery

A subquery is a SELECT statement that is embedded in a clause of another SELECT statement. They can be very useful when we need to select rows from a table with a condition that depends on the data in the table itself. The subquery generally executes first and its output is used to complete the query condition for the main or outer 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
i. 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);

ii. 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);

iii. 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);

iv. 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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>Equal to any member in the list</td>
</tr>
<tr>
<td>ANY</td>
<td>Compare value to each value returned by the subquery</td>
</tr>
<tr>
<td>ALL</td>
<td>Compare value to every value returned by the subquery</td>
</tr>
</tbody>
</table>

Table 5.1

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

**Examples**

i. 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);

ii. 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';

iii. 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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNION</td>
<td>All distinct rows selected by either query</td>
</tr>
<tr>
<td>UNION ALL</td>
<td>All rows selected by either query including all duplicates</td>
</tr>
<tr>
<td>INTERSECT</td>
<td>All distinct rows selected by both queries</td>
</tr>
<tr>
<td>MINUS</td>
<td>All distinct rows that are selected by the first SELECT statement and that are not selected in the second SELECT statement</td>
</tr>
</tbody>
</table>

Table 5.2

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. Why are subqueries needed in accessing data from the database?

__________________________________________________________________________
__________________________________________________________________________
__________________________________________________________________________
__________________________________________________________________________
__________________________________________________________________________
__________________________________________________________________________

__________________________________________________________________________
2. Write down the restrictions on using set operators in SQL.

__________________________________________________________________________
__________________________________________________________________________
__________________________________________________________________________
__________________________________________________________________________
__________________________________________________________________________
__________________________________________________________________________

3. 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 a department with any employee with a T in their name.
Lab Session 06

OBJECT

Creating readable output with SQL*Plus

THEORY

Interactive Reports
Using SQL*Plus, we can create reports that prompt the user to supply their own values to restrict the range of data returned. To create interactive reports, we can embed substitution variables in a command file or in a single SQL statement.

Substitution Variables
In SQL*Plus, we can use single-ampersand (&) substitution variables to temporarily store values. These variables can be used to dynamically restrict the data returned from a query.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;user_variable</td>
<td>Indicates a variable in a SQL statement; if the variable does not exist, SQL<em>Plus prompts the user for a value (SQL</em>Plus discards a new variable once it is used)</td>
</tr>
</tbody>
</table>

Table 6.1
For example, the following statement prompts the user for an employee number at runtime and displays employee number, name, salary and department number for that employee.

```sql
SELECT EMPNO, ENAME, SAL, DEPTNO
FROM EMP
WHERE EMPNO = &EMPLOYEE_NUM;
```

NOTE: With the single ampersand, the user is prompted every time the command is executed, if the variable does not exist.

Using SET VERIFY Command
To confirm the changes in the SQL statement, use the SQL*Plus SET VERIFY command. Setting SET VERIFY ON forces SQL*Plus to display the text of a command before and after it replaces substitution variables with values.

```sql
SQL> SET VERIFY ON
SQL> SELECT EMPNO, ENAME, SAL, DEPTNO
FROM EMP
WHERE EMPNO = &EMPLOYEE_NUM;
```

The following example displays the old as well as the new value of the column EMPNO.

Enter value for employee_num: 7369
Old  3: WHERE EMPNO = &EMPLOYEE_NUM
New  3: WHERE EMPNO = 7369

Character and Date Values with Substitution Variables
In a WHERE clause, date and character values must be enclosed within single quotation marks. The same rule applies to the substitution variables. To avoid entering the quotation
marks at runtime, enclose the variable in single quotation marks within the SQL statement itself.
For example, the query below retrieves the employee name, department number and annual salary of all employees based on the job title entered at the prompt by the user. It is also possible to use functions such as UPPER and LOWER with the ampersand. Use UPPER(‘&job_title’) so that the user does not have to enter the job title in uppercase.

```
SELECT ENAME, DEPTNO, SAL * 12
FROM EMP
WHERE JOB = ‘&job_title’;
```

**Customizing the SQL*Plus Environment**
We can control the environment in which SQL*Plus is currently operating by using the SET commands as follows:-

```
SET system_variable value
```

To verify what we have set by using the SHOW command.

```
SQL> SET ECHO ON
```

To see all SET variable values, use the SHOW ALL command.

**SET Command Variables**

<table>
<thead>
<tr>
<th>SET variable and values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COLSEP { _</td>
<td>text}</td>
</tr>
<tr>
<td>FEED[BACK]{6</td>
<td>n</td>
</tr>
<tr>
<td>HEA[ETING] {OFF</td>
<td>ON}</td>
</tr>
<tr>
<td>LIN[ESIZE] {80</td>
<td>n}</td>
</tr>
<tr>
<td>PAGESIZE {24</td>
<td>n}</td>
</tr>
<tr>
<td>PAUSE {OFF</td>
<td>ON</td>
</tr>
</tbody>
</table>

**Table 6.2**

NOTE: The value n represents a numeric value. The underlined value indicates default values. If we enter no values with the variable, SQL*Plus assumes the default value.

**Saving Customizations in login.sql file**
The `login.sql` file contains standard SET and other SQL*Plus commands that may be required for every session. The file is read and commands are implemented at login. When the session is logged out, all customized settings are lost.
The settings implemented by `login.sql` can be changed during the current session. Changes made are current only for that session. As soon as the session is logged out, those settings are lost. Add permanent changes to settings to the `login.sql` file.
SQL*Plus Format Commands
We can control the report features by using the following commands:-

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COL[UMN]</td>
<td>Controls column formats</td>
</tr>
<tr>
<td>TTI[TLE]</td>
<td>Specifies a header to appear at the top of each page</td>
</tr>
<tr>
<td>BTITLE</td>
<td>Specifies a footer to appear at the bottom of each page</td>
</tr>
<tr>
<td>BRE[AK]</td>
<td>Suppresses duplicate values and sections rows of data with line feeds</td>
</tr>
</tbody>
</table>

Table 6.3

The COLUMN Command
The column command controls display of a column.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLE[AR]</td>
<td>Clears any column formats</td>
</tr>
<tr>
<td>FOR[MAT]</td>
<td>Changes the display of the column data</td>
</tr>
<tr>
<td>HEA[DING] text</td>
<td>Sets the column headings (A vertical line (</td>
</tr>
<tr>
<td>JUS[TIFY] {align}</td>
<td>Justifies the column heading (not the data) to left, center or right</td>
</tr>
</tbody>
</table>

Table 6.4

For example to create different column headings
COLUMN ENAME HEADING ‘Employee | Name’ FORMAT A15
COLUMN SAL JUSTIFY LEFT FORMAT $99,990.00
COLUMN MGR FORMAT 9999999999 NULL ‘No Manager’

Display the current setting for the ENAME column.
COLUMN ENAME

To clear settings for the ENAME column.
COLUMN ENAME CLEAR

NOTE: If the command is lengthy, we can continue on the next line by ending the current line with a hyphen (-).

The BREAK Command
Use the BREAK command to section out rows and suppress duplicate values. To ensure that the BREAK command works effectively, use the ORDER BY clause to order the columns that you are breaking on.

To suppress duplicates
BREAK ON ENAME ON JOB
To section out rows at break values
BREAK ON ENAME SKIP 4 ON JOB SKIP 2
SKIP n is used to skip n number of lines when the break value changes.

Clear all BREAK settings by using the CLEAR command.
CLEAR BREAK
The TTITLE AND BTITLE Commands
Use the TTITLE command to format page headers and the BTITLE command for footers. Footers appear at the bottom of the page according to the PAGESIZE value. The syntax for BTITLE and TTITLE is identical. We can use vertical bar ( | ) to split the text of the title across several lines. The TTITLE example below sets the report header to display Salary centered on one line and Report centered below it. The BTITLE example sets the report footer to display Confidential. TITLE automatically puts date and page number on the report.

Creating a Script File to Run a Report
i. Create the SQL SELECT statement at the SQL prompt.
ii. Save the SELECT statement to a script file.
iii. Edit the script file to enter the SQL*Plus commands.
iv. Add the required formatting commands before the SELECT statement.
v. Verify that the SELECT statement is followed by a run character, either a semicolon (;) or a slash (/).
vi. Clear formatting commands after the SELECT statement.

Example
Create a script file to create a report that displays the job title, name and salary for every employee whose salary is less than $3000. Add a centered, two-lined header that reads Employee Report and a centered footer that reads Confidential. Rename the job title column to read Job Category split over two lines. Rename the employee name column to read Employee. Rename the salary column to read Salary and format it as $2,500.00

```
SET PAGESIZE 40
SET LINESIZE 60
SET FEEDBACK OFF
TTITLE 'Employee | Report'
BTITLE 'Confidential'
BREAK ON JOB
COLUMN JOB HEADING 'Job|Category' FORMAT A15
COLUMN ENAME HEADING 'Employee' FORMAT A15
COLUMN SAL HEADING 'Salary' FORMAT $99,999.99
REM ** Insert SELECT statement
SELECT  JOB, ENAME, SAL
FROM    EMP
WHERE   SAL < 3000
ORDER BY  JOB, ENAME
/
SET FEEDBACK ON
REM clear all formatting commands …
```
EXERCISES

1. Write a script file to display the employee name, job and hire date for all employees who started between a given range of dates. Concatenate the name and job together separated by a comma and space and label the column EMPLOYEES. Prompt the user for the two ranges and use the format MM/DD/YYYY.

2. Create a report containing the department name, employee name, hire date, salary and each employee’s annual salary for all employees in a given location. Prompt the user for the location. Label the columns DEPARTMENT NAME, EMPLOYEE NAME, START DATE, SALARY and ANNUAL SALARY, placing the labels on multiple lines. Suppress the duplicates on department name. Give suitable title to the report.
Lab Session 07

OBJECT

Data manipulation operations in SQL

THEORY

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 Oracle server must guarantee that all three SQL statements are performed to maintain the accounts in the proper balance. When something prevents one of the statements in the transaction from executing, the other statements of the transaction must be undone.

The SQL DML includes statements to perform following operations:-

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT</td>
<td>Enter new rows into tables</td>
</tr>
<tr>
<td>UPDATE</td>
<td>To change existing rows</td>
</tr>
<tr>
<td>DELETE</td>
<td>To delete existing rows</td>
</tr>
</tbody>
</table>

Table 7.1

Adding a new row to a table

We can add new rows to a table by using the INSERT statement. The syntax is

\[
\text{INSERT INTO table [(column [, column ...])] VALUES (value [, value ...])};
\]

Examples

i. Inserting a new row in the dept table

\[
\text{INSERT INTO dept (deptno, dname, loc) VALUES (50, 'DEVELOPMENT', 'DETROIT');}
\]

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)

ii. Inserting rows with Null values

   o Implicit Method: Omit the column from the column list.
     \[
     \text{INSERT INTO dept (deptno, dname) VALUES (60, 'MIS');}
     \]
   
   o Explicit Method: Specify the NULL keyword
     \[
     \text{INSERT INTO dept VALUES (70, 'FINANCE', NULL);}
     \]

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.
iii. 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.

iv. 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); 

v. 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

vi. 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.

**Examples**

i. To transfer an employee with number 7782 to department 20.
   
   UPDATE  emp 
   SET     deptno = 20 
   WHERE empno = 7782;

ii. All rows in the table are modified if the WHERE clause is omitted.
   
   UPDATE  emp 
   SET     deptno = 20;

iii. 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];

Examples
i. Specific rows are deleted from a table by specifying the WHERE clause.
   DELETE FROM department
       WHERE dname = 'DEVELOPMENT';
ii. All rows in the table are deleted if we omit the WHERE clause.
    DELETE FROM department;
iii. Remove all employees who started after January 1, 1997.
    DELETE FROM employee
        WHERE hiredate > TO_DATE('01.01.97', 'DD.MM.YY');
iv. 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Manipulation language (DML)</td>
<td>Consists of any number of DML statements that the Oracle Server treats as a single entity or a logical unit of work</td>
</tr>
<tr>
<td>Data Definition language (DDL)</td>
<td>Consists of only one DDL statement</td>
</tr>
<tr>
<td>Data Control language (DCL)</td>
<td>Consists of only one DCL statement</td>
</tr>
</tbody>
</table>

Table 7.2
A transaction begins when the first executable SQL statement is encountered and terminates
when one of the following occurs:
  v. A COMMIT or ROLLBACK statement is issued
  vi. A DDL statement, such as CREATE, is issued

44
vii. A DCL statement is issued
viii. The user exits SQL*Plus
ix. 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. Define Transaction. How it is terminated? Describe the different operations included in a transaction.

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

45
2. Write a transaction to insert following rows in EMP table.

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>ENAME</th>
<th>JOB</th>
<th>MGR</th>
<th>HIREDATE</th>
<th>SAL</th>
<th>COMM</th>
<th>DEPTNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>7123</td>
<td>RALPH</td>
<td>DESIGNER</td>
<td>7566</td>
<td>21-APR-85</td>
<td>2300</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>7890</td>
<td>GEORGE</td>
<td>CLERK</td>
<td>7566</td>
<td>03-MAY-85</td>
<td>1235</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>7629</td>
<td>BOB</td>
<td>SALESMAN</td>
<td>7698</td>
<td>06-MAR-86</td>
<td>1800</td>
<td>1000</td>
<td>30</td>
</tr>
</tbody>
</table>

3. 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

OBJECT

Creating and managing tables and views

THEORY

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**

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR2(size)</td>
<td>Variable-length character data (A maximum size must be specified. Default and minimum size is 1; maximum size is 4000)</td>
</tr>
<tr>
<td>CHAR(size)</td>
<td>Fixed-length character data of length size bytes (Default and minimum size is 1; maximum size is 2000)</td>
</tr>
<tr>
<td>NUMBER(p, s)</td>
<td>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.)</td>
</tr>
<tr>
<td>DATE</td>
<td>Date and time values between January 1, 4712 B.C. and December 31, 9999 A.D.</td>
</tr>
<tr>
<td>LONG</td>
<td>Variable length character data up to 2 gigabytes</td>
</tr>
<tr>
<td>Datatype</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>CLOB</td>
<td>Single-byte character data up to 2 gigabytes</td>
</tr>
<tr>
<td>RAW(size)</td>
<td>Raw binary data of length size (A maximum size must be specified. Maximum size is 2000.)</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>Raw binary data of variable length up to 2 gigabytes</td>
</tr>
<tr>
<td>BLOB</td>
<td>Binary data up to 4 gigabytes</td>
</tr>
<tr>
<td>BFILE</td>
<td>Binary data stored in an external file; up to 4 gigabytes</td>
</tr>
</tbody>
</table>

Table 8.1

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:

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

Table 8.2

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, 2) 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. A view contains no data of its own but is like a window through which data from tables can be viewed or changed. The tables in which a view is based are called base tables. The view is stored as a SELECT statement in the data dictionary.

Why Use Views?
Views are used for following reasons:-
- To restrict database access because the view can display a selective portion of the database.
- To make complex queries easy. For example, views allow users to query information from multiple tables without knowing how to write a join statement.
- To allow data independence for ad hoc users and application programs. One view can be used to retrieve data from several tables.
- To present different views of the same data to different users.

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.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Simple Views</th>
<th>Complex Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of tables</td>
<td>One</td>
<td>One or more</td>
</tr>
<tr>
<td>Contain functions</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Contain groups of data</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>DML through view</td>
<td>Yes</td>
<td>Not always</td>
</tr>
</tbody>
</table>

Table 8.3

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.

<table>
<thead>
<tr>
<th>S.N.</th>
<th>Designation</th>
<th>Grade</th>
<th>TotalPosts</th>
<th>PostsAvailable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CLERK</td>
<td>12</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>PROGRAMMAR</td>
<td>17</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>DATABASE DEVELOPER</td>
<td>17</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>SENIOR SYSTEM ANALYST</td>
<td>18</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>S.N.</th>
<th>Empno</th>
<th>Name</th>
<th>Designation</th>
<th>Qualification</th>
<th>JoinDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3400</td>
<td>ASIF</td>
<td>Database Developer</td>
<td>B.E.</td>
<td>25th Nov, 2000</td>
</tr>
<tr>
<td>2</td>
<td>3108</td>
<td>MEHMOOD</td>
<td>Senior System Analyst</td>
<td>M.S.</td>
<td>16th May, 1996</td>
</tr>
<tr>
<td>3</td>
<td>3345</td>
<td>FARHAN</td>
<td>Database Developer</td>
<td>B.E.</td>
<td>23rd April, 2001</td>
</tr>
<tr>
<td>4</td>
<td>3315</td>
<td>NAVEED</td>
<td>Clerk</td>
<td>B.A.</td>
<td>11th Jan, 1997</td>
</tr>
<tr>
<td>5</td>
<td>3300</td>
<td>NAVEEN</td>
<td>Clerk</td>
<td>B.A.</td>
<td>24th Feb, 1998</td>
</tr>
</tbody>
</table>
5. Write down a transaction to insert following data in TRAINING table. The data should be finally saved in the database.

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

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

<table>
<thead>
<tr>
<th>S.N.</th>
<th>PID</th>
<th>Title</th>
<th>Client</th>
<th>Duration (in weeks)</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>P023</td>
<td>Payroll</td>
<td>Superior Bank</td>
<td>12</td>
<td>New</td>
</tr>
<tr>
<td>2.</td>
<td>P321</td>
<td>Accounts</td>
<td>ABC Leasing</td>
<td>16</td>
<td>In Progress</td>
</tr>
<tr>
<td>3.</td>
<td>P178</td>
<td>Taxation</td>
<td>Farhan Motors</td>
<td>4</td>
<td>Complete</td>
</tr>
<tr>
<td>4.</td>
<td>P315</td>
<td>Payroll</td>
<td>ABC Leasing</td>
<td>8</td>
<td>New</td>
</tr>
<tr>
<td>5.</td>
<td>P300</td>
<td>Inventory</td>
<td>Khurran Textile Mills</td>
<td>12</td>
<td>In Progress</td>
</tr>
</tbody>
</table>
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. What is database schema? What are the different objects included in it?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

58
10. What are *large object* data types? What are their different applications with respect to data storage?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

11. What are the different uses of views? Differentiate between simple and complex views.

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
12. 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

OBJECT

Creating sequences, indexes and synonyms

THEORY

SEQUENCES

What is a Sequence?

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 constraint.
- **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;
```

**When to create an index**

The index should be created under following circumstances:-

- The column is used frequently in the WHERE clause or in a join condition.
- The column contains a wide range of values.
- The column contains a large number of null values
- Two or more columns are frequently used together in a WHERE clause or a join condition.
- The table is large and most queries are expected to retrieve less than 2-4% of the rows.

**When not to create an index**

The index should not be created under following circumstances:-

- The table is small
- The columns are not often used as a condition in the query.
- Most queries are expected to retrieve more than 2-4% of the rows.
- The table is updated frequently.

**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

4. What are the different situations in which it is not appropriate to create an index?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

5. What are the two different views in the oracle data dictionary that contains information about user indexes? What information is contained in these views?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

6. What are synonyms? What are their advantages?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

* * * * *
Lab Session 10

OBJECT

Managing profiles and controlling user access

THEORY

Controlling database access and resource limits are important aspects of the DBA’s function. Profiles are used to manage the database and system resources and to manage database passwords and password verification. Database access is controlled using privileges. Roles are created to manage the privileges.

PROFILES

In Oracle, a profile is a named set of password and resource limits. In essence, they are used for two major purposes:

i. Profiles are used to control the database and system resource usage. They restrict users from performing some operations that require heavy use of resources. Resource management limits can be enforced at the session level, the call level or both. Oracle provides a set of predefined resource parameters that can be used to monitor and control database resource usage. The DBA can define limits for each resource by using a database profile.

ii. Profiles are also used for password management to provide greater control over database security. e.g. to force a user to change password after a specified time.

It is possible to create various profiles for different user communities and assign a profile to each user. When the database is created, Oracle creates a profile named DEFAULT, and if we do not specify a profile for the user, the DEFAULT profile is assigned.

![Figure 10.1: Profiles allocate resources among users to varying degrees](image)

Oracle lets us control the following types of resource usage (both at session level and call level) through profiles:

- Concurrent sessions per user
- Elapsed and idle time connected to the database
- CPU time used (per session and per call)
- Logical reads performed (per session and per call)

Oracle lets us control the following password management features through profiles:

- Password aging and expiration
- Password history
- Password complexity verification
- Account Locking
Enforcing resource limits
Resource limits are enforced only when the parameter RESOURCE_LIMIT is set to TRUE. Even if the profiles are defined and assigned to users, Oracle enforces them only when this parameter is set to TRUE. This parameter can be set in the initialization parameter file so that every time the database starts, the resource usage is controlled for each user using the assigned profile. Resource limits can also be enabled or disabled using the ALTER SYSTEM command. The default value of RESOURCE_LIMIT is FALSE.

```
ALTER SYSTEM
SET RESOURCE_LIMIT = TRUE;
```

Most resource limits are set at the session level; a session is created when a user connects to the database. Certain limits can be controlled at the statement level (but not at the transaction level).

If a user exceeds a resource limit, Oracle aborts the current operation, rolls back the changes made by the statement, and returns an error. The user has the option of committing or rolling back the transaction, because the statements issued earlier in the transaction are not aborted. No other operation is permitted when a session level limit is reached. The user can disconnect, in which case the transaction is committed. In contrast to resource limits, password limits are always enforced.

Some of the parameters that are used to control resources are as follows:-
- SESSIONS_PER_USER
- CPU_PER_SESSION
- CPU_PER_CALL
- LOGICAL_READS_PER_SESSION
- LOGICAL_READS_PER_CALL
- CONNECT_TIME
- IDLE_TIME

Some of the parameters that are used for password management are as follows:-
- FAILED_LOGIN_ATTEMPTS
- PASSWORD_LOCK_TIME
- PASSWORD_LIFE_TIME
- PASSWORD_GRACE_TIME
- PASSWORD_REUSE_TIME
- PASSWORD_REUSE_MAX

Creating Profiles
Let’s create a profile to manage passwords and resources for the accounting department users. The users are required to change their password every 60 days and they cannot reuse a given password for 90 days. However a given password can be reused any number of times throughout life. The grace period for password change is 5 days. They are allowed to make a typo in the password only four consecutive times while connecting to the database; if the login fails for a fifth time, their account will be locked forever (until the DBA or security department unlocks the account). The accounting department users are allowed to have a maximum of six database connections; they can stay connected to the database for 24 hours, but an inactivity of half-an-hour will terminate their session.

```
CREATE PROFILE ACCOUNTING_USER LIMIT
SESSIONS_PER_USER  6
CONNECT_TIME   1440
```
IDLE_TIME 30
FAILED_LOGIN_ATTEMPTS 4
PASSWORD_LOCK_TIME UNLIMITED
PASSWORD_LIFE_TIME 60
PASSWORD_GRACE_TIME 5
PASSWORD_REUSE_TIME 90
PASSWORD_REUSE_MAX UNLIMITED;

Dropping profiles
Profiles are dropped using the DROP PROFILE command. If any user is assigned the profile you wish to drop, Oracle returns an error. You can drop such profiles by specifying CASCADE, in which case the users who have that profile will be assigned the DEFAULT profile.
DROP PROFILE ACCOUNTING_USER CASCADE;

Assigning Profiles
Profiles can be assigned to users by using the CREATE USER or ALTER USER command. Following example assigns the ACCOUNTING_USER profile to an existing user named SCOTT:-
ALTER USER SCOTT
PROFILE ACCOUNTING_USER;

Querying Password and Resource Limits Information
Information about password and resource limits can be obtained by querying the following data dictionary views:-
- DBA_USERS
- DBA_PROFILES

TABLESPACES AND DATAFILES
The database’s data is stored logically in tablespaces and physically in the data files corresponding to the tablespaces. The logical storage management is independent of the physical storage of the data files. A tablespace can have more than one data file associated with it whereas one data file belongs to only one tablespace.
A database can have one or more tablespaces. Figure 10.2 below shows the relationship between the database, tablespace, data files and the objects in the database.
Any object (such as a table, an index, etc.) created in the database is stored in a single tablespace. But the object’s physical storage can be on multiple data files belonging to that tablespace.
The idea of tablespaces allows managing space for users in an efficient way. We can group application-related data (e.g. accounts) together so that when maintenance is required for the application’s tablespace, only that tablespace need to be taken offline and the rest of the database will be available for other users. Moreover, we can back up the database one tablespace at a time as well as make part of the database read-only.
The size of the tablespace is the total size of all the data files belonging to the tablespace. The size of the database is the total size of all tablespaces in the database, which is the total size of all data files in the database.
Changing the size of the data files belonging to a tablespace can change the size of that tablespace. More space can be added to a tablespace by adding more data files to the tablespace or increasing the size of the existing data files.
When a database is created, Oracle creates the SYSTEM tablespace. All data dictionary objects are stored in it. The data files specified at the time of database creation are assigned to the SYSTEM tablespace. The PL/SQL program units (such as procedures, functions, packages, or triggers) created in the database are also stored in the SYSTEM tablespace. It is recommended that SYSTEM tablespace should contain only the objects in the data dictionary. Separating the data dictionary from other database objects reduces the contention between dictionary objects and database objects for the same data file.

**Creating Tablespace**
The following statement creates a tablespace ACCOUNT_DATA having two data files acct_data01.dbf and acct_data02.dbf,
```
CREATE TABLESPACE ACCOUNT_DATA 
DATAFILE 'd:\oradata\db01\acct_data01.dbf' SIZE 100M
    AUTOEXTEND ON NEXT 5M MAXSIZE 200M,
'D:\oradata\db01\acct_data02.dbf' SIZE 100M;
```
AUTOEXTEND ON enables the automatic extension of the data file. NEXT specifies the disk space to allocate to the data file when more space is needed. MAXSIZE specifies the maximum disk space allowed for allocation to the data file. If MAXSIZE is specified unlimited then this indicates that there is no limit on allocating disk space to the data file.

**Altering Tablespace**
We can add more space to a tablespace by adding more data files to it or by changing the size of the existing data files.
To add more data files,
ALTER TABLESPACE ACCOUNT_DATA ADD DATAFILE 'd:\oradata\db01\acct_data03.dbf' SIZE 100M;

We can increase or decrease the size of a data file by using the RESIZE clause of the ALTER DATABASE DATAFILE command.

ALTER DATABASE
DATAFILE 'd:\oradata\db01\appl_data01.dbf'
RESIZE 200M;

**Dropping Tablespace**
The following statement drops the tablespace APPL_DATA,
DROP TABLESPACE APPL_DATA;

**Undo Tablespace**
An undo tablespace is used to store undo segments. It cannot contain any other objects.
Unlike other tablespaces, the undo tablespace is limited to the DATAFILE. For example, to create an undo tablespace undo1
CREATE UNDO TABLESPACE undo1
DATAFILE 'd:\oradata\db01\undo01.dbf' SIZE 40M;

**Querying Tablespace Information**
Information about tablespaces can be obtained by querying the following data dictionary views:-
- DBA TABLESPACES
- V$TABLESPACE

**MANAGING USERS**
The following statement will create a user AHMED identified by password GREEN, having profile ACCOUNTING_USER. All the user’s objects will be created in the USERS tablespace in which the user AHMED will have a quota of 2MB. The user’s account will be initially locked.

CREATE USER AHMED
IDENTIFIED BY GREEN
DEFAULT TABLESPACE USERS
QUOTA 2M ON USERS
PROFILE ACCOUNTING_USER
ACCOUNT LOCK;

The DBA can later lock or unlock a user’s account as follows:-
ALTER USER AHMED
ACCOUNT UNLOCK;

The DBA can also expire the user’s password:
ALTER USER AHMED PASSWORD EXPIRE;
Users must change the password the next time they log in, or the DBA must change the password. If the password is expired, SQL*Plus prompts for a new password at login time.
The following example shows how to drop the user AHMED, with all the owned objects:

```
DROP USER AHMED CASCADE;
```

CONTROLLING USER ACCESS

PRIVILEGES

Privileges in the Oracle database control access to the data and restrict the actions users can perform. Through proper privileges, users can create, drop, or modify objects in their own schema or in another user’s schema. Privileges also determine what data a user should have access to.

Privileges can be granted to a user via two methods:
- Assign privileges directly to the user
- Assign privileges to a role, and then assign the role to the user

A role is a named set of privileges, which eases the management of privileges. For example, if there are 10 users needing access to the data in the accounting tables, we can grant the privileges required to a role and grant the role to the 10 users.

There are two types of privileges:

- **Object privileges** are granted on a specific object. The owner of the object has all the privileges on the object. The privileges can be on data (to read, modify, delete, add, or reference), on a program (to execute), or to modify an object (to change the structure).
- **System privileges** are the privileges that enable the user to perform an action on any schema in the database. They do not specify an object, but are granted at the database level. Like object privileges, system privileges also can be granted to a role. They are usually granted by DBA. Both system privileges and object privileges can be granted to a role.

PUBLIC is a user group defined in the database; it is not a database user or a role. Every user in the database belongs to this group. So if privileges are granted to PUBLIC, they are available to all users of the database.

**Granting Object Privileges**

Suppose user AHMED owns tables CUSTOMER (Customer_ID, Name, Address) and ORDER (Order_ID, Date, Customer_ID). AHMED wants to grant read and update privileges on CUSTOMER table to user YASIR. When multiple privileges are specified, they are separated by comma.

```
GRANT SELECT, UPDATE ON CUSTOMER
TO YASIR;
```

The INSERT and UPDATE privileges can be granted on columns also.

```
GRANT INSERT(Customer_id, Name), DELETE ON CUSTOMER
TO YASIR WITH GRANT OPTION;
```

The WITH GRANT OPTION clause allows YASIR to grant the privileges to others.

**Granting System Privileges**

Like object privileges, system privileges also can be granted to a user, role, or PUBLIC. There are many system privileges in Oracle. The CREATE, ALTER, and DROP privileges provide the ability to create, modify, and drop the object specified in the user’s schema. When a privilege is specified with ANY, it authorizes the user to perform the action on any schema in the database.

For example, to grant create session, create table and create index privilege to AHMED:

```
GRANT CREATE SESSION, CREATE TABLE,
```
CREATE INDEX TO AHMED WITH ADMIN OPTION;
The WITH ADMIN OPTION clause allows AHMED to grant the privileges to others.

The following statement allows AHMED to create and select a table in any schema as well as selecting any sequence,
GRANT CREATE ANY TABLE, SELECT ANY TABLE,
SELECT ANY SEQUENCE TO AHMED;

Revoking privileges
Object privileges and system privileges can be revoked from a user by using the REVOKE statement. To revoke the UPDATE privilege granted to YASIR from AHMED on AHMED’s CUSTOMER table:-

REVOKE UPDATE
ON CUSTOMER
FROM YASIR;

ROLES
A role is a named group of related privileges that can be granted to the user. This method makes it easier to revoke and maintain privileges.
A user can have access to several roles and several users can be assigned the same role. Roles are typically created for a database application.

e.g. to create a role manager
CREATE ROLE manager

To grant privileges to the manager role
GRANT CREATE TABLE, CREATE VIEW
TO manager;

To grant role to users,
GRANT manager TO AHMED, YASIR;

EXERCISES

1. Explain the purpose of profiles in a DBMS. How are they assigned to the users?
2. What are tablespaces? What are their different benefits in a database system?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

3. What different operations are involved in managing users in a DBMS?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

4. Differentiate between Object privileges and System privileges.

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
5. What are the advantages of defining a role in controlling user access?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

6. Write a statement to create a tablespace named ACCOUNTS_DATA having three files named DATA01, DATA02 and DATA03 of size 50M, 100M and 150M respectively in the path D:\DATAS. The first file should have unlimited size whereas the second and third should have max size 200MB. All of them should extend automatically by 10M.
7. Create two users ASAD and SAFIA identified by passwords GROUP and FRIEND respectively. Their default tablespaces should be APPL_DATA (with 2M quota) and ACCOUNTS_DATA (with 3M quota) respectively. ASAD would be granted default profile where as SAFIA would use ACCOUNTING_USER profile. The ASAD’s account would be locked where as SAFIA’s account would be unlocked initially. Also enable both users to create session.

8. Write SQL statements unlock ASAD’s account and extend quota on SAFIA’s account to 10M.

9. Write SQL statements to grant read, update (only on address) and delete privileges on CUSTOMER table to ASAD and SAFIA. Also grant read, insert and update privileges on ORDER to ASAD. SAFIA should also be able to grant her privileges to others.

* * * * *
Lab Session 11

OBJECT

Basic features of PL/SQL Programming

THEORY

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.

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

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 by 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.

The variables can be handled in the following manner:-
- Declare and initialize variables in the declaration section.
- Assign new values to variables in the executable section.
- Pass values into PL/SQL subprograms through parameters. There are three parameter modes, IN (default), OUT and IN OUT. The IN parameter is used to pass values to the subprogram being called, the OUT parameter is used to return values to the caller of a subprogram and the IN OUT parameter is used to pass initial values to the subprogram being called and to return updated values to the caller.
- View the results from a PL/SQL block through output variables.

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. The main datatypes are those that correspond to column types in Oracle Server tables; PL/SQL also supports Boolean variables. 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_count BINARY_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.

```sql
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.

```sql
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

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARCHAR2(\text{maximum length})</td>
<td>Base type for variable-length character data up to 32,767 bytes. There is no default size for VARCHAR2 variables and constants.</td>
</tr>
<tr>
<td>NUMBER [(\text{precision, scale})]</td>
<td>Base type for fixed and floating-point numbers.</td>
</tr>
<tr>
<td>DATE</td>
<td>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.</td>
</tr>
<tr>
<td>CHAR [(\text{maximum length})]</td>
<td>Base type for fixed-length character data up to 32,760 bytes. If the maximum length is not specified, the default length is set to 1.</td>
</tr>
<tr>
<td>LONG</td>
<td>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.</td>
</tr>
<tr>
<td>LONG RAW</td>
<td>Base type for binary data and byte strings up to 32,760 bytes. LONG RAW data is not interpreted by PL/SQL.</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>Base type that stores one of three possible values used for logical calculations TRUE, FALSE, or NULL.</td>
</tr>
<tr>
<td>BINARY INTEGER</td>
<td>Base type for integers between $-2,147,483,647$ and $2,147,483,647$.</td>
</tr>
<tr>
<td>PLS_INTEGER</td>
<td>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.</td>
</tr>
</tbody>
</table>

**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.

```sql
v_job VARCHAR2(9);
v_count BINARY_INTEGER := 0;
v_total_sal NUMBER(9, 2) := 0;
v_orderdate DATE := SYSDATE + 7;
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);

**Displaying Bind Variables**
To display the current value of bind variables in the SQL*Plus environment, we use the command PRINT. The following example illustrates a PRINT command.

VARIABLE g_n NUMBER
...
PRINT g_n

**Referencing Non-PL/SQL variables**
To reference host variables, prefix the references with a colon ( : ) to distinguish them from declared PL/SQL variables. The following example computes the monthly salary based upon the monthly annual salary supplied by the user. This script contains both SQL*Plus commands as well as a complete PL/SQL block.

| VARIABLE g_monthly_salary NUMBER 
| ACCEPT p_annual_sal PROMPT 'Please enter the annual salary : ' 
| DECLARE v_sal NUMBER(9, 2) := &p_annual_sal; 
| BEGIN 
| :g_monthly_sal := v_sal / 12; 
| END; 
| PRINT g_monthly_sal |

**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 |
EXERCISES

1. Write down the different features of PL/SQL programming language.

_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________

2. Describe the different block types in PL/SQL.

_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________

3. What are stored procedures and functions? How are they different from anonymous blocks?
4. What are the different types of variables in PL/SQL? Define each of them

5. What are the different numeric data types in PL/SQL? Describe each of them along with their range of values.
6. What are bind variables? How are they created? What are their advantages/uses?

7. 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%.
Lab Session 12

OBJECT

Writing executable statements in PL/SQL

THEORY

PL/SQL Block Syntax and Guidelines

Because PL/SQL is an extension of SQL, the general syntax rules that apply to SQL also apply to the PL/SQL language.

- Lexical units (for example, identifiers and literals) can be separated by one or more spaces or other delimiters that cannot be confused as being part of the lexical unit. We cannot embed spaces in lexical units except for string literals and comments.
- Statements can be split across lines, but keywords must not be split.

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

```plsql
... 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 12.1
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 12.2
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

```plsql
... x BINARY_INTEGER;
BEGIN
  ... DECLARE
    y NUMBER;
    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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>**, NOT</td>
<td>Exponentiation, logical navigation</td>
</tr>
<tr>
<td>+, -</td>
<td>Identity, negation</td>
</tr>
<tr>
<td>*, /</td>
<td>Multiplication, division</td>
</tr>
<tr>
<td>+, -,</td>
<td></td>
</tr>
<tr>
<td>=, !=, &lt;, &gt;, &lt;=, &gt;=, IS NULL, LIKE, BETWEEN, IN</td>
<td>Comparison</td>
</tr>
</tbody>
</table>
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.

INTO clause
The INTO clause is mandatory and occurs between the SELECT and FROM clauses. It is used to specify the names of variables to hold the values that SQL returns from the SELECT clause. We use the INTO clause to populate either PL/SQL variables or host variables. The SELECT statement queries must return one and only one row. More than one row or no row generates an error. PL/SQL deals with these errors by raising standard exceptions, which can be trapped in the exception section of the block with the NO_DATA_FOUND and TOO_MANY_ROWS exceptions.
Retrieving data in PL/SQL

**Example 1**
Retrieve the order date and the ship date for the specified order.

```sql
DECLARE
    V_orderdate ord.orderdate%TYPE;
    V_shipdate  ord.shipdate%TYPE;
BEGIN
    SELECT orderdate, shipdate
    INTO   v_orderdate, v_shipdate
    FROM  ord
    WHERE  id = 620;
    DBMS_OUTPUT.PUT_LINE('Order date = ' || v_orderdate);
    DBMS_OUTPUT.PUT_LINE('Ship date = ' || v_shipdate);
    ...
END;
```

**Example 2**
Return the sum of the salaries for all employees in the specified department.

```sql
DECLARE
    v_sum_sal emp.sal%TYPE;
    v_deptno  NUMBER NOT NULL := 10;
BEGIN
    SELECT SUM(sal)  -- group function
    INTO   v_sum_sal
    FROM  emp
    WHERE  deptno = v_deptno;
    DBMS_OUTPUT.PUT_LINE('Total salary in department ' || v_deptno || ' is ' || v_sum_sal);
    ...
END;
```

**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.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL%ROWCOUNT</td>
<td>Number of rows affected by the most recent SQL statement (an integer value)</td>
</tr>
<tr>
<td>SQL%FOUND</td>
<td>Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows</td>
</tr>
</tbody>
</table>
**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**

**PL/SQL Block**


define weight NUMBER(3) := 600;
define message VARCHAR2(255) := 'Product 10012';
BEGIN

/* SUB-BLOCK */
define weight NUMBER(3) := 1;
define message VARCHAR2(255) := 'Product 11001';
define new_locn VARCHAR2(50) := 'Europe';
BEGIN
  weight := weight + 1;
  new_locn := 'Western ' || new_locn;
END;

weight := weight + 1;
message := message || ' is in stock';
new_locn := 'Western ' || new_locn;
END;
1. Evaluate the PL/SQL block above and determine the datatype and value of each of the following variables according to the rules of scoping
   a) The value of V_WEIGHT in the subblock is: _______________________________________

   b) The value of V_NEW_LOCN in the subblock is: _______________________________________

   c) The value of V_WEIGHT in the main block is: _______________________________________

   d) The value of V_MESSAGE in the main block is: _______________________________________

   e) The value of V_NEW_LOCN in the main block is: _______________________________________

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. 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. What are identifiers? What are their naming rules in PL/SQL?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

4. What are literals? Define character and numeric literals and give examples of each.

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

5. What is the advantage of commenting code in programming? How are comments given in PL/SQL programs

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
6. 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 13

OBJECT

Using control structures in PL/SQL

THEORY

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.

- To set a flag for orders where there are fewer than five days between order date and ship date.
  ... 
  IF v_shipdate – v_orderdate < 5 THEN
    v_ship_flag := 'Acceptable';
  ELSE
    v_ship_flag := 'Unacceptable';
  END IF;
  Note: Complete the above program by entering orderid and print v_ship_flag.
• 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_count NUMBER(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;

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 rewards an employee by appending an asterisk in the STARS column (a new column in EMP table) for every 100 of the employee’s salary.
   - Accept the employee’s number as user input with a SQL*Plus substitution variable.
   - Initialize a variable that will contain a string of asterisks.
   - Append an asterisk to the string for every 100 of the salary amount. For example, if the employee has a salary
Lab Session 14

OBJECT

Writing explicit cursors in PL/SQL

THEORY

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.

<table>
<thead>
<tr>
<th>Cursor Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit</td>
<td>Implicit cursors are declared by PL/SQL implicitly for all DML and PL/SQL SELECT statements, including queries that return only one row.</td>
</tr>
<tr>
<td>Explicit</td>
<td>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.</td>
</tr>
</tbody>
</table>

Table 14.1

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 14.1 below show how an explicit cursor points to the current row in the active set.

![Figure 14.1: Explicit cursor containing the active set](image)

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/SQL 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.

```sql
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.

```sql
SET SERVEROUTPUT ON
DECLARE
    CURSOR emp_cursor IS
    SELECT ename, deptno
    FROM emp;
```

100
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.

```sql
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;
```

**EXERCISES**

1. What are explicit cursors? What are their uses in PL/SQL programming? Give two
examples.

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
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
Lab Session 15

OBJECT

Handling exceptions in PL/SQL

THEORY

What is an exception?
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:-

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Oracle Server Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUP_VAL_ON_INDEX</td>
<td>ORA-00001</td>
<td>Attempted to insert a duplicate value</td>
</tr>
<tr>
<td>INVALID_NUMBER</td>
<td>ORA-01722</td>
<td>Conversion of character string to number fails</td>
</tr>
<tr>
<td>NO_DATA_FOUND</td>
<td>ORA-01403</td>
<td>Single row SELECT returned no data</td>
</tr>
<tr>
<td>TIMEOUT_ON_RESOURCE</td>
<td>ORA-00051</td>
<td>Time-out occurred while Oracle is waiting for a resource</td>
</tr>
<tr>
<td>TOO_MANY_ROWS</td>
<td>ORA-01422</td>
<td>Single-row SELECT returned more than one row</td>
</tr>
<tr>
<td>VALUE_ERROR</td>
<td>ORA-06502</td>
<td>Arithmetic, conversion, truncation, or size-constraint error returned</td>
</tr>
<tr>
<td>ZERO_DIVIDE</td>
<td>ORA-01476</td>
<td>Attempted to divide by zero</td>
</tr>
</tbody>
</table>

Table 15.1

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.

```sql
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.

```sql
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. What is an exception? What are the different methods of raising an exception?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
2. 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
3. Write a PL/SQL block that prints the number of employees who make plus or minus 100 of the salary entered.
   a. If there is an employee within that salary range, print a message to the user indicating that is the case. Use an exception for this case.
   b. If there are one or more employees within that range, the message should indicate how many employees are in that salary range.
   c. Handle any other exception with an appropriate exception handler. The message should indicate that some other error occurred.

4. Write down a PL/SQL block to input an employee name and divide the employee’s salary with commission if the employee is a salesman and print the result. If the employee is not salesman, print message *employee not salesman* from the main block. If no record found or more than one record found, handle using exceptions. Also if the commission is zero, handle using ZERO_DIVIDE exception.
Appendix A

DATABASE TRIGGERS

What is a 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

```sql
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. For example, create one trigger to restrict all data manipulation events on the EMP table to certain business hours, Monday through Friday. Also 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.

<table>
<thead>
<tr>
<th>USER_NAME</th>
<th>TABLE_NAME</th>
<th>COLUMN_NAME</th>
<th>INS</th>
<th>UPD</th>
<th>DEL</th>
<th>MAX_INS</th>
<th>MAX_UPD</th>
<th>MAX_DEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCOTT</td>
<td>EMP</td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>SCOTT</td>
<td>EMP</td>
<td>SAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCOTT</td>
<td>EMP</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

```sql
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)
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**

<table>
<thead>
<tr>
<th><strong>Database Triggers</strong></th>
<th><strong>Stored Procedure</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Use CREATE TRIGGER</td>
<td>Use CREATE PROCEDURE</td>
</tr>
<tr>
<td>Data dictionary contains source and p-code</td>
<td>Data dictionary contains source and p-code</td>
</tr>
<tr>
<td>Implicitly invoked</td>
<td>Explicitly invoked</td>
</tr>
<tr>
<td>COMMIT, ROLLBACK And SAVEPOINT statements are not allowed within the trigger body</td>
<td>COMMIT, ROLLBACK And SAVEPOINT statements are permitted within the procedure body</td>
</tr>
</tbody>
</table>

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 TRIGGER 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 a 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.
Appendix B

STORED PROCEDURES

A 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

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

- 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.

<table>
<thead>
<tr>
<th>Type of Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN (default)</td>
<td>Passes a constant value from the calling environment into the procedure</td>
</tr>
<tr>
<td>OUT</td>
<td>Passes a value from the procedure to the calling environment</td>
</tr>
<tr>
<td>IN OUT</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Parameter Modes for Formal Parameters**

<table>
<thead>
<tr>
<th>IN</th>
<th>OUT</th>
<th>IN OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Must be specified</td>
<td>Must be specified</td>
</tr>
<tr>
<td>Value is passed into subprogram</td>
<td>Returned to calling environment</td>
<td>Passed into subprogram; returned to calling environment</td>
</tr>
<tr>
<td>Formal parameter acts as a constant</td>
<td>Uninitialized variable</td>
<td>Initialized variable</td>
</tr>
<tr>
<td>Actual parameter can be a literal, expression, constant or initialized variable</td>
<td>Must be a variable</td>
<td>Must be a variable</td>
</tr>
</tbody>
</table>
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 invoke 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;
12    END query_emp;
13    /
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.
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.

* * * * *
**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.

* * * * *
Appendix C

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

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

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;

EXERCISE

1. Differentiate between procedures and functions.

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

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

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

3. Consider the schema in lab session 6. Write down a stored function that takes employee number and returns the percentage of projects in which employee performance is excellent.
4. 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.

* * * * *