Syllabus

F. Y. B. Sc. (Information Technology)

Paper - III, SEM - III

ADVANCED SQL

Unit - I	**Structured Query Language :** Writing Basic SQL Select Statements, Restricting and Sorling Data, Single-Row Functions, Joins (Displaying Data from Multiple Tables), Aggregating Data using Group Functions, Subqueries, Manipulating Data, Creating and Managing Tables, Including Constraints, Creating Views, Creating other Database Objects (Sequences, Indexes and Synomyms)
Unit - II	**Advanced SQL :** Controlling user Access, using SET operators, Data Time Functions, Enhancements to Group by clause (cube, Rollup and Grouping), Advanced Subqueries (Multiple column subqueries, Subqueries in FROM clause, Scalar and correlated subqueries), WITH Clause, Hierarchical retrieval.
Unit - III	**PLSQL :** Introduction, Overview and benefits of PL/SQL, Subprograms, types of PL/SQL blocks, Simple Anonymous Block, Identifiers, types of identifiers, Declarative Section, variables, Scalar Data Types, The % Type attribute, bind variables, sequences in PL/SQL expressions, Executable statements, PL/SQL block syntax, comment the code, deployment of SQL functions in PL/SQL, Convert Data Types, newted blocks, operators. Interaction with the oracle server, Invoke SELECT Statements in PL/SQL, SQL cursor concept, Data Manipulation in the Server using PL/SQL, SQL Cursor Attributes to obtain Feedback on DML, Save and discard transactions.
Unit- IV	**Control Structures :** Conditional processing using IF statements and CASE statements, Loop Statement, while loop statement, for loop statement, the continue statement composite data types : PL/SQL records, The % ROWTYPE attribute, insert and update with PL/SQL records, INDEX by tables, INDEX BY Table Methods, Use INDEX BY Table of Records, Explicit Cursors, Declare the Cursor, Open the

	Cursor, Fetch data from the Cursor, Close the Cursor, Cursor FOR loop, The % NOTFOUND and % ROWCOUNT Attributes, the FOR UPDATE Clause and WHERE CURRENT Clause, Exception Handling, Handle Exceptions with PL/SQL, Trap Predefined and non-predefined Oracle Server Errors, User - Defined Exceptions, Propagate Exceptions, RAISE_APPLICATION_ERROR Procedure.
Unit-V	**Stored Procedures :** Create a Modularized and Layered Subprogram Design, the PL/SQL Execution Environment, differences between Anonymous Blocks and Subprograms, Create, Call, and Remove Stored Procedures, Implement Procedures Parameters and Parameters Modes, View Procedure Information, Stored Functions and Debugging Subprograms, Create, Call, and Remove a Stored Function, advantages of using Stored Functions, the steps to create a stored function, Invoke User-Defined Functions in SQL Statements, Restrictions when calling Functions, Control side effects when calling Functions, View Functions Information, debug Functions and Procedures, Packages, advantages of Packages, components of a Package, Develop a Package, enable visibility of a Package's Components, Create the Package Specification and Body using the SQL CREATE Statement and SQL Developer, Invoke the Package Constructs, View the PL/SQL Source Code using the Data Dictionary, Deploying Packages, Overloading Subprograms in PL/SQL, Use the STANDARD Package, Use Forward Declarations, Implement Package Functions in SQL and Restrictions, Persistent State of Packages, Persistent State of a Package Cursor, Control side effects of PL/SQL Subprograms, Invoke PL/SQL Tables of Records in Packages
Unit-VI	**Dynamic SQL :** The Execution Flow of SQL, Declare Cursor Variables, Dynamically Executing a PL/SQL Block, Configure Native Dynamic SQL to Compile PL/SQL Code, invoke DBMS_SQL Package, Implement DBMS_SQL with a Parameterized DML Statement, Dynamic SQL Functional Completeness, Triggers, the Triggers, Create DML Triggers using the CREATE TRIGGER Statement and SQL Developer, Identify the Trigger Event Types, Body, and Firing (Timing), Statement Level Triggers and Row Level Triggers, Create Instead of and Disabled Triggers,

	Manage, Test and Remove Triggers. Creating Compound, DDL and Event Database Triggers, Compound Trigger Structure for Tables and Views, Compound Trigger to Resolve the Mutating Table Error, Comparison of Database Triggers and Stored Procedures, Create Triggers on DDL Statements, Create Database-Event and System-Events Triggers, System Privileges Required to Manage Triggers

Books:

Murach's Oracle SQL and PLSQL by Joel Murach, Murach and Associates.

Oracle Database 11g PL/SQL Programming Workbook, ISBN : 9780070702264, By : Michael McLaughlin, John Harper, Tata McGraw-Hill.

Reference :

Oracle PL/SQL Programming, Fifth Edition By Steven Feuerstein, Bill Pribyl

Oracle 11g : SQL Reference Oracle press

Oracle 11g : PL/SQL Reference Oracle Press.

Expert Oracle PL/SQL, By : Ron Hardman, Michael McLaughlin, Tata McGraw-Hill

Oracle database 11g : hands on SQL/PL SQL by Satish Asnani (PHI) EEE edition

Unit - I

1

BASIC STRUCTURED QUERY LANGUAGE – I

Unit Structure

1.0 OBJECTIVES

After going through this chapter, you will be able to

- Understand RDBMS
- What SQL can do?
- Classify SQL elements
- Classify SQL Commands
- Understand common data types in SQL
- Create and Manage Tables
- Apply Constraints to Tables

1.1 INTRODUCTION

The amount of information available to us is literally exploding, and the value of data as an organizational asset is widely recognized. To get the most out of their large and complex datasets, users require tools that simplify the tasks of managing the data and extracting useful information in a timely fashion. Otherwise, data can become a liability, with the cost of acquiring it and managing it far exceeding the value derived from it.

A database management system, or DBMS, is software designed to assist in maintaining and utilizing large collections of data. The collection of data, usually referred to as database, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient.

A **data model** is a collection of high-level data description constructs that hide many low-level storage details. A DBMS allows a user to define the data to be stored in terms of a data model. Most database management systems today are based on the **relational data model**. A relational database consists of a collection of tables (mathematical concept of relation). A row in a table represents a relationship among a set of values. Informally, a table is an entity set, and a row is an entity.

The relational model is very simple and elegant: a database is a collection of one or more *relations,* where each relation is a table with rows and columns. This simple tabular representation enables even novice users to understand the contents of a

database, and it permits the use of simple, high-level languages to query the data.

1.2 What is RDBMS?

RDBMS stands for Relational Database Management System. RDBMS is the basis of SQL, and for all modern database system like MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access. It is a software system that manages the storage of data contained in a database. The relational model was first proposed by Dr. E. F. Codd in 1970. The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries and it consists of columns and rows.

1.2.1 Concepts of Relational Database

Relational Database is a structure consisting of a set of objects related together and data integrity methods which aims in the efficient storage of electronic data. In other words, it is a place to store the data, a way to create and retrieve the data, and a way to make sure that the data is logically consistent. The objects can be of various types like tables, indexes, queries etc.

1.3 INTRODUCTION TO SQL

SQL stands for "**Structured Query Language**" and can be pronounced as "SQL" or "sequel – (Structured English Query Language)". It is a query language used for accessing and modifying information in the database. It has become a Standard Universal Language used by most of the relational database management systems (RDBMS). SQL is tied very closely to the relational model. Few of the SQL commands used in SQL programming are SELECT Statement, UPDATE Statement, INSERT INTO Statement, DELETE Statement, WHERE Clause, ORDER BY Clause, GROUP BY Clause, ORDER Clause, Joins, Views, GROUP Functions, Indexes etc.

In a simple manner, SQL is a non-procedural, English-like language that processes data in groups of records rather than one record at a time. Few functions of SQL are:

- store data
- modify data
- retrieve data
- delete data
- create tables and other database objects

The SQL language has several parts:

Data-Definition Language (DDL): The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.

Interactive Data-Manipulation Language (DML): The SQL DML includes a query language based on both the relational algebra and the tuple relational calculus. It also includes commands to insert into tuples, delete tuples from, and modify tuples in the database.

View Definition: The SQL DDL includes commands for defining views.

Transaction Control: SQL includes commands for specifying the beginning and ending of transactions.

Embedded SQL and Dynamic SQL: Embedded and Dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, Java, PL/I, Cobol, Pascal, and Fortran.

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.

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

1.4 WHAT CAN SQL DO?

SQL can execute queries against a database

1. SQL can retrieve data from a database
2. SQL can insert records in a database
3. SQL can update records in a database
4. SQL can delete records from a database
5. SQL can create new databases
6. SQL can create new tables in a database
7. SQL can create stored procedures in a database
8. SQL can create views in a database
9. SQL can set permissions on tables, procedures, and views

1.5 SQL LANGUAGE ELEMENTS

The SQL language is subdivided into several language elements, including:

- **Clauses**, which are constituent components of statements and queries.

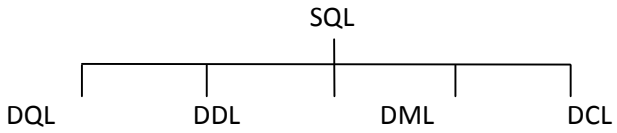
- **Expressions**, which can produce either scalar values or tables consisting of columns and rows of data.
- **Predicates**, which specify conditions that can be evaluated to SQL three-valued logic (3VL) or Boolean (true/false/unknown) truth values and which are used to limit the effects of statements and queries, or to change program flow.
- **Queries**, which retrieve the data based on specific criteria. This is the most important element of **SQL**.
- **Statements**, which may have a persistent effect on schemata and data, or which may control transactions, program flow, connections, sessions, or diagnostics.
  - SQL statements also include the semicolon (";") statement terminator.
- **Insignificant whitespace** is generally ignored in SQL statements and queries, making it easier to format SQL code for readability.

The following table shows the most common data types

Data Type	**Description**
NUMBER	Holds numeric data of any precision. The precision can range from 1 to 38.
CHAR(W)	Holds fixed length alphanumeric data up to w width
VARCHAR2(W)	Holds variable length alphanumeric data up to w width. A varchar2 value can contain up to 4000 bytes of data
DATE	DATE is a data type used to store date and time values in a 7-byte structure
RAW(size)	Raw binary data of varying length(size) bytes. Maximum size is 2000 bytes
TIMESTAMP	Timestamp is an extension of the Date data type that can store date and time data

1.6 CLASSIFICATION OF SQL COMMANDS

In order to learn SQL we will have to learn various commands or statements supported by SQL language. All the SQL commands can be broadly classified into five categories as shown in the figure below.



1.6.1 Data Query Language (DQL)

These commands are used to view records from tables. The actual data in the form of tables is stored onto the hard disk and DQL commands enable us to view the information/records stored in the database, for example, using the SELECT command.

1.6.2 Data Definition Language (DDL)

DDL is a subset of SQL commands that allows you to make changes to the structure of the database. The DDL part of SQL permits database tables to be created or deleted. It also defines indexes (keys), specifies links between tables, and imposes constraints between tables. Table structure refers to the number of columns and the data types of columns and constraints which can be applied or revoked as desired, using commands like CREATE, DROP, ALTER.

1.6.3 Data Manipulation Language (DML)

DML describes the portion of SQL that allows you to manipulate or control your data. For adding new records, removing existing records or changing values of existing records we will use DML statements like INSERT, DELETE, UPDATE.

1.6.4 Data Control Language (DCL)

Database Management Systems are multi user oriented. Each user has his own assigned area where his data is stored. By default one user cannot view, modify or delete the information stored in another user's area. The owner can grant the privilege of modifying the data owned by him to another user. These commands control how and to what extent one user can view, modify and delete information in another user's area using commands like GRANT, REVOKE.

1.6.5 Transaction Control Language (TCL)

Each operation which is done on database is called a Transaction. Transaction can be addition of new record, modification in values of existing records or deletion of existing records. Each transaction can be done permanently or can be

undone by using the transaction control statements, like COMMIT, ROLLBACK, SAVEPOINT.

1.7 CREATING AND MANAGING TABLES

Structured Query Language (SQL) is the language used to manipulate relational databases. In the relational model data is stored in structures called relations or tables. A table is the fundamental building block of a database application. Tables are used to store data on various entities like employees, products, customers, orders, sales etc.

There are a few things that one should note before typing and executing SQL commands:

- Commands may be on a single line, or many lines.
- For the sake of readability, place different clauses on separate lines and make use of tabs and indents.
- SQL command words cannot be split or abbreviated.
- SQL commands are not case sensitive.
- Place a semicolon(;) at the end of the last clause.

1.7.1 How to create a table?

Data Definition Language statements mentioned in the previous chapter allows you to make changes to the structure of the database. The create table statement is used to create a table object.

Syntax:

CREATE TABLE <tablename>

(<attribute1 name> <data type> (size),

<attribute2 name> <data type> (size),);

For example to create a STUDENT table, the command will be as follos:

```
CREATE TABLE student
    (
      rollno char(4),
      name varchar2(10),
      date_of_birth date,
      marks number(3)
    );
```

Rules for creating tables are as follows:

- The name of the table can be of 30 characters at the maximum.
- Alphabets from A to Z and 0 to 9 are allowed.
- Special characters like under score (_) are allowed.
- SQL reserved keywords like SELECT, WHERE etc are not allowed.

1.8 APPLYING DATA CONSTRAINTS

To maintain database integrity we need to enforce some rules on data being stored in a table. There are several ways of controlling what kind of data can be input into a table. The various controls/constraints are as follows:

- PRIMARY KEY
- NOT NULL
- UNIQUE
- DEFAULT
- CHECK
- FOREIGN KEY

1.8.1 Classification of Constraints

Constraints can be broadly classified as:

- Column Level
- Table Level

Column Level Constraints

When a constraint is applied to a single column then it is called a Column Level Constraint. This constraint will affect the values being entered for that particular column only irrespective of values of other columns.

Table Level Constraints

When a single constraint is applied to more than one column then it is called Table Level Constraint. These constraints impact the values being entered for a combination of column values, for example, salary can never be less than the commission is a table level constraint.

1.8.2 Primary Key

Primary key can be defined as the single column or combination of columns which uniquely identify a row in a table, for example, the rollno column in the “student” table is the primary key. No two students can have the same rollno. The following are the features of a primary key:

- A table can have one and only one primary key.
- Primary key implies UNIQUE as well as NOT NULL values,
- UNIQUE implies duplicate values cannot be entered.
- NOT NULL implies value cannot be unknown (NULL) for any of the records.
- Primary key columns are automatically indexed.
- A Primary Key can contain more than 1 column known as a Composite Key.

Primary Key = UNIQUE + NOT NULL + Automatic Indexed

An example of Primary key constraint:

```
CREATE TABLE student
    (
```

```
  rollno CHAR(4)      PRIMARY KEY,
  name VARCHAR2(10),
  date_of_birth DATE,
  marks NUMBER(3)
);
```

OR

```
CREATE TABLE student
(
  rollno char(4),
  name varchar2(10),
  date_of_birth date,
  marks number(3),
  CONSTRAINT stud_pk PRIMARY KEY (rollno)
);
```

An example of **Composite key constraint:**

```
CREATE TABLE student
(
  rollno char(4),
  name varchar2(10),
  date_of_birth date,
  marks number(3),
  CONSTRAINT stud_ck PRIMARY KEY (rollno, name)
);
```

1.8.3 NOT NULL

Specifying NOT NULL as constraint means that NULL values cannot be inserted for that particular field although duplicate

values for that column can be entered. It is a type of Domain Integrity Constraint.

An example of NOT NULL constraint:

```
CREATE TABLE student
    (
      rollno char(4),
      name varchar2(10)        NOT NULL,
      date_of_birth date NOT NULL,
      marks number(3)
    );
```

1.8.4 UNIQUE

UNIQUE implies duplicate values for the same field cannot be entered but NULL values can be inserted for that column. Since two NULL values cannot be compared, UNIQUE constraint does not prevent from supplying NULL values. It is a type of Domain Integrity Constraint.

An example of UNIQUE constraint:

```
CREATE TABLE student
    (
      rollno char(4)        UNIQUE,
      name varchar2(10)        NOT NULL,
      date_of_birth date NOT NULL,
      marks number(3)
    );
```

1.8.5 DEFAULT

DEFAULT implies that if we do not specify a value for a column in the INSERT statement, then the value specified in the Default clause will get inserted.

An example of DEFAULT constraint:

```
CREATE TABLE student
    (
      rollno char(4)        UNIQUE,
      name varchar2(10)         NOT NULL,
      date_of_birth date NOT NULL,
      marks number(3)   DEFAULT 0
    );
```

1.8.6 CHECK

The CHECK constraint allows verifying the values being supplied against specified conditions. For example, marks cannot be negative and also cannot exceed the maximum marks (i.e. marks should be between 0 and 100).

An example of CHECK constraint:

```
CREATE TABLE student
  (
    rollno char(4)      UNIQUE,
    name varchar2(10)       NOT NULL,
    date_of_birth date      NOT NULL,
    marks number(3)CHECK (marks >= 0 AND marks <= 100)
        );
```

1.8.7 FOREIGN KEY

The FOREIGN KEY constraint is used to define a foreign key which represents relationships between tables. The foreign key is used to enforce referential integrity. Features of foreign key are:

- A foreign key can refer to a primary key.

- A table can have multiple foreign keys referring to different tables for different columns.
- Defining foreign key establishes a PARENT/CHILD relationship between the two tables.
- Defining foreign key implies two conditions:
    - We can insert only those records in the CHILD table for which corresponding records exist in the PARENT table.
    - We can delete only those records from the PARENT table for which there are no corresponding records in the CHILD table.

An example of FOREIGN KEY constraint:

```
CREATE TABLE stud_detail
        (
          rollno char(4)        PRIMARY KEY,
          name varchar2(10)         NOT NULL,
          address date        NOT NULL,
          phone_no number(3) UNIQUE NOT NULL,
          CONSTRAINT std_fk FOREIGN KEY (rollno)
REFERENCES student (rollno)
        );
```

If you want to delete records from the PARENT table for which there are corresponding records in the CHILD table then you need to specify the ON DELETE CASCADE option while defining the Foreign Key relation. If you do not specify this option then the database server will not allow you to delete data if other table references it. If you specify this option then the corresponding records of the CHILD table will also be deleted with the records from the PARENT table.

An example of FOREIGN KEY constraint using ON DELETE CASCADE option:

```
CREATE TABLE stud_detail
```

```
(
  rollno char(4)        PRIMARY KEY,
  name varchar2(10)         NOT NULL,
  address date        NOT NULL,
  phone_no number(3) UNIQUE NOT NULL,
  CONSTRAINT std_fk FOREIGN KEY (rollno)
REFERENCES student (rollno) ON DELETE CASCADE
);
```

1.8.8 Adding, Removing and Altering Constraints

The constraints we applied at the time of table creation can also be specified after the table has been created (without constraints).

```
CREATE TABLE student
(
  rollno char(4),
  name varchar2(10),
  date_of_birth date,
  marks number(3)
);
```

The above table has been created without any constraints. An example for adding constraints after table creation:

```
ALTER TABLE student
    ADD PRIMARY KEY (rollno, name);
ALTER TABLE student
    ADD CONSTRAINT def_mark marks DEFAULT 0;
```

Suppose we do not want roll no and name to be the primary key in the above mentioned table. The constraint can be removed using

the ALTER TABLE command. An example for removing constraints:

```
ALTER TABLE student
    DROP PRIMARY KEY;
```

1.8.9 Enabling and Disabling Constraints

It is not a good programming practice to remove constraints. Removing constraints can lead to problems in the table like data redundancy. Instead of dropping or removing constraints we can temporarily disable constraints. We can disable constraints when we need to and then enable the constraints when required. Consider the following example about the table STUDENT which has the primary key as ROLL NO. We can disable the primary key constraint using the DISABLE CONSTRAINT command with the ALTER TABLE statement and in order to enable the constraint we need to use the ENABLE CONSTRAINT command with the ALTER TABLE statement.

```
CREATE TABLE student
    (
      rollno char(4),
      name varchar2(10),
      date_of_birth date,
      marks number(3),
      CONSTRAINT stud_pk PRIMARY KEY (rollno)
    );
```

Disabling Constraints

```
ALTER TABLE student DISABLE CONSTRAINT stud_pk;
```

Enabling Constraints

```
ALTER TABLE student ENABLE CONSTRAINT stud_pk;
```

1.9 SUMMARY

- A database management system, or DBMS, is software designed to assist in maintaining and utilizing large collections of data.
- The collection of data, usually referred to as database, contains information relevant to an enterprise.
- A relational database consists of a collection of tables.
- Relational Database is a structure consisting of a set of objects related together and data integrity methods which aims in the efficient storage of electronic data.
- SQL stands for “Structured Query Language” and is a query language used for accessing and modifying information in the database.
- SQL can execute queries against a database to store, retrieve, modify and delete data.
- SQL commands can be categorized into Data Query Language (DQL), Data Definition Language (DDL), Data Manipulation Language (DML), Data Control Language (DCL) and Transaction Control Language (TCL).
- A table is the fundamental building block of a database application. Tables are used to store data on various entities like employees, products, customers, orders, sales etc.
- The CREATE TABLE statement is used to create a table object.
- The various constraints that can be applied on a table are as follows:
  - PRIMARY KEY
  - NOT NULL
  - UNIQUE
  - DEFAULT
  - CHECK
  - FOREIGN KEY
- The structure of a table can be changed after its creation using the ALTER TABLE statement.
- Constraints can also be applied to table after its creation using the ALTER TABLE statement.
- Constraints can be enabled or disabled using the ENABLE/DISABLE CONSTRAINT statement.

1.10 REVIEW QUESTIONS

(1) What is SQL? What can be done using SQL?
(2) List and Explain the common data types in SQL.
(3) List and Explain the different elements in SQL language.
(4) Explain the parts of SQL language?
(5) Explain the classification of SQL statements?
(6) List the rules for creating a table? How can you create a table?
(7) List and Explain the different types of constraints with examples?
(8) Explain the commands used to alter table?

1.11 LAB ASSIGNMENT

1. Create table DEPT with the following columns and constraints

Column name	Data type	Size	Constraint
DEPTNO	NUMBER	2	PRIMARY KEY
DNAME	VARCHAR2	10	UNIQUE + NOT NULL
LOCATION	VARCHAR2	10	UNIQUE + NOT NULL

2. Create table EMPLOYEE with the following columns and constraints

Column name	Data type	Size	Constraint
EMPNO	CHAR	4	PRIMARY KEY
ENAME	VARCHAR2	10	NOT NULL
JOB	VARCHAR2	10	
MGR	CHAR	4	
HIREDATE	TIMESTAMP		NOT NULL
GENDER	CHAR	1	'M' OR 'F' ONLY
SAL	NUMBER	8,2	DEFAULT 0
COMM	NUMBER	8,2	DEFAULT 0
DEPTNO	NUMBER	2	FOREIGN KEY REFERRING TO DEPTNO of DEPT table

3. Insert 5 records in both the tables.

4. Add table level constraint such that commission cannot be greater than 30% of salary after the table has been created. Assign the constraint name COMM_30_SAL.

5. Add new constraint with the name DEPT_CHK_LOCATION to DEPT table such that LOCATION can be any one of the following cities MUMBAI, PUNE, BENGALURU, LONDON, SAN FRANSISCO only.

6. Remove the UNIQUE constraint from the LOCATION column.

1.12 BIBLIOGRAPHY, REFERENCES AND FURTHER READING

- Database Management Systems, Third Edition by RamaKrishnan, Gehre. McGraw Hill
- Database System Concepts, Fifth Edition by Silberschatz, Korth, Sudarshan. McGraw Hill
- Murach's Oracle SQL and PL/SQL by Joel Murach. Shroff Publishers & Distributors
- Oracle Database 11g by Satish Asnani. PHI Learning Private Limited
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, Tata McGraw-Hill
- SQL, PL/SQL The programming language of Oracle, Bayross Ivan, BPB Publications
- Fundamentals of Database Systems, Elmasri Ramez and Navathe B. Shamkant, Pearson

1.13 ONLINE REFERENCES

Wikipedia Link
http://en.wikipedia.org/wiki/SQL

Oracle Database PL/SQL language Reference 11g Release 2 (11.2), part number E25519-05
http://docs.oracle.com/cd/E11882_01/appdev.920/a96590/adg09dyn.htm

2

BASIC STRUCTURED QUERY LANGUAGE – II

Unit Structure

2.0 OBJECTIVES

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

- Retreive data using SELECT statement
- Restrict and Sort Data
- Manipulate Data using Single Row functions
- Display Data from Multiple Tables using Joins

2.1 INTRODUCTION

In the previous chapter, we created tables using CREATE TABLE statement and managed tables. In this chapter, we will manipulate data from a single table and also retreive data and join multiple tables.

2.2 BASIC DATA RETREIVAL

The **SELECT** statement retreives data from the database and returns in the form of query results.

The syntax is:

SELECT [ALL/DISTINCT] select-item

FROM table-specification

[WHERE search-condition]

[GROUP BY grouping-column]

[HAVING search-condition]

[ORDER BY sort-specification]

The SELECT and FROM clauses are required and remaining four clauses are optional.

SELECT statement forms a part of the Data Query Language (DQL). SELECT statement is used to query the database in order to find data. SELECT is the most complex statement in SQL, with optional keywords and clauses that include:

- The FROM clause which indicates the table(s) from which data is to be retrieved. The FROM clause can include optional JOIN subclauses to specify the rules for joining tables.

- The WHERE clause includes a comparison predicate, which restricts the rows returned by the query. The WHERE clause eliminates all rows from the result set for which the comparison predicate does not evaluate to true.
- The GROUP BY clause is used to project rows having common values into a smaller set of rows. GROUP BY is often used in conjunction with SQL aggregation functions or to eliminate duplicate rows from a result set. The WHERE clause is applied before the GROUP BY clause.
- The HAVING clause includes a predicate used to filter rows resulting from the GROUP BY clause. Because it acts on the results of the GROUP BY clause, aggregation functions can be used in the HAVING clause predicate.
- The ORDER BY clause identifies which columns are used to sort the resulting data, and in which direction they should be sorted (options are ascending or descending). Without an ORDER BY clause, the order of rows returned by an SQL query is undefined.

To illustrate the SQL commands in this chapter we consider the records in the following tables

Student Table:

Rollno	Name	Course_no	Grade
201	Rohit	SC01	A
209	Raj	SC01	B
325	Rita	COM02	A
355	Parag	SC02	A
365	Mohini	SC03	C

Stud_details Table:

Rollno	Name	Age	Address
201	Rohit	25	MG road, Goregaon
209	Raj	27	Thakur City, Kandivali
310	Manisha	24	Vasai(W)
325	Rita	26	Borivali(W)
355	Parag	23	Thane(E)
365	Mohini	22	Aarey Colony, Goregaon

Course Table:

Course_no	Course_name	Major
SC01	B.Sc IT	Information Technology
SC02	B.Sc CS	Computer Science
SC03	B.Sc Maths	Mathematics
COM01	B.Com	Accounts
COM02	B.M.S	Management Studies

We need to execute the following queries

- To display all student records from table student

SELECT * FROM Student;

- To display name and address of the Students

SELECT Name, Address FROM Stud_Details;

- To display all course details

SELECT * FROM Course;

2.2.1 Column Aliases

The column headings are by default based on the column name, but sometimes we may want a customized column heading. It can be done using Column Aliases.

Syntax:

SELECT <column name1> “column alias1”, <column name2> “column alias2”, <column name3> “column alias3”.....

FROM <tablename>;

Rules for declaring Column Aliases

10. Column aliases must not contain any whitespace.
11. The case is ignored.
12. The Alias must be enclosed in double quotes.

2.2.2 Duplicate Rows

Many columns in a table can contain duplicate data. When querying data the duplicate data can come more than once in the query solution. For example, in an EMPLOYEE table the names of a few employees can same or repeated. Duplicate rows can be avoided by simply using the keyword **DISTINCT.**

Syntax:

SELECT DISTINCT <column name> FROM <table name>;

For Example:

SELECT DISTINCT Course_no, Name “Student Name”

FROM Student;

2.3 RESTRICTING AND SORTING DATA

A simple **SELECT** statement will return all rows from a particular table, but if we need to list a record on the basis of some conditions we need to use the **WHERE** clause.

Rules of using a WHERE clause:

- The WHERE clause MUST appear after the FROM clause.
- WHERE clause consists of the keyword WHERE followed by a search condition that specifies the rows to be retrieved.
- Character strings and date values are enclosed with single quote.
- Character values are case sensitive and date values are format sensitive.
- An column alias cannot be used in the WHERE clause.

Consider the following queries

- To display all records of students whose grade = 'A'
  SELECT * FROM Student WHERE Grade = 'A';

- To display records of students who have enrolled for course 'SC02'
  SELECT * FROM Student

  WHERE Course_no = 'SC02';

Apart from using the WHERE clause for equality conditions, we can use it for other conditions like:

Comparison Test (<>, <, >, <=, >=)

Compares the value of one expression to the value of another expression.

Syntax: SELECT <list> FROM <table name> WHERE <column name> comparison-operator <value>;

Range Test (BETWEEN)

Tests whether the value of an expression falls within a specified range of values.

Syntax: SELECT <list> FROM <table name> WHERE <column name> BETWEEN <lower-limit> AND <higher-limit>

Set Membership Test (IN)

Tests whether a data value matches one of a list of target values.

Syntax: SELECT <list> FROM <table name> WHERE <column name> IN (list of constants separated by comma);

NULL Value Test (IS NULL)

Used to check explicitly for NULL values in a search condition.
Syntax: SELECT <list> FROM <table name> WHERE <column name> IS NULL;

Pattern Matching Test (LIKE)

Checks to see whether the data value in a column matches a specified pattern. The pattern is a string that may include one or more wildcard characters.

Symbol	**Represents**
%	It matches any sequence of zero or more characters.
_	The under score matches any single character.
[]	Any single character within the specified range ([a-f]).
[^]	Any single character not within the specified range ([^a-f]).

2.3.1 Ordering Data

When SELECT statement is executed, the order of rows returned in the output is undefined. To define a specific order in the output, we need to use the **ORDER BY** clause with the SELECT statement. Using the ORDER BY clause, records can be arranged

in a specified sequence may be alphabetically or by value. The ORDER BY clause needs to be added to the end of the SELECT statement.

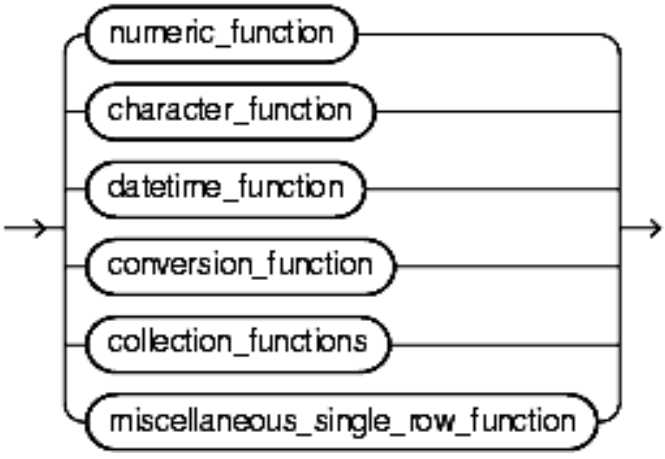
Syntax: SELECT <list> FROM <table name> ORDER BY <column name> [ASC | DESC]

2.4 DUAL TABLE

The Dual table is a special one-row table present by default in all Oracle database installations. It is suitable for use in selecting a pseudo-column such as SYSDATE or USER. The table has a single VARCHAR2(1) column called DUMMY that has a value of “X”. The Dual table can also be used to understand the functioning of various single row functions. In the next section we will learn about single row functions using the DUAL table.

2.5 SINGLE ROW FUNCTIONS

Single row functions, as the name suggests, operates on single row and returns a single result row for every row of a queried table. These functions can appear in SELECT lists, WHERE clause, and other SQL statements. There are various types of single row functions as shown in the diagram below.



2.5.1 Numeric Functions:

Numeric functions accept numeric values as input and return numeric values as output.

ABS: The ABS function calculates the absolute value of an expression. Since, the absolute value of a real number is its numeric value without regard to its sign (for example, 3 is the absolute value of both 3 and -3), this function always returns a positive value.

Syntax: ABS(*expression*)

Usage: SELECT ABS (expression/column name) FROM <table name>;

CEIL: The CEIL function returns the smallest whole number greater than or equal to a specified number.

Syntax: CEIL(*n*)

Usage: SHOW CEIL(15.7);

RESULT: 16

FLOOR: The FLOOR function returns the largest whole number equal to or less than a specified number.

Syntax: FLOOR(*n*)

Usage: SHOW FLOOR(15.7);

RESULT: 15

ROUND: When a number is specified as an argument, the ROUND function returns the number rounded to the nearest multiple of a second number you specify or to the number of decimal places indicated by the second number.

Syntax: ROUND(*number_exp, roundvalue*)

Usage: SHOW ROUND(2/3, .1);

RESULT: 0.70

TRUNC: When you specify a number as an argument, the TRUNCATE function truncates a number to a specified number of decimal places.

Syntax: TRUNC (*number*, *truncvalue*)

Usage: SHOW TRUNC (15.79, 1);

RESULT: 15.7

2.5.2 Character Functions:

Character functions accept character values as input and can return character or numeric values as output.

LOWER: The LOWER function converts all alphabetic characters in a text expression into lowercase.

Syntax: LOWER(*text-expression*)

UPPER: The UPPER function converts all alphabetic characters in a text expression into uppercase.

Syntax: UPPER (*text-expression*)

CONCAT: CONCAT returns *char1* concatenated with *char2*. Both *char1* and *char2* can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The string returned is in the same character set as *char1*. Its data type depends on the data types of the arguments.

Syntax: CONCAT (char1, char2)

SUBSTR: The SUBSTR function return a portion of *char*, beginning at character *position*, *substring_length* characters long.

Syntax: SUBSTR (char, position, substring_length)

LENGTH: The LENGTH functions return the length of *char*.

Syntax: LENGTH (char)

2.5.3 DateTime Functions:

DateTime functions operate on date, timestamp and interval values.

CURRENT_DATE: CURRENT_DATE returns the current date in the session time zone, in a value in the Gregorian calendar of data type DATE.

Syntax: SELECT CURRENT_DATE FROM dual;

CURRENT_TIMESTAMP: CURRENT_TIMESTAMP returns the current date and time in the session time zone, in a value of data type TIMESTAMP WITH TIME ZONE.

Syntax: SELECT CURRENT_TIMESTAMP FROM dual;

Note: DateTime Functions have been covered in more detail in later chapters.

2.5.4 Conversion Functions:

Conversion function converts a value from one datatype to another datatype.

TO_CHAR: This function converts number or date data type to character data type.

Syntax: TO_CHAR (value, [format mask])

TO_DATE: This function converts string data type to date data type.

Syntax: TO_DATE (string, [format mask])

TO_NUMBER: This function converts string data type to number data type.

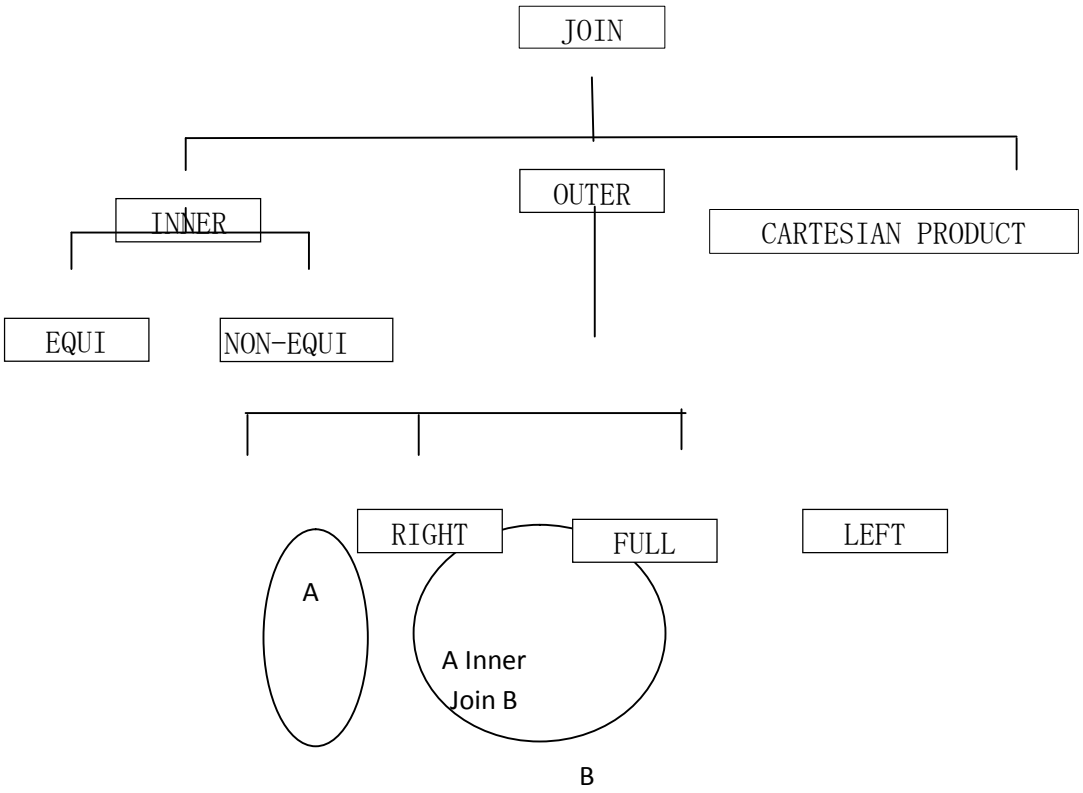
Syntax: TO_NUMBER (string, [format mask])

Note: For more information about different functions and the various categories of functions please refer to the Oracle Documentation available on the Oracle web site.

2.6 JOINS

Till now we have been selecting records from a single table at a time. However, in real life we do not work only on a single table but we need to select columns from two or more tables at a time for a single query. In order to combine two or more tables for a single query we have to use a concept called "**JOIN**" to acheive the desired result.

Joins can be classified as shown in the figure.



2.6.1 INNER EQUI JOIN

When two tables are joined using the EQUAL TO operator then that join is called Inner Equi Join.

2.6.2 INNER NON-EQUI JOIN

When two tables are joined using any comparison operator (<, >, <=, >=, !=) other than EQUAL TO operator then that join is called Inner Non-Equi Join.

Example for Inner Equi Join

We want to display the course_name and major subject alongwith the name and rollno of the student from the tables mentioned above as reference.

SELECT student.rollno, student.name, course.course_name, course.major

FROM student, course

WHERE student.course_no = course.course_no;

Rollno	Name	Course_name	Major
201	Rohit	B.Sc IT	Information Technology
209	Raj	B.Sc IT	Information Technology
325	Rita	B.M.S	Management Studies
355	Parag	B.Sc CS	Computer Science
365	Mohini	B.Sc Maths	Mathematics

Notice the “WHERE” clause specifying the join of two tables based on the matching course_no column is known as the JOIN condition. The Join condition may involve more than one column.

Above query can also be written as

SELECT student.rollno, student.name, course.course_name, course.major

FROM student INNER JOIN course

ON student.course_no = course.course_no;

Aliasing

Aliasing is the process whereby we can assign a short name to the tables being joined and use those alias names for preceding the column names. The above given Inner Equi Join can be written in the following manner:

SELECT s.rollno, s.name, c.course_name, c.major

FROM student s, course c

WHERE s.course_no = c.course_no;

Example for Inner Non-Equi Join

We want to display the course_name and major subject alongwith the name and rollno of the student whose age is greater than 25 from the tables mentioned above as reference.

SELECT s.rollno, s.name, c.course_name, c.major

FROM student s, course c

WHERE s.course_no = c.course_no AND s.age > 25;

Rollno	Name	Course_name	Major
209	Raj	B.Sc IT	Information Technology
325	Rita	B.M.S	Management Studies

2.6.3 Joining a Table to itself using Self Join

When a table is joined with its own then it is called “Self-join”. Although self joins are rare, some queries are best solved using self joins. In a self join instead of duplicating the table, SQL lets you refer to it by a different table alias.

Example of Self Join

SELECT s1.rollno, s1.name, s2.age, s2,address

FROM stud_details s1, stud_details s2

WHERE s1.rollno = s2.rollno;

Note: Although the same table has been used, SQL considers them as two different tables because separate aliases have been used.

2.6.4 Outer Joins

An outer join extends the result of an inner join. An outer join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other satisfy the join condition.

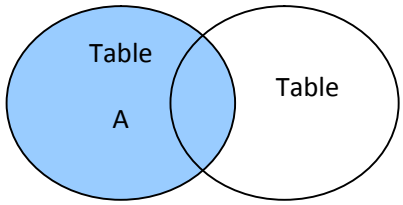
Outer Joins can be classified as

(9) Left Outer Join
(10) Right Outer Join
(11) Full Outer Join

2.6.5 LEFT OUTER JOIN

A query that performs a join on two tables A and B, and returns all rows from table A and only matching rows from table B is

called as LEFT OUTER JOIN. The result set contains all rows from the first or the left table and only matching records from the second table.



Syntax:

SELECT <list>

FROM <table name1> LEFT OUTER JOIN <table name2>

ON JOIN CONDITION

Example for Left Outer Join

Suppose we want to display the list of names and addresses of all students alongwith their course_no and grade.

SELECT sd.rollno, sd.name, sd.address, s.course_no, s.grade

FROM stud_details st LEFT OUTER JOIN student s

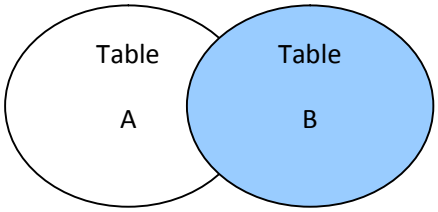
ON sd.rollno = s.rollno;

Rollno	Name	Address	Course_no	Grade
201	Rohit	MG road, Goregaon	SC01	A
209	Raj	Thakur City, Kandivali	SC01	B
310	Manisha	Vasai(W)	-	-
325	Rita	Borivali(W)	COM02	A
355	Parag	Thane(E)	SC02	A
365	Mohini	Aarey Colony, Goregaon	SC03	C

In the above result, all columns from the stud_details table have been listed and only matching columns from the student table have been listed.

2.6.6 RIGHT OUTER JOIN

A query that performs a join on two tables A and B, and returns all rows from table B and only matching rows from table A is called as RIGHT OUTER JOIN. The result set contains all rows from the second or the right table and only matching records from the first table.



Syntax:

SELECT <list>

FROM <table name1> RIGHT OUTER JOIN <table name2>

ON JOIN CONDITION

Example for Right Outer Join

Suppose we want to display the list of names of all students alongwith their course_name and major subject.

SELECT s.rollno, s.name, c.course_no, c.course_name, c.major

FROM student s RIGHT OUTER JOIN course c

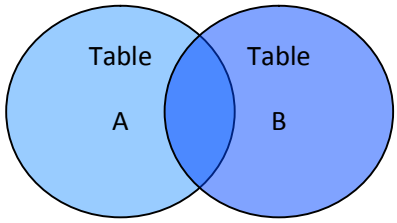
ON s.course_no = c.course_no;

Rollno	Name	Course_no	Course_name	Major
201	Rohit	SC01	B.Sc IT	Information Technology
209	Raj	SC01	B.Sc IT	Information Technology
325	Rita	COM02	B.M.S	Management Studies
355	Parag	SC02	B.Sc CS	Computer Science
365	Mohini	SC03	B.Sc Maths	Mathematics
-	-	COM01	B.Com	Accounts

In the above result, all columns from the course table have been listed and only matching columns from the student table have been listed.

2.6.7 FULL OUTER JOIN

A query that performs a join on two tables A and B, and returns all rows from table A and B alonwith the non-matching rows is called as FULL OUTER JOIN. The result set contains all rows from both the tables.



Syntax:

SELECT <list>

FROM <table name1> FULL OUTER JOIN <table name2>

ON JOIN CONDITION

2.6.8 CARTESIAN PRODUCT

A join in which each record from table 1 gets associated with each and every record of table 2 is called as Cartesian Product. A Cartesian product always generates many rows and is rarely useful. For example, the Cartesian product of two tables, each with 100 rows, has 10,000 rows.

Syntax:

SELECT <list>

FROM <table name1>, <table name2>

Note: In a cartesian product, no WHERE clause is used. Cartesian Product can also be called as CROSS JOIN.

2.7 SUMMARY

- The SELECT statement retreives data from the database and returns in the form of query results.

- Duplicate rows can be avoided by simply using the keyword DISTINCT.

- We need to use the WHERE clause in order to list a record on the basis of some conditions.

- WHERE clause can be used for
  - Equality conditions
  - Comparison Test
  - Range Test
  - Set Membership Test
  - NULL Value Test
  - Pattern Matching Test

- To define a specific order in the output, we need to use the ORDER BY clause with the SELECT statement.
- Single row functions operate on a single row and returns a single result row for every row of a queried table.There are various types of single row functions

  - Numeric Functions
  - Character Functions
  - DateTime Functions
  - Conversion Functions

- In order to combine two or more tables for a single query we have to use a concept called “JOIN” to acheive the desired result.

- Joins can be classified as
  - Inner - Equi & Non-Equi
  - Self
  - Outer - Left, Right, & Full
  - Cartesian Product

2.8 REVIEW QUESTIONS

- Explain the Where clause. List and explain the conditions under which you can use it.
- What is a single row function? Explain and give examples.
- What is a join? Explain the various categories of joins that can be used to join two tables?

2.9 LAB ASSIGNMENT

1. Create a CUSTOMER table with the following columns and constraints

Column name	Data type	Size	Constraint
CUSTOMER_ID	CHAR	6	PRIMARY KEY. MUST BEGIN WITH 'C'
CUSTOMER_NAME	VARCHAR2	20	NOT NULL
ADDRESS	VARCHAR2	20	UNIQUE
CITY	VARCHAR2	20	
PINCODE	NUMBER	6	
STATE	VARCHAR2	20	
BALANCE_DUE	NUMBER	8,2	

2. Create a PRODUCT table with the following columns and constraints

Column name	Data type	Size	Constraint
PRODUCT_CODE	CHAR	6	PRIMARY KEY
PRODUCT_NAME	VARCHAR2	20	UNIQUE
QTY_AVAIL	NUMBER	5	
COST_PRICE	NUMBER	8,2	
SELLING_PRICE	NUMBER	8,2	

3. Create a ORDER table with the following columns and constraints

Column name	Data type	Size	Constraint
ORDER_NO	CHAR	6	
ORDER_DATE	TIMESTAMP		
CUSTOMER_ID	CHAR	6	
PRODUCT_CODE	CHAR	6	
QUANTITY	NUMBER	5	

PRIMARY KEY = ORDER_NO + ORDER_DATE + CUSTOMER_ID + PRODUCT_CODE

- Insert 5-10 records in all the tables.
- Apply UNIQUE constraint on CUSTOMER_ID+ PRODUCT_CODE on the ORDER table.
- Define a foreign key on CUSTOMER_ID of ORDER table referring to CUSTOMER_ID of CUSTOMER table.
- Define a foreign key on PRODUCT_CODE of ORDER table referring to PRODUCT_CODE of PRODUCT table.
- List the customer names whose balance due is more than 4000/-.
- For all products display the product name along with Net Profit as Selling price – Cost price.
- Calculate the profit earned on each order.
- Display the orders placed during 2008.

[Hint: TO_CHAR(ORDER_DATE, 'YYYY') = '2008')

- Arrange the customers first by STATE and then by NAME.
- Display the customers from CUSTOMER table such that the customer with maximum due balance is displayed at the top.
- Find out the product names and their quantity which have been delivered in the month of February.
- Find the product name, customer name and total quantity purchased by various customers.
- Display the total sale amount and quantity for each customer. The report should display customer name instead of customer id.

2.10 BIBLIOGRAPHY, REFERENCES AND FURTHER READING

- Database Management Systems, Third Edition by RamaKrishnan, Gehre. McGraw Hill
- Database System Concepts, Fifth Edition by Silberschatz, Korth, Sudarshan. McGraw Hill
- Murach's Oracle SQL and PL/SQL by Joel Murach. Shroff Publishers & Distributors
- Oracle Database 11g by Satish Asnani. PHI Learning Private Limited
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, Tata McGraw-Hill
- SQL, PL/SQL The programming language of Oracle, Bayross Ivan, BPB Publications
- Fundamentals of Database Systems, Elmasri Ramez and Navathe B. Shamkant, Pearson

2.11 ONLINE REFERENCES

Wikipedia Link

http://en.wikipedia.org/wiki/SQL

Oracle Database PL/SQL language Reference 11g Release 2 (11.2), part number E25519-05

http://docs.oracle.com/cd/E11882_01/appdev.920/a96590/adg09dyn.htm

❖❖❖❖

3

ADVANCED QUERIES AND DATABASE OBJECTS

Unit Structure

3.0 OBJECTIVES

At the end of this chapter you will be able to:

- Aggregate Data using Group Functions
- Aggregate Data using Group By Having clause
- Create Database Objects like Views, Sequences, Indexes and Synonyms
- Use Subqueries

3.1 INTRODUCTION

In the previous chapter, you were introduced to scalar (single row) functions, which operate on a single value and return a single value. In this chapter, you'll learn how to code queries that summarize data.

3.2 AGGREGATE FUNCTIONS

Aggregate functions operate on a series of values and return a single summary value. Aggregate functions allow you to do jobs like calculate averages, summarize totals, or find the highest value for a given column. Aggregate functions can appear in select lists and in ORDER BY and HAVING clauses. They are commonly used with the GROUP BY clause in a SELECT statement, where the rows of a queried table are divided into groups.

Many (but not all) aggregate functions that take a single argument accept these clauses:

- DISTINCT – causes an aggregate function to consider only distinct values of the argument expression.
- ALL – causes an aggregate function to consider all values, including all duplicates.

For example, the DISTINCT average of 1, 1, 1, and 3 is 2. The ALL average is 1.5. If you specify neither, then the default is ALL. Null values are always excluded from these functions.

The table given below presents the syntax of the most common aggregate functions.

Syntax	**Description**
AVG ([ALL | DISTINCT] expression)	Returns the average of non-null values in the expression.
SUM ([ALL | DISTINCT] expression)	Returns the total of non-null values in the expression.
MIN ([ALL | DISTINCT] expression)	Returns the lowest non-null value in the expression.
MAX ([ALL | DISTINCT] expression)	Returns the highest non-null value in the expression.

COUNT ([ALL | DISTINCT] expression)	Returns the number of non-null values in the expression.
COUNT(*)	Returns the number of rows selected by the query.

Since the purpose of these functions are self-explanatory, we'll focus mainly on how to use them.

Assume records in the EMP table as shown below.

EMPNO	ENAME	HIREDATE	DEPTNO	GENDER	SALARY	COMM
111	Satish	19-DEC-2008	10	M	10,000	1000
222	Rashmi	01-JAN-1987	20	F	8000	550
333	Rishi	05-JUN-1976	10	M	7000	450
444	Anil	16-APR-1967	10	M	12,000	2000
555	Anita	-	30	F	8000	1000
666	Nilesh	20-MAY-1987	20	M	13,000	-
777	Ruchi	11-JUN-2000	30	F	5000	-
888	Sarika	-	20	F	4000	-

- Find the number of distinct departments from emp table.

```
SELECT COUNT(DISTINCT deptno) FROM emp;
```

Result:

COUNT(DISTINCT deptno)

3

- Find the number of employees in the emp table.

SELECT COUNT(*) FROM emp;

Result:

COUNT(*)

8

- Find the total salary and the average commission from the emp table.

SELECT SUM(salary), AVG(comm) FROM emp;

Result:

SUM(salary) AVG(comm)

67000 1000

3.3 GROUP BY HAVING CLAUSE

GROUP BY clause forms groups on the specified columns. The GROUP BY clause is used alongwith the aggregate functions to retreive data grouped according to one or more columns. The group by clause should contain all the columns to be displayed except those used alongwith the aggregate functions. The GROUP BY clause groups rows but does not guarantee the order of the result set. To order the groupings, use the ORDER BY clause.

For example,

- Display the number of employees from each department.

SELECT deptno, count(*)

FROM emp

GROUP BY deptno;

DEPTNO	COUNT(*)
10	3
30	2
20	3

- Display department wise total salary from the emp table.

SELECT deptno, sum(salary)

FROM emp

GROUP BY deptno;

DEPTNO	SUM(salary)
10	29,000
30	13,000
20	25,000

The **HAVING** clause to restrict the groups of returned rows to those groups for which the specified *condition* is TRUE. In other words, the HAVING clause is used to filter the records which a GROUP BY clause returns. This is similar to the WHERE clause but is used with the GROUP BY clause. The WHERE clause cannot be used with the GROUP BY clause.

For example,

(12) Display department wise total salary from the emp table such that only those departments are displayed where the total salary is greater than 20,000.

```
SELECT deptno, sum(salary)

FROM emp

GROUP BY deptno

HAVING SUM(salary) > 20000;
```

DEPTNO	SUM(salary)
10	29,000
20	25,000

(13) Display the number of employees from each department where the number of employees is equal to 2.

```
SELECT deptno, count(empno) “empcount”

FROM emp

GROUP BY deptno

HAVING COUNT(empno) = 2;
```

DEPTNO	EMPCOUNT
30	2

3.3.1 Comparing HAVING clause and WHERE clause

A WHERE clause in a SELECT statement that uses grouping & aggregates, the search condition is applied before rows are grouped and aggregates are calculated. That way, only rows that satisfy the condition are grouped. A HAVING clause in a SELECT statement that uses grouping & aggregates, the search condition is applied after rows are grouped and aggregates are calculated. That way, only groups that satisfy the condition are included in the result set.

A WHERE clause can refer to any column in the table. A HAVING clause can only refer to a column included in the SELECT clause.

A WHERE clause cannot contain aggregate functions. Aggregate functions can only be coded in the HAVING clause.

3.4 CREATING OTHER DATABASE OBJECTS

SQL allows you to create various database objects other than table like views, sequences, indexes, synonyms. We will understand these different database objects in this section.

Assume table **PRODUCT** with the following records

product_id	**product_name**	**company_name**	**unit_price**
100	Shampoo	Pantene	180
101	Deospray	Denim	400
102	Tooth paste	Colgate	150
103	Soap	Lux	75
104	Hair gel	Laureal	300

Assume table **ORDER** with the following records

order_id	product_id	total_units	Customer
O1	101	30	Lifestyle
O2	101	5	Shoppers stop
O3	103	25	Spencer
O4	101	10	Food bazaar
O5	103	200	Big bazaar

3.4.1 VIEWS

A **view** is a logical representation of one or more tables. In essence, a view is a stored query. A view derives its data from the tables on which it is based,called base tables. Base tables can be tables or other views. All operations performed on a view actually affect the base tables. You can use views in most places where tables are used. You can query, insert, update and delete from views. Views can be handled as any other table but they do not occupy any space.

Unlike a table, a view is not allocated storage space, nor does a view contain data. Rather, a view is defined by a query that extracts or derives data from the base tables referenced by the view. Because a view is based on other objects, it requires no storage other than storage for the query that defines the view in the **data dictionary**.

Benefits of using Views

Views enable you to tailor the presentation of data to different types of users. Views are often used to:

- Provide an additional level of table security by restricting access to a predetermined set of rows or columns of a table.

- Hide data complexity.

  For example, a single view can be defined with a **join**, which is a collection of related columns or rows in multiple tables. However, the view hides the fact that this information actually originates from several tables. A query might also perform extensive calculations with table information. Thus, users can query a view without knowing how to perform a join or calculations.

- Present the data in a different perspective from that of the base table.

  For example, the columns of a view can be renamed without affecting the tables on which the view is based.

- Isolate applications from changes in definitions of base tables.

  For example, if the defining query of a view references three columns of a four column table, and a fifth column is added to the table, then the definition of the view is not affected, and all applications using the view are not affected.

3.4.1.1 Classification of Views

Views can be classified as updateable views and non-updateable views.

3.4.1.2 Updateable Views

By updateable we mean to say that one can insert, update and delete records from the view. Actually all the DML operations are performed on the base table.

View with the following characteristics is called an updateable view.

- It is created from a single table.
- It includes all PRIMARY KEYS and NOT NULL columns of the base table.
- Aggregate functions like SUM, AVG have not been used.
- It should not have DISTINCT, GROUP BY, HAVING clauses.
- It must not use constants, strings or value expressions like salary * 2.
- It must not any function calls (e.g. RPAD, SUBSTR, etc.).
- If a view is defined from another view then that view must also be updateable.

3.4.1.3 Non-updateable Views

Non-updateable means we cannot insert, update and delete records from that view.

View with the following characteristics is called a non-updateable view.

7. It is created from more than one table.
8. It has DISTINCT, GROUP BY, HAVING clause. Even if view is derived from a single table but contains any of these clauses then it is not updateable.
9. It does not include all the PRIMARY KEYS and NOT NULL columns of base tables.

CREATING VIEWS

Syntax:

CREATE OR REPLACE VIEW <view name> AS

(SELECT query)

[WITH READ ONLY CONSTRAINT <constraint name>];

Create a view showing the details of the products which have been ordered by the customers.

CREATE OR REPLACE VIEW prod_ordered AS

SELECT product_name, company_name, total_units

FROM product INNER JOIN order

ON product.product_id = order.product_id;

To see the details of the products ordered

SELECT * FROM prod_ordered

product_name	company_name	total_units
Deospray	Denim	30
Deospray	Denim	5
Soap	Lux	25
Deospray	Denim	10
Soap	Lux	200

VIEW WITH READ ONLY CONSTRAINT

The WITH READ ONLY option allows the user to create a read-only view. You cannot use the DELETE, INSERT or UPDATE commands to modify data for the view.

To create a read only view “prodreadonly” for all products in the product table which have a unit_price more than 100.

CREATE or REPLACE VIEW prodreadonly AS

SELECT * FROM product WHERE unit_price > 100

WITH READ ONLY CONSTRAINT view_no;

Dropping VIEWS

To remove views we use the command DROP VIEW. You can change the definition of a view by dropping and re-creating it. If you delete a view, you can no more access the virtual tables based on the view.

Syntax:

DROP VIEW <view name>

3.4.2 INDEXES

Index is an object which can be defined as the ordered list of values of a column or combination of columns used for faster searching and sorting of data. An index speeds up joins and searches by providing a way for a database management system to go directly to a row rather than having to search through all the rows until it finds the one you want. By default, indexes are created for primary keys and unique constraints of a table. However, creating indexes must be avoided on columns that are updated frequently since this slows down insert, update and delete operations.

Creating Indexes

To create an index, you need to use the CREATE INDEX command. In addition, you can use the UNIQUE keyword to specify that an index contains only unique values.

Syntax:

CREATE [UNIQUE] INDEX index_name

ON table_name (column_name1 [ASC | DESC] [, column_name2 [ASC | DESC]].....)

Removing Indexes

To remove or delete an index, we need to use the DROP INDEX command.

Syntax:

DROP INDEX <index name>

3.4.3 SEQUENCES

SEQUENCE is a type of database object which can be used to generate numbers in a sequence. This can be used to generate values for primary keys.

Create Sequence

Sequences can be created using the CREATE SEQUENCE command which has the following syntax:

CREATE SEQUENCE <sequence name>

START WITH <integer-value>

INCREMENT BY <integer-value>

MAXVALUE <integer-value> OR NOMAXVALUE

MINVALUE <integer-value> OR NOMINVALUE

CYCLE OR NOCYCLE

CACHE OR NOCACHE

ORDER OR NOORDER

START WITH <integer-value>: specifies the 1st sequence number to be generated.

INCREMENT BY <integer-value>: The integer number by which sequence number should be incremented for generating the next number. If it is positive then values are ascending and if it is negative then values are descending. The default value is 1.

MAXVALUE <integer-value>: If the increment value is positive then MAXVALUE determines the maximum value up to which the sequence numbers will be generated.

NOMAXVALUE: Specifies the maximum value of 10^27 for an ascending sequence or -1 for a descending sequence.

MINVALUE <integer-value>: If the increment value is negative then MINVALUE determines the minimum value up to which the sequence numbers will be generated.

NOMINVALUE: Specifies the minimum value of 1 for an ascending sequence or -10^26 for a descending sequence.

CYCLE: Causes the sequences to automatically recycle to minvalue when maxvalue is reached for ascending sequences; for descending sequences, it causes to recycle from minvalue back to maxvalue.

NOCYCLE: Sequence numbers will not be generated after reaching the maximum value for ascending sequences or minimum value for descending sequences.

CACHE: Specifies how many values are pre-allocated in buffers for faster access. Default value is 20.

NOCACHE: Sequence numbers are not pre-allocated.

ORDER: Generates the number in a serial order.

NOORDER: Generates the number in a random order.

Initializing and Accessing Sequence

A sequence needs to be initialized before being used. Every sequence is initialized by a pseudo column NEXTVAL. Once you've created a sequence, you typically use it within an INSERT statement. The NEXTVAL pseudo column gets the next value from the sequence so it can be inserted into the table. The CURRVAL pseudo column is used to check the current value of the sequence.

Modifying Sequence

It may be required later on to change certain parameters of an already created sequence. This can be done using the ALTER SEQUENCE command.

Syntax:

ALTER SEQUENCE <sequence name>

[sequence attributes]

Dropping Sequence

A sequence can be deleted using the DROP SEQUENCE command.

Syntax:

DROP SEQUENCE <sequence name>

3.4.4 SYNONYMS

Synonyms are alternative names for an existing object which are permanently stored in the database. We have used alias names for accessing tables with shorter names in various queries. The difference is that these alias names are of temporary nature and are lost from one query to another, whereas synonyms are permanent alias names for objects.

Advantages of using synonyms

- Synonyms are often used for security and convenience.
- They can do the following things:
  - They can hide or mask the name and owner of an object.
  - Provide location transparency for remote objects of a distributed database.
  - Simplify SQL statements for database users.
- With the help of synonyms the user can insert, update or retreive records from any database object. Synonyms cannot be used in a DROP TABLE, DROP VIEW or TRUNCATE TABLE statement.

The various objects for which synonyms can be created are as follows:

- Tables
- Views
- Materialized Views
- Stored Function
- Stored Procedures
- Packages
- Sequences

- Synonyms

There are two kinds of synonyms – public and private.

Public Synonym: These are accessible to all users provided they have the appropriate object privilege on the object on which the synonym is created.

Private Synonym: These belong only to the user who creates it.

Creating Synonyms

Synonyms can be created using the CREATE SYNONYM command

Syntax:

CREATE [PUBLIC] SYNONYM <synonym name> FOR <object name>

Renaming Synonyms

Only Private Synonyms can be renamed using the Rename statement.

Syntax:

RENAME <old synonym name> TO <new synonym name>

Modifying a Synonym

To modify or alter or change a synonym use the OR REPLACE clause. You can use this clause to change the definition of an existing synonym without dropping it.

Syntax:

CREATE OR REPLACE [PUBLIC] SYNONYM <synonym name> FOR <object name>

Removing a Synonym

Synonyms can be removed or deleted using the DROP SYNONYM statement.

Syntax:

DROP SYNONYM <synonym name>

3.5 SUBQUERIES

Queries can be nested so that the results of one query can be used in another query via a relational operator or aggregation function. A nested query is known as a **subquery**. While joins and

other table operations provide computationally superior (i.e. faster) alternatives in many cases, the use of subqueries introduces a hierarchy in execution which can be useful or necessary.

Since you know how to code SELECT statements, you already know how to code a subquery. It's simply a SELECT statement that's coded within another SQL statement. A subquery can return a single value, a result set that contains a single column (single row subquery), or a result set that contains one or more columns (multiple row subquery).

The following is the list of comparison operators used in a single row subquery.

Symbol	**Description**
	Equal
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
<>	Not equal
!=	Not equal

The following is the list of comparison operators used in multiple row subquery.

Subquery Operator	**Description**
IN, =ANY	Look for a match in any of the subquery rows
>ANY	Look for a value greater than any of the subquery rows

<ANY	Look for a value less than any of the subquery rows
>ALL	Look for a value greater than all of the subquery rows
<ALL	Look for a value less than all of the subquery rows
!=ALL, NOT IN	Look for a value not present in any of the subquery rows

Four ways to introduce a subquery in a SELECT statement

- In a WHERE clause as a search condition.
- In a HAVING clause as a search condition.
- In a FROM clause as a table specification.
- In a SELECT clause as a column specification.

Examples for subqueries:

- Get the product name and company name of the products which have the maximum price.

```
SELECT product_name, company_name FROM product
WHERE unit_price = (SELECT MAX(unit_price) FROM product);
```

- Get the names of products which have a unit price less than or equal to the average price of the products.

```
SELECT product_name FROM product
WHERE unit_price <= (SELECT AVG(unit_price) FROM product);
```

- Get the names of the product which has the highest price.

```
SELECT product_name FROM product
WHERE unit_price >= ALL (SELECT unit_price FROM product);
```

- Get the names of the products which have been ordered.

```
SELECT product_name FROM product
WHERE product_id IN (SELECT product_id FROM order_prod);
```

Or

SELECT product_name FROM product

WHERE product_id = ANY (SELECT product_id FROM order_prod);

- Get the names of the products which have not been ordered by any customers.

SELECT product_name FROM product

WHERE product_id NOT IN (SELECT product_id FROM order_prod);

Or

SELECT product_name FROM product

WHERE product_id != ALL (SELECT product_id FROM order_prod);

- Get the names of the products which have been ordered in maximum quantity.

SELECT product_name FROM product

WHERE product_id IN (SELECT product_id FROM order_prod

GROUP BY product_id HAVING SUM(total_units) > = ALL

(SELECT SUM(total_units) FROM order_prod GROUP BY product_id));

3.5.1 SUBQUERY in DDL and DML commands

Subqueries can be used in DDL commands to create a new table from an existing table. The subquery is used to retreive the data using which the new table is created. The structure of the new table will be same as the structure of the query.

Subquery can be used to insert, update and delete rows from the existing table.

For example,

- To create a table student_new containing roll, name and grade of students enrolled in semester 3.

CREATE TABLE student_new AS

(SELECT roll, name, grade FROM student WHERE semester = 3);

- To insert a record in the table student_new (created above) with roll number 1 more than the maximum roll number of the table.

```
INSERT INTO student_new VALUES
((SELECT MAX(roll) + 1 FROM student_new), 'Pravin', 'A');
```

3.6 SUMMARY

- Aggregate functions operate on a series of values and return a single summary value.
- Aggregate functions are commonly used with the GROUP BY clause in a SELECT statement, where the rows of a queried table are divided into groups.
- The most common aggregate functions are
  - AVG
  - SUM
  - MIN
  - MAX
  - COUNT
- GROUP BY clause forms groups on the specified columns. The GROUP BY clause is used alongwith the aggregate functions to retreive data grouped according to one or more columns.
- The HAVING clause to restrict the groups of returned rows to those groups for which the specified condition is TRUE.
- A view is a logical representation of one or more tables. In essence, a view is a stored query.
- Views can be classified as updateable views and non-updateable views.
- Index is an object which can be defined as the ordered list of values of a column or combination of columns used for faster searching and sorting of data.
- By default, indexes are created for primary keys and unique constraints of a table.
- Creating indexes must be avoided on columns that are updated frequently since this slows down insert, update and delete operations.
- SEQUENCE is a type of database object which can be used to

generate numbers in a sequence. This can be used to generate values for primary keys.

- Synonyms are alternative names for an existing object which are permanently stored in the database.
- There are two kinds of synonyms – public and private.
- Queries can be nested so that the results of one query can be used in another query via a relational operator or aggregation function. A nested query is also known as a subquery.
- Subqueries can be used in DDL commands to create a new table from an existing table.
- Subquery can be used to insert, update and delete rows from the existing table.

3.7 REVIEW QUESTIONS

1. What are aggregate functions? explain in detail.
2. Explain the Group By Having clause with examples.
3. Differentiate between the WHERE clause and HAVING clause?
4. What is view? What are the benefits of using views?
5. Explain the types of views in detail.
6. Write a short note on indexes.
7. What is a sequence? Explain the syntax for creating a sequence.
8. What is a synonym? Why should you use a synonym?
9. Explain the different types of synonyms? List the objects for which synonyms can be created.
10. What is a subquery? Explain in detail with examples.

3.8 LAB ASSIGNMENT

1. Create a CUSTOMER table with the following columns and constraints

Column name	Data type	Size	Constraint
CUSTOMER_ID	CHAR	6	PRIMARY KEY. MUST BEGIN WITH 'C'
CUSTOMER_NAME	VARCHAR2	20	NOT NULL

ADDRESS	VARCHAR2	20	UNIQUE
CITY	VARCHAR2	20	
PINCODE	NUMBER	6	
STATE	VARCHAR2	20	
BALANCE_DUE	NUMBER	8,2	

2. Create a PRODUCT table with the following columns and constraints

Column name	Data type	Size	Constraint
PRODUCT_CODE	CHAR	6	PRIMARY KEY
PRODUCT_NAME	VARCHAR2		UNIQUE
QTY_AVAIL	NUMBER	5	
COST_PRICE	NUMBER	8,2	
SELLING_PRICE	NUMBER	8,2	

3. Create a ORDER table with the following columns and constraints

Column name	Data type	Size	Constraint
ORDER_NO	CHAR	6	
ORDER_DATE	TIMESTAMP		
CUSTOMER_ID	CHAR	6	
PRODUCT_CODE	CHAR	6	
QUANTITY	NUMBER	5	

PRIMARY KEY = ORDER_NO + ORDER_DATE + CUSTOMER_ID + PRODUCT_CODE

4. Insert 5-10 records in all the tables.
5. Apply UNIQUE constraint on CUSTOMER_ID+PRODUCT_CODE on the ORDER table.
6. Define a foreign key on CUSTOMER_ID of ORDER table referring to CUSTOMER_ID of CUSTOMER table.
7. Define a foreign key on PRODUCT_CODE of ORDER table referring to PRODUCT_CODE of PRODUCT table.
8. Count the customerwise number of orders.
9. Calculate the average selling price of all products.
10. List the customer names for which we have orders in hand.
11. List the yearwise number of orders placed.
12. Display the customers who have placed some order. Use IN and EXISTS operator.
13. Find the customer who has placed maximum number of orders.
14. Create unique index on ORDER_NO, CUSTOMER_ID and PRODUCT_CD of ORDER table.
15. Create a non unique index on STATE of CUSTOMER table.
16. Create a view named “vw_balance” which will display customer names with balances more than 4000.
17. Add two new customers through the view.
18. Create a view named “vw_city” which will contain citywise total balances.

3.9 BIBLIOGRAPY, REFERENCES AND FURTHER READING

- Database Management Systems, Third Edition by RamaKrishnan, Gehre. McGraw Hill
- Database System Concepts, Fifth Edition by Silberschatz, Korth, Sudarshan. McGraw Hill
- Murach's Oracle SQL and PL/SQL by Joel Murach. Shroff Publishers & Distributors
- Oracle Database 11g by Satish Asnani. PHI Learning Private Limited
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, Tata McGraw-Hill

- SQL, PL/SQL The programming language of Oracle, Bayross Ivan, BPB Publications
- Fundamentals of Database Systems, Elmasri Ramez and Navathe B. Shamkant, Pearson

3.10 ONLINE REFERENCES

Wikipedia Link

http://en.wikipedia.org/wiki/SQL

Oracle Database PL/SQL language Reference 11g Release 2 (11.2), part number E25519-05

http://docs.oracle.com/cd/E11882_01/appdev.920/a96590/adg09dyn.htm

Unit - II

4

SECURITY PRIVILEGES, SET OPERATORS & DATETIME FUNCTIONS

Unit Structure

4.0 OBJECTIVES

At the end of this chapter you will be able to:

- Get summary information using CUBE, ROLLUP and GROUPING
- Combine queries using SET operators
- Use DateTime Functions
- Control User Access

4.1 INTRODUCTION

In the previous unit, we saw basic SQL operators, statements, queries and subqueries. In this unit, we shall see advanced subqueries, various advanced operators and security privileges.

In the previous chapter we discussed SQL keywords and functions. In this chapter we will see the extensions of GROUP BY clause: ROLLUP and CUBE operators, and GROUPING function. Also we will combine queries using SET operators and see various DateTime functions.

4.2 ENHANCEMENTS TO GROUP BY FUNCTION

In the previous chapter we saw that the **GROUP BY** clause forms groups on specified columns. The **HAVING** clause filters these groups depending on some conditions.

Assume the records in the EMP table as shown below.

EMPNO	ENAME	HIREDATE	DEPTNO	JOB	SALARY	COMM
111	Satish	19-DEC-2008	10	CLERK	7000	1000
222	Rashmi	01-JAN-1987	20	ANALYST	8000	550
333	Rishi	05-JUN-1976	10	MANAGER	10,000	450
444	Anil	16-APR-1967	10	PRESIDENT	15,000	2000
555	Anita	-	30	MANAGER	8000	1000
666	Nilesh	20-MAY-1987	20	MANAGER	13,000	-
777	Ruchi	11-JUN-2000	30	SALESMAN	5000	-
888	Sarika	-	20	SALESMAN	4000	-

A simple GROUP BY clause will show the following result.

- Display the amount of salary being paid jobwise for each department.

SELECT deptno, job, SUM(sal) FROM emp

GROUP BY deptno, job

ORDER BY deptno, job;

DEPTNO	JOB	SUM(SAL)
10	CLERK	7000
10	MANAGER	10000
10	PRESIDENT	15000
20	ANALYST	8000
20	MANAGER	13000
20	SALESMAN	4000
30	MANAGER	8000
30	SALESMAN	5000

4.2.1 ROLLUP Operator

The **ROLLUP** operator can be used to add one or more summary rows to a result set that uses grouping and aggregates. A summary is provided for each aggregate column included in the select list. All other columns, except the ones that identify which group is being summarized, are assigned null values. It also adds a summary row to the end of the result set that summarizes the entire result set.

- Display the amount of salary being paid jobwise for each department.

SELECT deptno, job, SUM(sal) FROM emp

GROUP BY ROLLUP (deptno, job)

ORDER BY deptno, job;

DEPTNO	JOB	SUM(SAL)
10	CLERK	7000
10	MANAGER	10000
10	PRESIDENT	15000
10	(null)	32000
20	ANALYST	8000
20	MANAGER	13000
20	SALESMAN	4000
20	(null)	25000
30	MANAGER	8000
30	SALESMAN	5000
30	(null)	13000
(null)	(null)	70000

4.2.2 CUBE Operator

The **CUBE** operator is similar to the ROLLUP operator, except it adds summary rows for every combination of groups specified in the GROUP BY clause. It also adds a summary row to the end result set that summarizes the entire result set.

(14) Display the amount of salary being paid jobwise for each department.

```
SELECT deptno, job, SUM(sal) FROM emp

GROUP BY CUBE (deptno, job)

ORDER BY deptno, job;
```

DEPTNO	JOB	SUM(SAL)
10	CLERK	7000
10	MANAGER	10000
10	PRESIDENT	15000
10	(null)	32000
20	ANALYST	8000
20	MANAGER	13000

20	SALESMAN	4000
20	(null)	25000
30	MANAGER	8000
30	SALESMAN	5000
30	(null)	13000
(null)	ANALYST	8000
(null)	CLERK	7000
(null)	MANAGER	31000
(null)	PRESIDENT	15000
(null)	SALESMAN	9000
(null)	(null)	70000

4.2.3 GROUPING Function

Using the ROLLUP and CUBE operator introduces a null value to the column in a summary row that hasn't been summarized. If you want to assign a value other than null to these columns, you can do it by using the **GROUPING** function. The GROUPING function determines when a null value is assigned to a column as a result of the ROLLUP or CUBE operator. The column named in this function must be one of the columns named in the GROUP BY clause.

If a null value is assigned to the specified column as a result of the ROLLUP or CUBE operator, the GROUPING function returns a value of 1. Otherwise it returns a value of 0.

- Display the amount of salary being paid jobwise for each department.

```
SELECT
        CASE
        WHEN GROUPING (deptno) = 1 THEN '=========='
                ELSE deptno
        END AS dept_no,
        CASE
                WHEN GROUPING (job) = 1 THEN '=========='
                ELSE job
```

```
    END AS job,
    SUM(sal)
FROM emp
GROUP BY ROLLUP (deptno, job)
ORDER BY deptno, job;
```

DEPTNO	JOB	SUM(SAL)
10	CLERK	7000
10	MANAGER	10000
10	PRESIDENT	15000
10	==========	32000
20	ANALYST	8000
20	MANAGER	13000
20	SALESMAN	4000
20	==========	25000
30	MANAGER	8000
30	SALESMAN	5000
30	==========	13000
==========	==========	70000

4.3 SET OPERATORS

Set operators combine the results of two component queries into a single result. Queries containing set operators are called **compound queries**. Like joins, set operators combine data from two or more tables but there is a big difference. Joins try to combine columns from the base tables, however, set operators combine rows from two or more result sets.

Generally **SET operations** are applied on multiple SELECT statements. The records returned by each SELECT statement are treated as a SET of values and the final result is obtained depending on the SET operator used. There are four SET operators available:

- UNION
- UNION ALL
- INTERSECT
- MINUS

Conditions for SET operations

Two SELECT statements can be combined into a compund query by a SET operation if they satisfy the following conditions:

- The result set of both the queries must have the same number of columns.
- The data type of each column in the first result set must match with the data types of the columns of the second result set.

Restrictions on SET operations

- The column names for the resultant data set will come from the first query.
- If you want to use the ORDER BY clause in the query involving SET operations, you must place the ORDER BY clause only once at the end of the compound query. The component queries can't have individual ORDER BY clauses.

Syntax:

<query 1>

[SET OPERATOR]

<query 2>

Assume two tables “student” and “student_details” as shown below.

CLASS	ROLLNO	NAME
201	1	Satish
201	2	Rashmi
205	3	Rishi
205	4	anil

ROLLNO	SUBJECT	MARKS
1	Maths	12
1	Physics	23
2	Maths	34
3	Maths	35

4.3.1 INTERSECT Operator

Returns only those rows which are common to both queries.

- Display the rollnos of students that are in both the student and student_details table.

```
SELECT rollno FROM student

INTERSECT

SELECT rollno FROM student_details;
```

ROLLNO
1
2
3

- Display the details of students from the student table for which marks have been entered in the student_details table.

```
SELECT * FROM student

WHERE rollno IN

(

        SELECT rollno FROM student

        INTERSECT

        SELECT rollno FROM student_details

);
```

CLASS	ROLLNO	NAME
201	1	Satish
201	2	Rashmi
205	3	Rishi

4.3.2 UNION Operator

Returns the values which exist in either of the two queries. By default, a UNION eliminates duplicate rows.

10. Display all the rollnos which exist either in the student or in the student_details table.

```
SELECT rollno FROM student

UNION

SELECT rollno FROM student_details;
```

ROLLNO
1
2
3
4

4.3.3 UNION ALL Operator

Returns the values which exist in either of the two queries. The **UNION** operator has an additional clause ALL which displays the duplicate values also.

Display all the rollnos which exist either in the student or in the student_details table.

```
SELECT rollno FROM student

UNION ALL

SELECT rollno FROM student_details;
```

ROLLNO
1
2
3
4
1
1
2
3

Using ORDER BY clause

In order to arrange the final result of any SET operator, we can use the ORDER BY clause at the end of the last SELECT statement.

- Display all the rollnos which exist either in the student or in the student_details table.

```
SELECT rollno FROM student

UNION ALL

SELECT rollno FROM student_details

ORDER BY rollno;
```

ROLLNO
1
1
1
2
2
3
3
4

4.3.4 MINUS Operator

The MINUS operator takes the result set of one SELECT statement, and removes those rows that are also returned by a second SELECT statement. It removes the records that are common to both the queries and displays the remaining records from the query 1.

19. Display the rollnos whose marks have not been entered in the student_details table.

```
SELECT rollno FROM student

MINUS

SELECT rollno FROM student_details;
```

ROLLNO
4

4.4 DATETIME FUNCTIONS

We saw a brief overview of DateTime functions in chapter 2. In this section we discuss about these functions in greater detail.

Datetime functions operate on date (DATE), timestamp (TIMESTAMP, TIMESTAMP WITH TIME ZONE, and TIMESTAMP WITH LOCAL TIME ZONE), and interval (INTERVAL DAY TO SECOND, INTERVAL YEAR TO MONTH) values.

Two operators, plus and minus signs, can be used to work with dates. Summary of these operators is provided in the table below.

Operator	Description
+	Adds the specified number of days to a date
-	Subtracts the specified number of days from a date. Or, subtracts one date from another and returns the number of days between the two dates.

The table given below gives the syntax of commonly used date time functions and also their explanation. If you study the summaries and examples of these functions, you shouldn't have much trouble using them.

Function	Description
SYSDATE	Returns the current local date and time based on the operating system's clock
CURRENT_DATE	Returns the local date and time adjusted for the current session time zone

ROUND (date [, date_format])	Returns the date rounded to the unit specified by the date format. If the format is omitted, rounds to the nearest day.
TRUNC (date [, date_format])	Works like the ROUND function but truncates the date.
MONTHS_BETWEEN (date1, date2)	Returns the number of months between date1 and date2.
ADD_MONTHS (date, integer_months)	Adds the specified number of months to the specified date and returns the resulting date.
LAST_DAY (date)	Returns the date for the last day of the month for the specified date.
NEXT_DAY (date, day_of_week)	Returns the date for the next day of the week that comes after the specified date.

Examples that use date/time functions

Example	**Result**
SYSDATE	19-OCT-09 04:23:36 PM
ROUND (SYSDATE)	20-OCT-09 12:00:00 AM
ROUND (SYSDATE, 'MI')	19-OCT-09 04:24:00 PM
TRUNC (SYSDATE, 'MI')	19-OCT-09 04:23:00 PM
MONTHS_BETWEEN ('15-SEP-08', '01-AUG-08')	1.45161290.....
ADD_MONTHS ('19-OCT-09', -1)	19-SEP-09

LAST_DAY ('15-FEB-09')	28-FEB-09
NEXT_DAY ('15-AUG-08', 'THURS')	21-AUG-08
SYSDATE – 1	18-OCT-09

For Example, let us find out the number of years for various employees who served the company. (refer to the emp table)

SELECT empno, ename, hiredate, ROUND((sysdate – hiredate)/365) no_of_years

FROM emp;

EMPNO	**ENAME**	**HIREDATE**	**NO_OF_YEARS**
111	Satish	19-DEC-2008	01
222	Rashmi	01-JAN-1987	22
333	Rishi	05-JUN-1976	33
444	Anil	16-APR-1967	42
666	Nilesh	20-MAY-1987	22
777	Ruchi	11-JUN-2000	9

Note: The employees whose hire date is NULL are not listed. Since the ROUND function is used without any specified format, rounding is done for the next year (i.e. for the first row the time between hiredate (19/12/08) and sysdate (19/10/2009) is 10 months).

4.4.1 Parsing Date and Time

TO_CHAR function can be used to return various parts of a DATE value as a string. To do that you need to specify the appropriate date format element for the part of the DATE value that you want to return.

Example	Result
TO_CHAR (SYSDATE, 'DD-MON-RR HH:MI:SS')	19-OCT-09 04:23:36 PM
TO_CHAR (SYSDATE, 'YEAR')	TWO THOUSAND NINE
TO_CHAR (SYSDATE, 'YYYY')	2009
TO_CHAR (SYSDATE, 'YY')	09
TO_CHAR (SYSDATE, 'MONTH')	OCTOBER
TO_CHAR (SYSDATE, 'MON')	OCT
TO_CHAR (SYSDATE, 'MM')	10
TO_CHAR (SYSDATE, 'DAY')	MONDAY
TO_CHAR (SYSDATE, 'DY')	MON
TO_CHAR (SYSDATE, 'DD')	19
TO_CHAR (SYSDATE, 'HH24')	16
TO_CHAR (SYSDATE, 'HH')	04
TO_CHAR (SYSDATE, 'MI')	23
TO_CHAR (SYSDATE, 'Q')	4

4.5 CONTROLLING USER ACCESS

Oracle is a multi-user RDBMS and provides a secure environment such that the objects owned by a user are by default not accessible to the other users. It provides the facility that the owner of an object can grant various permissions to other database users as per requirements.

Authorization includes primarily two processes:

- Permitting only certain users to access, process, or alter data.

- Applying varying limitations on user access or actions. The limitations placed on (or removed from) users can apply to objects such as schemas, tables, or rows or to resources such as time (CPU, connect, or idle times).

A user **privilege** is the right to run a particular type of SQL statement, or the right to access an object that belongs to another user, run a PL/SQL package, and so on.

Roles are created by users (usually administrators) to group together privileges or other roles. They are a way to facilitate the granting of multiple privileges or roles to users.

There are two types of privileges: system privileges and object privileges.

4.5.1 System privileges:

A **system privilege** is the right to perform a particular action or to perform an action on any schema objects of a particular type. For example, the privileges to create tablespaces and to delete the rows of any table in a database are system privileges.

Some of the most common system privileges are:

- CREATE / ALTER / DROP USER
- CREATE SESSION
- CREATE / ALTER / DROP TABLE
- CREATE VIEWS
- CREATE PROCEDURE
- CREATE SEQUENCE
- CREATE PUBLIC SYNONYM

4.5.2 Object Privileges:

An **object privilege** is a right that you grant to a user on a database objects like tables, views, sequences, packages, procedures. Some examples of object privileges include the right to:

- Use an edition
- Update a table
- Select rows from another user's table
- Execute a stored procedure of another user

4.5.3 What a user can grant?

A user can grant privileges on any object he/she owns. Different objects have different permissions to be assigned to other users as specified in the table below.

Object	Privileges
Table	SELECT, INSERT, UPDATE, DELETE, ALTER, INDEX
View, Materialized Views	SELECT, INSERT, UPDATE, DELETE
Sequence	SELECT, ALTER
Functions, Procedures, Packages	EXECUTE
Index	EXECUTE

4.5.4 GRANT/REVOKE PRIVILEGES

The Data Control Language commands are used to enforce database security in a multiple user database environment. The Data Control Language (DCL) authorizes users and groups of users to access and manipulate data. Its two main statements are:

13. GRANT: authorizes one or more users to perform an operation or a set of operations on an object.
14. REVOKE: eliminates a grant, which may be the default grant.

4.5.4.1 GRANT COMMAND

The **GRANT** command is used to grant system and object privileges to a role or a user.

Granting System Privileges

You can grant system privileges to users and roles. If you grant system privileges to roles, then you can use the roles to exercise system privileges.

The **syntax** of GRANT statement for system privileges

GRANT <system_privilege>

TO <user_or_role>

[WITH ADMIN OPTION]

The **WITH ADMIN OPTION** clause allows the user or role to grant the specified system privileges to other users or roles.

For example,

- A statement that grants a system privilege to a role

GRANT CREATE SESSION TO ap_user;

- A statement that grants a system privilege to a role with the admin option

GRANT CREATE SESSION TO ap_developer WITH ADMIN OPTION;

Granting Object Privileges

Each type of object has different privileges associated with it. Object privileges can be granted to users and roles. If you grant object privileges to roles, then you can make the privileges selectively available.

The **syntax** of GRANT statement for object privileges

GRANT <object_privilege>

ON <[schema_name.]object_name [(column [,.....])]>

TO <user_or_role>

[WITH GRANT OPTION]

The **WITH GRANT OPTION** clause allows the user or role to grant the specified object privileges to other users or roles.

Consider the following examples,

- A statement that grants an object privilege to a role

GRANT SELECT ON emp TO ap_user;

- A statement that grants all object privileges to a role with the grant option

GRANT SELECT, INSERT, UPDATE, DELETE ON emp TO ap_developer WITH GRANT OPTION;

4.5.4.2 REVOKE COMMAND

The **REVOKE** command is used to revoke system or object privileges from a role or user.

Revoking System Privileges

You can revoke system privileges from roles or users.

The **syntax** of REVOKE statement for system privileges

REVOKE <system_privilege>

FROM <user_or_role>

Consider the following examples,

- A statement that revokes a system privilege from a role

REVOKE DROP ANY VIEW FROM ap_user;

Revoking Object Privileges

Each type of object has different privileges associated with it. Object privileges can be revoked from users and roles.

The **syntax** of REVOKE statement for object privileges

REVOKE [GRANT OPTION FOR] <object_privilege>

ON <[schema_name.]object_name [(column [,.....])]>

FROM <user_or_role>

[RESTRICT | CASCADE]

The **GRANT OPTION FOR** clause allows the user or role to revoke the specified object privileges from other users or roles.

The **RESTRICT** clause revokes all privileges for the user.

The **CASCADE** clause revokes all privileges for the user and given by the user to other users.

Consider the following examples,

- A statement that revokes an object privilege from a role

REVOKE SELECT ON emp FROM ap_user;

- A statement that revokes selected object privileges from a role with the grant option

REVOKE GRANT OPTION FOR SELECT, INSERT, ON emp FROM ap_developer RESTRICT;

NOTE:

You can specify **ALL [PRIVILEGES]** to grant or revoke all available object privileges for an object. **ALL** is not a privilege; rather, it is a shortcut, or a way of granting or revoking all object privileges with one **GRANT** and **REVOKE** statement. If all object privileges are granted using the ALL shortcut, then individual privileges can still be revoked.

Similarly, you can revoke all individually granted privileges by specifying ALL.

Consider the following examples,

- A statement that grants all object privileges to a role.

GRANT ALL ON emp TO ap_developer;

- A statement that revokes all object privilege from a role

REVOKE ALL ON emp FROM ap_user;

4.6 SUMMARY

- The ROLLUP operator can be used to add one or more summary rows to a result set that uses grouping and aggregates.
- The CUBE operator is similar to the ROLLUP operator, except it adds summary rows for every combination of groups specified in the GROUP BY clause.
- The GROUPING function determines when a null value is assigned to a column as a result of the ROLLUP or CUBE operator.
- Set operators combine the results of two component queries into a single result. Queries containing set operators are called compound queries.
- The summary of various SET operators is as follows:

Operator	Description
UNION ALL	Combines the results of two SELECT statements into one result set.
UNION	Combines the results of two SELECT statements into one result set, and then eliminates any duplicate rows from that result set.
MINUS	Takes the result set of one SELECT statement, and removes those rows that are also returned by the another SELECT statement.
INTERSECT	Returns only those rows that are returned by each of the two SELECT statements.

- Datetime functions operate on date, timestamp, and interval values.
- TO_CHAR function can be used to return various parts of a DATE value as a string.
- A user privilege is the right to run a particular type of SQL statement, or the right to access an object that belongs to another user, run a PL/SQL package, and so on.
- There are two types of privileges: system privileges and object privileges.
- A system privilege is the right to perform a particular action or to perform an action on any schema objects of a particular type.
- An object privilege is a right that you grant to a user on a database objects like tables, views, sequences, packages, procedures.
- The GRANT command is used to grant system and object privileges to a role or a user.
- The REVOKE command is used to revoke system or object privileges from a role or user.

4.7 REVIEW QUESTIONS

1. Explain the ROLLUP and CUBE operator with examples.
2. What are SET operators? List and explain the different types of SET operators.
3. Using DateTime functions, how to calculate age from date of birth?
4. What is a privilege? Explain the different types of privileges.
5. Explain the GRANT command in detail with examples.
6. Explain the REVOKE command in detail with examples.

4.8 LAB ASSIGNMENT

1. Create a CUSTOMER table with the following columns and constraints

Column name	Data type	Size	Constraint
CUSTOMER_ID	CHAR	6	PRIMARY KEY. MUST BEGIN WITH 'C'
CUSTOMER_NAME	VARCHAR2	20	NOT NULL
ADDRESS	VARCHAR2	20	UNIQUE
CITY	VARCHAR2	20	

PINCODE	NUMBER	6	
STATE	VARCHAR2	20	
BALANCE_DUE	NUMBER	8,2	

2. Create a PRODUCT table with the following columns and constraints

Column name	Data type	Size	Constraint
PRODUCT_CODE	CHAR	6	PRIMARY KEY
PRODUCT_NAME	VARCHAR2	20	UNIQUE
QTY_AVAIL	NUMBER	5	
COST_PRICE	NUMBER	8,2	
SELLING_PRICE	NUMBER	8,2	

3. Create a ORDER table with the following columns and constraints

Column name	Data type	Size	Constraint
ORDER_NO	CHAR	6	
ORDER_DATE	TIMESTAMP		
CUSTOMER_ID	CHAR	6	
PRODUCT_CODE	CHAR	6	
QUANTITY	NUMBER	5	

PRIMARY KEY = ORDER_NO + ORDER_DATE + CUSTOMER_ID + PRODUCT_CODE

4. Insert 5-10 records in all the tables.
5. Apply UNIQUE constraint on CUSTOMER_ID+PRODUCT_CODE on the ORDER table.
6. Define a foreign key on CUSTOMER_ID of ORDER table referring to CUSTOMER_ID of CUSTOMER table.
7. Define a foreign key on PRODUCT_CODE of ORDER table referring to PRODUCT_CODE of PRODUCT table.
8. Find the total number of orders and the customerwise number of orders from the ORDER table.
9. Display the orders placed during the year '2008'.
10. What would be the date 10 days from today.
11. 28/02/2010 would fall on which day (Mon, Tue, ...)?
12. List the customer codes that have placed orders using SET operators.
13. Display the customer codes that have not placed any order using SET operators.
14. Display the customer codes that have placed some order using SET operators.

15. Permit the user “Ram” to be able to INSERT and DELETE commands in the CUSTOMER table.
16. Grant INSERT permission to the user “Ravi” such that he can further grant the INSERT permission to other users for the CUSTOMER table.
17. Withdraw the DELETE permission from “Ram”.
18. Give the UPDATE permission on ADDRESS and STATE column of the CUSTOMER table to “Kishan”.

4.9 BIBLIOGRAPHY, REFERENCES AND FURTHER READING

- Database Management Systems, Third Edition by RamaKrishnan, Gehre. McGraw Hill
- Database System Concepts, Fifth Edition by Silberschatz, Korth, Sudarshan. McGraw Hill
- Murach's Oracle SQL and PL/SQL by Joel Murach. Shroff Publishers & Distributors
- Oracle Database 11g by Satish Asnani. PHI Learning Private Limited
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, Tata McGraw-Hill
- SQL, PL/SQL The programming language of Oracle, Bayross Ivan, BPB Publications
- Fundamentals of Database Systems, Elmasri Ramez and Navathe B. Shamkant, Pearson

4.10 ONLINE REFERENCES

Wikipedia Link

http://en.wikipedia.org/wiki/SQL

Oracle Database PL/SQL language Reference 11g Release 2 (11.2), part number E25519-05

http://docs.oracle.com/cd/E11882_01/appdev.920/a96590/adg09dyn.htm

❖❖❖❖

5

ADVANCED SUBQUERIES

Unit Structure

5.0 OBJECTIVES

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

15. Understand Scalar and Correlated Subqueries
16. Write and Execute Multiple Column Subqueries
17. Understand WITH clause
18. Understand Hierarchical queries

5.1 INTRODUCTION

In the previous chapters we learnt that, queries can be nested so that the results of one query can be used in another query via a relational operator or aggregation function. This type of nested query is known as a subquery. In this chapter we will discuss more about the specifics of using subqueries.

To illustrate the concepts in this chapter, assume the tables "**EMP**" and "**EMP_DETAILS**" having the following records.

Table EMP:

EMPID	DESIGNATION	DEPT	SALARY
1001	Manager	Finance	50000
1002	Executive	Finance	25000
1003	Senior Executive	Finance	35000
1004	Manager	HR	20000
1005	Executive	HR	20000
1006	Senior Executive	HR	30000
1007	Manager	Admin	55000
1008	Executive	Finance	25000
1009	Executive	Finance	25000
1010	Executive	HR	25000

Table EMP_DETAILS:

EMPID	ENAME	AGE
1001	Rajesh	23
1002	Tejal	25

1003	Himesh	35
1004	Himali	37
1005	Rehan	40
1006	Kiran	28
1007	Ashima	21
1008	Vikram	31
1009	Ridhi	26

5.2 MULTIPLE COLUMN SUBQUERIES

Multiple column subquery returns the rows on the basis of matching of a pair of given columns for a given row. It first selects the row on the basis of the WHERE clause and then finds the other rows on the basis of a matching pair of columns of the particular row. There are two types of comparison in multiple column subqueries.

Pair wise: In pair wise comparison we search both the columns match in the same subquery, e.g. "**WHERE** (column1, column2) **IN** (subquery1)"

Non Pair wise: In non pair wise comparison we search the columns in separate subqueries, e.g. "**WHERE** (column1) **IN** (subquery1) **AND** (column2) **IN** (subquery2)".

- List the empid of employees who have the same salary and designation as the employee having empid 1009.
  SELECT DISTINCT empid FROM emp

  WHERE (designation, salary)

  IN (SELECT designation, salary FROM emp WHERE empid = 1009)

  ORDER BY empid;

  Or

  SELECT DISTINCT empid FROM emp

  WHERE ((designation) IN (SELECT designation, salary FROM emp WHERE empid = 1009)) AND ((salary) IN (SELECT designation, salary FROM emp WHERE empid = 1009))

  ORDER BY empid;

Both the queries above will give the same result set. The first query is the pair wise comparison while the second query is the non pair wise comparison. The output of the above query is:

EMPID
1002
1008
1009
1010

5.2.1 CODING SUBQUERIES IN THE FROM CLAUSE

A subquery can be coded in place of a table specification i.e. in the **FROM** clause. The results of the subquery are joined with another table. When you use a subquery in this way, it can return any number of rows and columns. This type of subquery, in the FROM clause of a SELECT statement, is referred to as an **inline view** since it works like a view that is temporarily created and stored in the memory. Inline views are most useful when you need to summarize the results of a summary query. When you create an inline view, you must assign an alias to it. Then, you can use the inline view within the outer query just as you would any other table.

For example,

- To find the department in which the maximum number of employees work.

  SELECT MAX (inview.total_emp)

  FROM (SELECT COUNT (empid) as total_emp FROM emp GROUP BY dept) inview;

MAX(INVIEW.TOTAL_EMP)
5

In the above query, we use the alias **'inview'** for the subquery that begins after the FROM clause. The subquery counts the number of employees that work in each department & the outer query takes the values from the

subquery using the MAX function finds the maximum number of employees to give the result set.

5.3 SCALAR SUBQUERIES

A **scalar subquery** expression is a subquery that returns exactly one column value from one row. The value of the scalar subquery expression is the value of the select list item of the subquery. If the subquery returns 0 rows, then the value of the scalar subquery expression is NULL. If the subquery returns more than one row, then Oracle returns an error.

You can use a scalar subquery expression in most syntax that calls for an expression (*expr*).

For example,

- Get the name and age of the employee enjoying maximum salary.

  SELECT ename, age FROM emp_details

  WHERE empid = (SELECT empid FROM emp WHERE salary =

  (SELECT MAX(salary) FROM emp));

ENAME	AGE
Ashima	21

In the above example, the innermost scalar subquery gets executed first providing the maximum salary from the EMP table, then the empid of the employee is selected and given to the outer query to get the name and age of the employee.

5.4 CORRELATED SUBQUERY

A **correlated subquery** is a subquery that is executed once for each row processed by the outer query. It's similar to using a loop to do repetitive processing in a procedural programming. In contrast, a noncorrelated subquery is executed only once. In correlated subquery, the outer query executes first and the inner query will execute second. Each subquery is executed once for every row of the outer query.

For example,

- Get the list of employees whose salary are higher than or equal to the average salary of their respective departments.

```
SELECT e1.empid, e1.dept FROM emp e1
WHERE salary >= (SELECT AVG(salary) FROM emp e2 GROUP
BY e2.dept
    HAVING e1.empid = e2.empid);
```

EMPID	DEPT
1001	Finance
1003	Finance
1006	HR
1007	Admin

To run the inner query we need to know the “dept” of the employee selected in the outer query. For each and every department value coming from the outer query the average salary of the department is calculated in the inner query and compared with the salary of the employee in the outer query.

5.5 WITH CLAUSE (SUBQUERY FACTORING CLAUSE)

The WITH *query_name* clause lets you assign a name to a subquery block. You can then reference the subquery block multiple places in the query by specifying *query_name*. SQL optimizes the query by treating the query name as either an inline view or as a temporary table.

Syntax:

WITH query_name ([c_alias [, c_alias]...]) **AS** (subquery)

[, query_name ([c_alias [, c_alias]...]) **AS** (subquery)]...

The column aliases following the *query_name* and the set operators separating multiple subqueries in the AS clause are valid. You can specify this clause in any top-level SELECT statement and in most types of subqueries. The query name is visible to the main query and to all subsequent subqueries. To code multiple subquery factoring clauses, separate them with commas. Then each clause can refer to itself and any previously defined subquery factoring clauses in the same WITH clause.

5.5.1 FUNCTIONS OF THE WITH CLAUSE

(15) A named query can be referenced any number of times.

(16) Any number of named queries can be created.

(17) Named queries can reference other named queries that came before them and even correlate to previous named queries.

(18) The scope of the WITH clause is local to the SELECT in which they are defined.

Consider the following examples,

- Get the empid of the employee who works in the HR department and gets the maximum salary.

  WITH hr_sal AS

  (SELECT empid, salary FROM emp WHERE dept = 'HR')

  SELECT MAX (salary) FROM hr_sal;

MAX (SALARY)
30000

- Get the average age of employees working in the "Finance" department.

  WITH fin_age AS

  (SELECT empid, salary FROM emp WHERE dept = 'Finance'

  SELECT AVG (age) FROM emp_details

  WHERE empid IN (SELECT empid FROM fin_age);

5.6 HIERARCHICAL QUERIES

A **hierarchical query** loops through a result set and returns rows in a hierarchical sequence. If a table contains hierarchical data, then you can select rows in a hierarchical order using the hierarchical query clause.

Syntax:

SELECT select_list

FROM table_name

[**WHERE** search_condition]

START WITH row_specification

CONNECT BY PRIOR connect_expr

SELECT statements that contain hierarchical queries can contain the **LEVEL** pseudocolumn in the select list. LEVEL returns the value 1 for a root node, 2 for a child node of a root node, 3 for a grandchild,

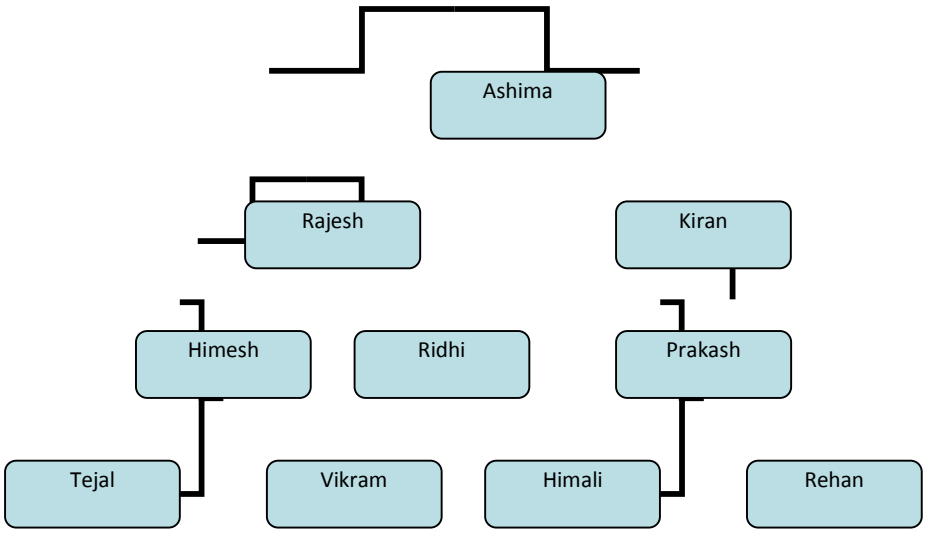
and so on. The number of levels returned by a hierarchical query may be limited by available user memory.

START WITH Clause – used to specify a condition that identifies the row(s) to be used as the root(s) of a hierarchical query.

CONNECT BY Clause – followed by the **PRIOR** keyword is used to specify a condition that identifies the relationship between parent rows and child rows of the hierarchy.

Assume the records in the EMP_NEW table

EMPID	ENAME	DESIGNATION	DEPT	SALARY	MGRID
1001	Rajesh	Manager	Finance	50000	1007
1002	Tejal	Executive	Finance	25000	1003
1003	Himesh	Senior Executive	Finance	35000	1001
1004	Himali	Manager	HR	20000	1010
1005	Rehan	Executive	HR	20000	1010
1006	Kiran	Senior Executive	HR	30000	1007
1007	Ashima	President	Admin	55000	-
1008	Vikram	Executive	Finance	25000	1003
1009	Ridhi	Executive	Finance	25000	1001
1010	Prakash	Executive	HR	25000	1006



A query that returns hierarchical data

SELECT LEVEL, empid, ename, mgrid FROM emp_new

START WITH ename = “Ashima”

CONNECT BY PRIOR empid = mgrid

ORDER BY LEVEL, empid;

LEVEL	EMPID	ENAME	MGRID
1	1007	Ashima	-
2	1001	Rajesh	1007
2	1006	Kiran	1007
3	1003	Himesh	1001
3	1009	Ridhi	1001
3	1010	Prakash	1006
4	1002	Tejal	1003
4	1004	Himali	1010
4	1005	Rehan	1010
4	1008	Vikram	1003

The EMP_NEW table uses the MGR column to identify the manager for each employee. Here, Ashima is the top level manager since she doesn’t have a manager. Rajesh and Kiran report to Ashima and so on.

The hierarchical query uses the LEVEL pseudo-column to return a column that identifies the level of the employee within the hierarchy. In addition, this query uses the LEVEL pseudo-column in the ORDER BY clause to sort by this column.

After the FROM clause, this query uses the START WITH clause to identify the row to be used as the root of the hierarchy. Finally the CONNECT BY clause specifies the condition that identifies the relationship between the parent rows and the child rows.

5.7 SUMMARY

- **Multiple column subquery** returns the rows on the basis of matching of a pair of given columns for a given row. It first selects the row on the basis of the WHERE clause and then finds the other rows on the basis of a matching pair of columns of the particular row.
- There are two types of comparison in multiple column subqueries - **Pair-Wise and Non Pair-wise**.
- A **subquery** can be coded in place of a table specification i.e. in the **FROM** clause. The results of the subquery are joined with another table.
- A **scalar subquery** expression is a subquery that returns exactly one column value from one row.
- A **correlated subquery** is a subquery that is executed once for each row processed by the outer query. It's similar to using a loop to do repetitive processing in a procedural programming.
- The **WITH query_name** clause lets you assign a name to a subquery block.
- To code multiple subquery factoring clauses, separate them with commas. Then each clause can refer to itself and any previously defined subquery factoring clauses in the same WITH clause.
- If a table contains hierarchical data, then you can select rows in a hierarchical order using the **hierarchical query** clause.

5.8 REVIEW QUESTIONS

- Explain multiple column subqueries with suitable examples.
- What is an inline view? When can you use it? Explain with example.
- What is a scalar subquery? Explain with suitable example.
- What is a correlated subquery? Explain with suitable example.

- Explain the WITH clause. Why should you use the WITH clause?
- What is a hierarchical query? Illustrate with the help of an example.

5.9 LAB ASSIGNMENT

1. Create a CUSTOMER table with the following columns and constraints

Column name	Data type	Size	Constraint
CUSTOMER_ID	CHAR	6	PRIMARY KEY. MUST BEGIN WITH 'C'
CUSTOMER_NAME	VARCHAR2	20	NOT NULL
ADDRESS	VARCHAR2	20	UNIQUE
CITY	VARCHAR2	20	
PINCODE	NUMBER	6	
STATE	VARCHAR2	20	
BALANCE_DUE	NUMBER	8,2	

2. Create a PRODUCT table with the following columns and constraints

Column name	Data type	Size	Constraint
PRODUCT_CODE	CHAR	6	PRIMARY KEY
PRODUCT_NAME	VARCHAR2	20	UNIQUE
QTY_AVAIL	NUMBER	5	
COST_PRICE	NUMBER	8,2	
SELLING_PRICE	NUMBER	8,2	

3. Create a ORDER table with the following columns and constraints

Column name	Data type	Size	Constraint
ORDER_NO	CHAR	6	
ORDER_DATE	TIMESTAMP		
CUSTOMER_ID	CHAR	6	
PRODUCT_CODE	CHAR	6	
QUANTITY	NUMBER	5	

PRIMARY KEY = ORDER_NO + ORDER_DATE + CUSTOMER_ID + PRODUCT_CODE

- Insert 5-10 records in all the tables.
- Apply UNIQUE constraint on CUSTOMER_ID+PRODUCT_CODE on the ORDER table.
- Define a foreign key on CUSTOMER_ID of ORDER table referring to CUSTOMER_ID of CUSTOMER table.
- Define a foreign key on PRODUCT_CODE of ORDER table referring to PRODUCT_CODE of PRODUCT table.
- List the customer names in the order of decreasing quantity ordered.
- Calculate the average selling price of all products.
- List the customer names for which we have orders in hand.
- List the details of all products whose price is less than average price of the products.
- List the customers who have ordered for the same products.
- Get the name, price and quantity in stock of the costliest product.

5.10 BIBLIOGRAPHY, REFERENCES AND FURTHER READING

- Database Management Systems, Third Edition by RamaKrishnan, Gehre. McGraw Hill
- Database System Concepts, Fifth Edition by Silberschatz, Korth, Sudarshan. McGraw Hill
- Murach's Oracle SQL and PL/SQL by Joel Murach. Shroff Publishers & Distributors

- Oracle Database 11g by Satish Asnani. PHI Learning Private Limited
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, Tata McGraw-Hill
- SQL, PL/SQL The programming language of Oracle, Bayross Ivan, BPB Publications
- Fundamentals of Database Systems, Elmasri Ramez and Navathe B. Shamkant, Pearson

5.11 ONLINE REFERENCES

Wikipedia Link

http://en.wikipedia.org/wiki/SQL

Oracle Database PL/SQL language Reference 11g Release (11.2), part number E25519-05 2

http://docs.oracle.com/cd/E11882_01/appdev.920/a96590/adg09dyn.htm

Unit - III

6

INTRODUCTION TO PL/SQL

Unit Structure

6.0 OBJECTIVES

At the end of this chapter you will be able to

11. Understand Basic Concepts of PL/SQL coding,
12. Understand variables, identifiers and its types,
13. Different data types,
14. Different types of PL/SQL blocks,
15. The %TYPE Attribute and Sequences in PL/SQL Expressions.

6.1 INTRODUCTION

In Oracle, there is a special language available for developers to code stored procedures that seamlessly integrate with database object access via the language of database objects, SQL. However, this language offers far more execution potential than simple updates, selects, inserts, and deletes. This language offers a procedural extension that allows for modularity, variable declaration, loops and other logic constructs, and advanced error handling. This language is known as PL/SQL. PL/SQL stands for Procedural Language extension of SQL.

6.2 PL/SQL OVERVIEW

PL/SQL was developed by **Oracle Corporation** in the early 90's to enhance the capabilities of SQL. PL/SQL is a combination of SQL along with the procedural features of programming languages. In other words, it is a database-oriented programming language that is a powerful extension of SQL with procedural capabilities. The key strength of PL/SQL is its tight integration with the Oracle database.

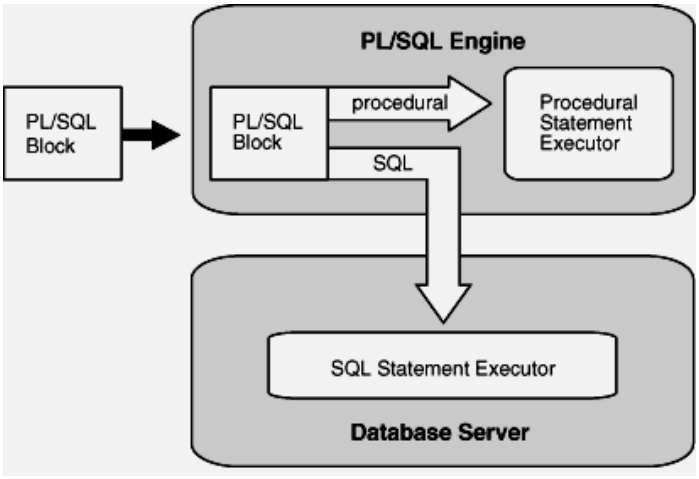
Being a procedural language, it has many standard capabilities including

Variable Definition and Assignment
Conditional Processing
Loop Constructs
Error Handling
Seamless Integration of SQL and SQL Functions

6.2.1 PL/SQL Engine

The **PL/SQL engine** is the tool used to define, compile, and run PL/SQL program units. The engine can be installed in the database or in an application development tool, such as Oracle Forms. In either environment, the PL/SQL engine accepts as input

any valid PL/SQL unit. The engine runs procedural statements, but sends SQL statements to the SQL engine in the database, as shown in the figure below.



Typically, the database processes PL/SQL units. When an application development tool processes PL/SQL units, it passes them to its local PL/SQL engine. If a PL/SQL unit contains no SQL statements, the local engine processes the entire PL/SQL unit. This is useful if the application development tool can benefit from conditional and iterative control.

For example, Oracle Forms applications frequently use SQL statements to test the values of field entries and do simple computations. By using PL/SQL instead of SQL, these applications can avoid calls to the database.

6.2.2 Advantages of PL/SQL

PL/SQL has these advantages:

- **Block Structure:** PL SQL consists of blocks of code, which can be nested within each other. Each block forms a unit of a task or a logical module. PL/SQL Blocks can be stored in the database and reused.

- **Tight Integration with SQL:** PL/SQL is tightly integrated with SQL, the most widely used database manipulation language. For example: PL/SQL lets you use all SQL data manipulation, cursor control, and transaction control statements, and all SQL functions, operators, and pseudo-columns. It also fully supports SQL data types.

- **Procedural Language Capability:** PL SQL consists of procedural language constructs such as conditional statements (if else statements) and loops like (FOR loops).

- **High Performance:** PL/SQL lets you send a block of statements to the database, significantly reducing traffic between the application and the database.

- **High Productivity:** PL/SQL lets you write compact code for manipulating data. Just as a scripting language like PERL can read, transform, and write data in files, PL/SQL can query, transform, and update data in a database.

- **Portability:** You can run PL/SQL applications on any operating system and platform where Oracle Database runs.

- **Scalability:** PL/SQL stored subprograms increase scalability by centralizing application processing on the database server. The shared memory facilities of the shared server let Oracle Database support thousands of concurrent users on a single node.

- **Manageability:** PL/SQL stored subprograms increase manageability because you can maintain only one copy of a subprogram, on the database server, rather than one copy on each client system. Any number of applications can use the subprograms, and you can change the subprograms without affecting the applications that invoke them.

- **Support for Object-Oriented Programming:** PL/SQL supports object-oriented programming with "Abstract Data Types".

- **Support for Developing Web Applications:** PL/SQL lets you create applications that generate web pages directly from the database, allowing you to make your database available on the Web and make back-office data accessible on the intranet.

- **Support for Developing Server Pages:** PL/SQL Server Pages (PSPs) let you develop web pages with dynamic content. PSPs are an alternative to coding a stored subprogram that writes the HTML code for a web page one line at a time.
- **Error Handling:** PL/SQL handles errors or exceptions effectively during the execution of a PL/SQL program. Once an exception is

caught, specific actions can be taken depending upon the type of the exception or it can be displayed to the user with a message.

6.3 PL/SQL BLOCKS

PL/SQL provides a server-side, stored procedural language that is easy-to-use, seamless with SQL, robust, portable, and secure. You can access and manipulate database data using procedural schema objects called PL/SQL blocks. The basic unit in PL/SQL is a block, which groups related declarations and statements. All PL/SQL programs are made up of blocks, which can be nested within each other.

PL/SQL blocks generally are categorized as follows:

- A **subprogram** is a PL/SQL block that is stored in the database and can be called by name from an application. When you create a subprogram, the database parses the subprogram and stores its parsed representation in the database. You can declare a subprogram as a procedure or a function.
- An **anonymous block** is a PL/SQL block that appears in your application and is not named or stored in the database. In many applications, PL/SQL blocks can appear wherever SQL statements can appear.

6.3.1 *PL/SQL Block Structure*

```
HEADER    -- Header part (optional)

<< label >> (optional)

DECLARE    -- Declarative part (optional)

  -- Declarations of local types, variables, & subprograms

BEGIN      -- Executable part (required)

  -- Statements (which can use items declared in declarative part)

[EXCEPTION -- Exception-handling part (optional)

  -- Exception handlers for exceptions (errors) raised in executable part]

END;
```

A PL/SQL Block consists of the following sections:

(19) The Header section (optional).

(20) The Declaration section (optional).

(21) The Execution section (mandatory).

(22) The Exception (Error Handling) section (optional).

- **Header Section:**

The Header section of a PL/SQL Block is relevant only for subprograms. The Header determines the way that the subprogram must be called. The header includes the name, parameter list and RETURN clause (for a function only).

- **Declaration Section:**

The Declaration section of a PL/SQL Block starts with the reserved keyword DECLARE. This section is optional and is used to declare any placeholders like variables, constants, records and cursors, which are used to manipulate data in the execution section. Placeholders may be any of Variables, Constants and Records, which stores data temporarily. Cursors are also declared in this section.

- **Execution Section:**

The Execution section of a PL/SQL Block starts with the reserved keyword BEGIN and ends with END. This is a mandatory section and is the section where the program logic is written to perform any task. The programmatic constructs like loops, conditional statement and SQL statements form a part of execution section.

- **Exception Section:**

The Exception section of a PL/SQL Block starts with the reserved keyword EXCEPTION. This section is optional. Any errors in the program can be handled in this section, so that the PL/SQL Blocks terminates gracefully. If the PL/SQL Block contains exceptions that cannot be handled, the Block terminates abruptly with errors.

Every statement in the above three sections must end with a semicolon (**;**). PL/SQL blocks can be nested within other PL/SQL blocks. Comments can be used to document code.

6.3.2 PL/SQL Subprograms

A PL/SQL **subprogram** is a named PL/SQL block that permits the caller to supply parameters that can be input only, output only, or input and output values. A subprogram solves a specific problem or performs related tasks and serves as a building block for modular, maintainable database applications.

A subprogram is either a procedure or a function. Procedures and functions are identical except that functions always return a single value to the caller, whereas procedures do not. Procedures and functions are explained in detail in UNIT V.

- **6.3.3 PL/SQL Anonymous Blocks**

An anonymous block is an unnamed, non-persistent PL/SQL unit. It is a block of PL/SQL that's coded within a script. Typical uses for anonymous blocks include:

19. Initiating calls to subprograms and package constructs
20. Isolating exception handling
21. Managing control by nesting code within other PL/SQL blocks

Anonymous blocks do not have the code reuse advantages of stored subprograms.

The syntax for an anonymous PL/SQL block:

DECLARE
    Variable declaration
BEGIN
    Program Execution
EXCEPTION
    Exception handling
END;

An anonymous block begins with the DECLARE optional section, the header section is altogether missing from an anonymous block.

For Example,

To print a message "Hello, Welcome to the world of PL SQL"

```
BEGIN

DBMS_OUTPUT.PUT_LINE ('Hello, Welcome to the world of PL SQL');

END;

/
```

To execute a PL/SQL block you must code a front slash (/) after the END keyword.

DBMS_OUTPUT.PUT_LINE is a function that is used to generate output on the screen. There are built-in packages that offer a number of ways to generate output from within the PL/SQL program.

6.4 PL/SQL IDENTIFIERS

Identifiers are used to name PL/SQL program items & units which include

- Constants
- Variables
- Exceptions

- Cursors
- Subprograms
- Packages
- Keywords
- Labels
- Reserved words

Rules for naming identifiers

20. It must start with a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs.
21. Other characters such as hyphens, slashes, and spaces are not allowed.
22. Any reserved keyword in PL/SQL cannot be used as an identifier
23. Identifiers in PL/SQL are not case – sensitive. For e.g. the identifiers lastname, LastName and LASTNAME are the same.

Identifiers can be divided into the following **types**:

- Reserved words and keywords
- Predefined identifiers
- User-defined identifiers

6.4.1 Reserved words and Keywords

Reserved words and keywords are identifiers that have special syntactic meaning in PL/SQL. You cannot use them as ordinary user-defined identifiers. Some examples of reserved words and keywords are ALL, ALTER, BEGIN, ADD, CALL, GENERAL, HEAP, HASH etc.

6.4.2 Predefined Identifiers

Predefined identifiers are declared in the predefined package STANDARD. An example of a predefined identifier is the exception INVALID_NUMBER.

For a list of predefined identifiers, use this query:

SELECT TYPE_NAME FROM ALL_TYPES WHERE PREDEFINED='YES';

You can use predefined identifiers as user-defined identifiers, but it is not recommended. Your local declaration overrides the global declaration

6.4.3 User-defined Identifiers

A **user-defined identifier** is:

- Composed of characters from the database character set
- Either ordinary or quoted

6.4.3.1 Ordinary User-Defined Identifiers

An ordinary user-defined identifier:

- Begins with a letter
- Can include letters, digits, and these symbols:
  - Dollar sign ($)
  - Number sign (#)
  - Underscore (_)
- Is not a reserved word.

Examples of acceptable ordinary user-defined identifiers:

X
t2
phone#
credit_limit
LastName
oracle$number
money$$$tree
SN##
try_again_

6.4.3.2 Quoted User-Defined Identifiers

A quoted user-defined identifier is enclosed in double quotation marks. Between the double quotation marks, any characters from the database character set are allowed except double quotation marks, new line characters, and null characters. For example, these identifiers are acceptable:

"X+Y"
"last name"
"on/off switch"
"employee(s)"
"*** header info ***"

• 6.5 PL/SQL Placeholders

Placeholders are temporary storage area. Placeholders can be any of Variables, Constants and Records. Oracle defines **placeholders** to store data temporarily, which are used to manipulate data during the execution of a PL SQL block. Depending on the kind of data you want to store, you can define placeholders with a name and a datatype. Few of the datatypes used to define placeholders are as given below. Number (n,m) , Char (n) , Varchar2 (n) , Date , Long etc.

6.5.1 PL/SQL Variables

These are placeholders that store the values that can change through the PL/SQL Block. In order to use a variable, you need to declare it in the declaration section of the PL/SQL block.

The General Syntax to declare a variable is:

variable_name datatype [NOT NULL := value];

- variable_name is the name of the variable.
- datatype is a valid PL/SQL datatype.
- NOT NULL is an optional specification on the variable that is used when you want to compulsorily initialize a variable with a value before using it.
- value is also an optional specification, where you can initialize a variable.
- Each variable declaration is a separate statement and must be terminated by a semicolon.

For example, if you want to store the current salary of an employee, you can use a variable.

DECLARE

salary number (6);

“salary” is a variable of datatype number and of length 6.

The value of a variable can change in the execution or exception section of the PL/SQL Block. We can assign values to variables in the two ways given below.

- We can directly assign values to variables. The General Syntax is:

variable_name:= value;

- We can assign values to variables directly from the database columns by using a SELECT.. INTO statement. The General Syntax is:

SELECT column_name

INTO variable_name

FROM table_name

[WHERE condition];

Example: The below program will get the salary of an employee with id '1116' and display it on the screen.

```
DECLARE
        var_salary number(6);
        var_emp_id number(6) = 1116;
BEGIN
        SELECT salary
        INTO var_salary
        FROM employee
        WHERE emp_id = var_emp_id;
        dbms_output.put_line(var_salary);
        dbms_output.put_line('The employee ' || var_emp_id || ' has
salary  ' || var_salary);
END;
/
```

Scope of Variables

PL/SQL allows the nesting of Blocks within Blocks i.e, the Execution section (BEGIN block) of an outer block can contain inner blocks. Therefore, a variable which is accessible to an outer Block is also accessible to all nested inner Blocks. The variables declared in the inner blocks are not accessible to outer blocks. Based on their declaration we can classify variables into two types.

- **Local variables** - These are declared in a inner block and cannot be referenced by outside Blocks.
- **Global variables** - These are declared in a outer block and can be referenced by its itself and by its inner blocks.

6.5.2 PL/SQL Constants

A constant is the name of a memory location which stores a value used in a PL/SQL block that remains unchanged throughout the program. Similar to a variable a constant also needs to be declared in the declaration section.

The General Syntax to declare a constant is:

constant_name CONSTANT *data_type* := VALUE;

- constant_name is the name of the constant.
- datatype is a valid PL/SQL datatype.
- CONSTANT is a reserved word that ensures that the value of the memory location does not change.
- value is a specification, where you can initialize the constant.

For example:

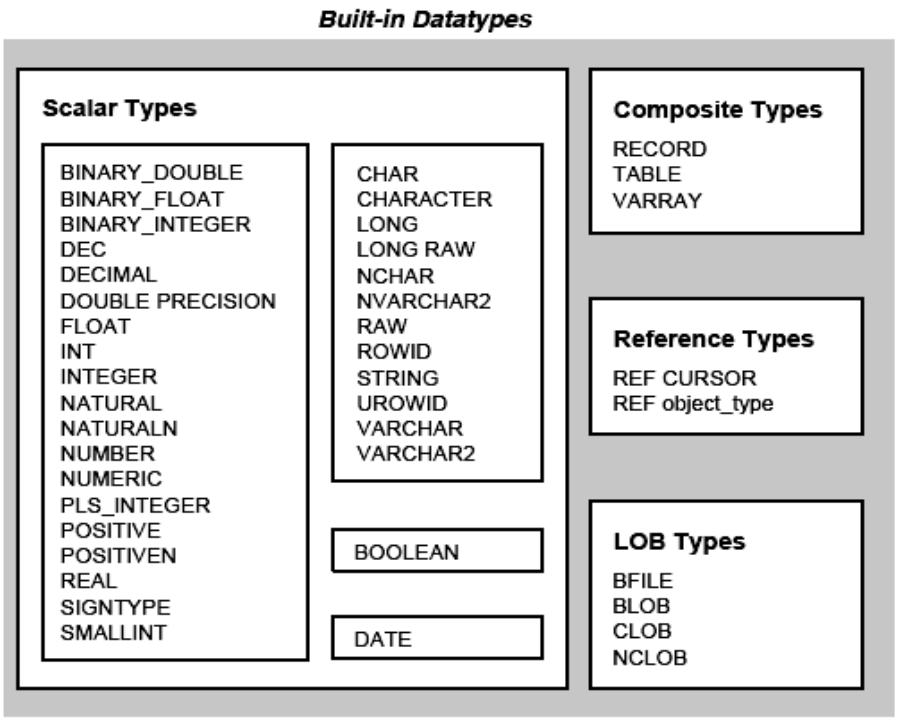
pi CONSTANT NUMBER (5,4) := 3.1412;

6.6 PL/SQL DATA TYPES

Every PL/SQL constant, variable, parameter, and function return value has a data type that determines its storage format and its valid values and operations.

Data types in PL/SQL can be divided into:

- Scalar Data Types – used to store single value with no internal components. Examples: boolean, number, etc.
- Composite Data Types – used to store values that have internal components. Examples: collections and records.
- Reference Data Types – used to hold values, called pointers, used to designate other program items.
- LOB Data Types – use to hold values, called LOB locators, that specify the location of large objects, such as text blocks or graphic images, that are stored separately from other database data.



Note: In this chapter, we will discuss only about scalar data types. The remaining data types will be discussed in further chapters.

6.6.1 SCALAR DATA TYPE

A scalar type has no internal components. It holds a single value, such as number or character string. The scalar types fall into 4 families, which store **numeric**, **character**, **boolean**, & **datetime** data.

6.6.1.1 Character Data Type

The PL/SQL data type Character stores character (alphanumeric) data in strings. The most commonly used character data type is VARCHAR2, which is the most efficient option for storing character data. The different character data types available for use in PL/SQL programs are

CHAR, NCHAR, VARCHAR2, NVARCHAR2, LONG, RAW, STRING, ROWID etc.

6.6.1.2 Numeric Data Type

The PL/SQL data type Numeric stores fixed and floating-point numbers, zero, and infinity. Some numeric types also store values that are the undefined result of an operation, which is known as "not a number" or NAN. The most commonly used numeric data type is NUMBER. The different numeric data types available for use in PL/SQL programs are

BINARY_FLOAT, BINARY_INTEGER, DECIMAL, FLOAT, INTEGER, NUMBER, PLS_INTEGER, REAL, SMALLINT, NATURAL, etc.

6.6.1.3 Boolean Data Type

The PL/SQL data type BOOLEAN stores logical values, which are the Boolean values TRUE and FALSE and the value NULL. NULL represents an unknown value.

The only value that you can assign to a BOOLEAN variable is a BOOLEAN expression. Because SQL has no data type equivalent to BOOLEAN, you cannot:

- Assign a BOOLEAN value to a database table column.
- Select or fetch the value of a database table column into a BOOLEAN variable.
- Use a BOOLEAN value in a SQL statement, SQL function, or PL/SQL function invoked from a SQL statement.

6.6.1.4 DateTime Data Type

The PL/SQL data type DateTime stores date and time values with fractional precision of seconds. The different DateTime data types available for use in PL/SQL programs are

DATE, TIME, TIMESTAMP, TIMESTAMP WITH TIME ZONE, INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND, etc.

6.7 %TYPE ATTRIBUTE

In general, the variables that deal with table columns should have the same datatype as the column itself. Rather than look it up, the developer can use PL/SQL's special syntactic feature that allows the developer simply to identify the table column to which this variable's datatype should correspond. This syntax uses a special keyword known as **%TYPE**.

The %TYPE attribute lets you declare a data item of the same data type as a previously declared variable or column (without knowing what that type is). The item declared with %TYPE is the **referencing item**, and the previously declared item is the **referenced item**.

The referencing item inherits the following from the referenced item:

- Data type and size
- Constraints (unless the referenced item is a column)

The referencing item does not inherit the initial value of the referenced item.

If the declaration of the referenced item changes, then the declaration of the referencing item changes accordingly.

The %TYPE attribute is particularly useful when declaring variables to hold database values. The syntax for declaring a variable of the same type as a column is:

variable_name table_name.column_name%TYPE;

For Example, Declaring Variable of Same Type as Column

```
DECLARE
  surname  employees.last_name%TYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('surname=' || surname);
END;
/
```

6.8 USING BIND VARIABLES

With a regular variable, each time a new entry occurs at the cache even if the statements are similar. Each new statement must be verified, parsed and have an execution plan generated and stored. Further holding so many similar statements is a waste of memory.

Bind variable is a special type of variable that has several advantages over a regular variable.

First, SQL statements that use bind variables usually run more efficiently than SQL statements that use regular variables. As the bind variables reuse the SQL statements by only changing the value of the bind variable resulting in faster and efficient processing.

Second, a bind variable can be used in regular SQL statements that are executed outside of the PL/SQL block.

Third, a bind variable stays in scope for an entire script. More accurately, the bind variable stays in scope for the entire session. As a result, you can use a bind variable across all statements in a script.

You can **create** bind variables with the **VARIABLE** command. The syntax is:

```
VARIABLE variable_name data_type;
```

For example,

```
VARIABLE ret_value NUMBER;
```

After you declare a bind variable, you can reference it by prefacing it with a **colon (:)**. For example,

```
:ret_val := 1;
```

6.9 SEQUENCES IN PL/SQL EXPRESSIONS

Sequences in PL/SQL expressions execute in a similar way as sequences in SQL expressions. After a sequence is created, you can access its values in SQL statements with the CURRVAL pseudocolumn, which returns the current value of the sequence, or the NEXTVAL pseudocolumn, which increments the sequence and returns the new value.

Earlier to Oracle 11g referring to Sequence values within PL/SQL required using SELECT INTO variable clause which is not very intuitive.

Consider the following PL/SQL code:

CREATE SEQUENCE seq_temp START WITH 100 INCREMENT BY 1;

```
SET SERVEROUTPUT ON

DECLARE

    a number;

BEGIN

    SELECT seq_temp.NEXTVAL INTO a FROM dual;

        DBMS_OUTPUT.PUT_LINE(a);

END;

/
```

Note that before Oracle 11g we could not assign a sequence value to a variablein PL/SQL. The only way to use sequence values within PL/SQL was to use SELECT sequence.NEXTVAL INTO variable.

Oracle 11g allows to refer to sequence values in a very intuitive way using variables. Consider the above code rewritten as follows:

```
CREATE SEQUENCE seq_temp START WITH 100 INCREMENT BY 1;

SET SERVEROUTPUT ON

DECLARE

    a number;

BEGIN

    a := seq_temp.NEXTVAL;

    DBMS_OUTPUT.PUT_LINE(a);

END;

/
```

As of Oracle Database 11*g* Release 1, you can use *sequence_name*.CURRVAL and *sequence_name*.NEXTVAL in a PL/SQL expression wherever you can use a NUMBER expression. However:

- Using *sequence_name*.CURRVAL or *sequence_name*.NEXTVAL to provide a default value for an ADT method parameter causes a compilation error.
- PL/SQL evaluates every occurrence of *sequence_name*.CURRVAL and *sequence_name*.NEXTVAL (unlike SQL, which evaluates a sequence expression for every row in which it appears).

N**ote:** Each time you reference *sequence_name*.NEXTVAL, the sequence is incremented immediately and permanently, whether you commit or roll back the transaction.

6.10 SUMMARY

- PL/SQL was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.
- PL/SQL is a combination of SQL along with the procedural features of programming languages.
- The PL/SQL engine is the tool used to define, compile, and run PL/SQL program units.
- The basic unit in PL/SQL is a block, which groups related declarations and statements. All PL/SQL programs are made up of blocks, which can be nested within each other.
- PL/SQL blocks are categorized into subprograms (procedures and functions) and anonymous block.
- A PL/SQL Block consists of the following sections:
  - The Header section.
  - The Declaration section.
  - The Execution section.
  - The Exception (Error Handling) section.
- A PL/SQL subprogram is a named PL/SQL block that is either a procedure or a function.
- An anonymous block is an unnamed, non-persistent PL/SQL unit.
- Identifiers are used to name PL/SQL program items & units.
- Identifiers can be divided into the following types:
  - Reserved words and keywords
  - Predefined identifiers
  - User-defined identifiers
- Oracle defines placeholders to store data temporarily, which are used to manipulate data during the execution of a PL SQL block.

- Placeholders can be any of Variables, Constants and Records.

- Every PL/SQL constant, variable, parameter, and function return value has a data type that determines its storage format and its valid values and operations.

- Data types in PL/SQL can be divided into
  - Scalar Data Types
  - Composite Data Types
  - Reference Data Types
  - LOB Data Types

- A scalar type has no internal components. It holds a single value, such as number or character string.

- The scalar types fall into 4 families, which store numeric, character, boolean, & datetime data.

- The %TYPE attribute lets you declare a data item of the same data type as a previously declared variable or column (without knowing what that type is).

- Bind variable is a special type of variable that runs more efficiently than SQL statements that use regular variables.

- Sequences in PL/SQL expressions execute in a similar way as sequences in SQL expressions using the CURRVAL pseudocolumn and the NEXTVAL pseudocolumn.

6.11 REVIEW QUESTIONS

- Explain PL/SQL engine in detail.
- List and explain the advantages of PL/SQL.
- What is a PL/SQL block. Explain its types.
- Explain the structure of PL/SQL block.
- Explain PL/SQL anonymous block.
- What is an identifier? What are the rules for naming them.
- Explain the different types of identifiers.
- Explain PL/SQL variables in detail.
- Explain the different data types in PL/SQL.

- Explain the different types of scalar data types used in PL/SQL.
- Why is %TYPE attribute used?

6.12 BIBLIOGRAPHY, REFERENCES AND FURTHER READING

- Database Management Systems, Third Edition by RamaKrishnan, Gehre. McGraw Hill
- Database System Concepts, Fifth Edition by Silberschatz, Korth, Sudarshan. McGraw Hill
- Murach's Oracle SQL and PL/SQL by Joel Murach. Shroff Publishers & Distributors
- Oracle Database 11g by Satish Asnani. PHI Learning Private Limited
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, Tata McGraw-Hill
- SQL, PL/SQL The programming language of Oracle, Bayross Ivan, BPB Publications
- Fundamentals of Database Systems, Elmasri Ramez and Navathe B. Shamkant, Pearson

6.13 ONLINE REFERENCES

O'Reilly "Mastering Oracle SQL"

http://oreilly.com/catalog/mastorasql/

Oracle Database PL/SQL language Reference 11g Release 2 (11.2), part number E25519-05

http://docs.oracle.com/cd/E11882_01/appdev.920/a96590/adg09dyn.htm

Chapter 5 Introducing PL/SQL

http://www.cs.kent.edu/~wfan/link/dbapre/dbatest/54905f.htm

Oracle SQL & PL/SQL

http://sql-plsql.blogspot.in/2007/05/oracle-plsql-cursors-with-parameters.html

http://sql-plsql.blogspot.in/2007/03/plsql-introduction.html

PL/SQL tutorial

http://plsql-tutorial.com/index.htm

http://www.academictutorials.com/pl-sql/introduction.asp

Wikipedia links

http://en.wikipedia.org/wiki/PL/SQL

7

WRITING EXECUTABLE STATEMENTS

Unit Structure

7.0 OBJECTIVES

At the end of this chapter you will be able to:

(23) Understand guidelines for PL/SQL block syntax
(24) Convert Data Types
(25) Use Nested Blocks
(26) Add Comments to your code
(27) Understand Operators used in PL/SQL

7.1 INTRODUCTION

PL/SQL stands for Procedural Language/SQL. PL/SQL extends SQL by adding constructs found in procedural languages, resulting in a structural language that is more powerful than SQL. In other words, PL/SQL is a **procedural** programming language that enhances the functionalities of SQL. We can write various types of queries with SQL statements but with PL/SQL we can group the related queries together to perform a particular task or an activity.

PL/SQL offers many advantages over other programming languages for handling the logic and business rule enforcement of database applications. It is a straightforward language with all the common logic constructs associated with a programming language, plus many things other languages don't have, such as robust error handling and modularization of code blocks. The PL/SQL code used to **interface** with the database is also stored directly on the Oracle database, and is the only programming language that interfaces with the Oracle database natively and within the database environment.

7.2 BASIC GUIDELINES FOR PL/SQL BLOCK SYNTAX

We have already discussed the basic steps of how to create a PL/SQL block. Following are certain guidelines to PL/SQL programming to improve readability and maintainability of code.

7.2.1 Rules of block Structure

- Every unit of PL/SQL must constitute a **block**. As a minimum there must be the delimiting words BEGIN and END around the executable statements.
- **SELECT** statements within PL/SQL blocks are embedded SQL (an ANSI category). As such they must return one row only. SELECT statements that return no rows or more than one row will generate an error. If you want to deal with groups of rows you must place the returned data into a cursor. The INTO clause is mandatory for

SELECT statements within PL/SQL blocks (which are not within a cursor definition), you must store the returned values from a SELECT.

- If PL/SQL variables or objects are defined for use in a block then you must also have a **DECLARE** section.
- If you include an **EXCEPTION** section the statements within it are only processed if the condition to which they refer occurs. Block execution is terminated after an exception handling routine is executed.
- PL/SQL blocks may be nested, **nesting** can occur wherever an executable statement could be placed (including the EXCEPTION section).

7.2.2 Scope and Visibility of Identifiers

The scope of an identifier is the region of a PL/SQL unit from which you can reference the identifier. The visibility of an identifier is the region of a PL/SQL unit from which you can reference the identifier without qualifying it. An identifier is local to the PL/SQL unit that declares it. If that unit has subunits, the identifier is global to them.

If a subunit redeclares a global identifier, then inside the subunit, both identifiers are in scope, but only the local identifier is visible. To reference the global identifier, the subunit must qualify it with the name of the unit that declared it. If that unit has no name, then the subunit cannot reference the global identifier.

A PL/SQL unit cannot reference identifiers declared in other units at the same level, because those identifiers are neither local nor global to the block.

7.2.3 Handling variables in PL/SQL

22. Variables must be declared first before the usage. The PL/SQL variables can be a scalar type such as DATE, NUMBER, VARCHAR2, BOOLEAN, LONG and CHAR, or a composite type, such as VARRAY.
23. Only TRUE, FALSE or NULL can be assigned to BOOLEAN type of variables.
24. AND, OR, NOT operators can be used to connect BOOLEAN values.
25. %TYPE attribute can be used to define a variable which is of the same type as a database column's type definition.

7.2.4 Guidelines for PL/SQL coding

- Capitalize all keywords, and use lowercase for the other code in a PL/SQL statement.
- Separate the words in names with underscores, as in last_name, first_name.
- Start each clause on new line.
- Break long clauses into multiple lines and indent continued lines.
- Use comments only for portions of code that are difficult to

understand. Make sure comments are correct and up-to-date.

- Line breaks, white space, indentation and capitalization have no effect on the operation of statement.
- Comments can be used to document what a statement does or what specific parts of a statement do.
- Variables and function identifiers should not have the same name as a database column name.
- Identifiers must begin with an alphabet.
- Code blocks can be nested.
- It is recommended that variable names are prefixed by v_, and parameter names in procedures/functions are prefixed by _p.

7.3 COMMENTS IN CODE

When executing PL/SQL program, **comments** are ignored by the PL/SQL compiler. Adding comments promotes readability and maintainability of the code. Comments are also used to provide information about the various logics applied for better understanding of the program code. Some programmers even specify the name and date on which a program was developed along with the purpose for proper documentation. Comments cannot be nested within each other. PL/SQL supports two types of comments.

- **Single-Line comments:** Begins with double hyphen (--) anywhere on a line and extends to the end of line.

- **Multi-Line comments:** Begins with slash-asterisk (/*), ends with an asterisk-slash (*/), and can span multiple rows.

The following program illustrates the use of comments to improve readability of the PL/SQL program.

```
DECLARE
  some_condition  BOOLEAN;
  pi              NUMBER := 3.1415926;
  radius          NUMBER := 15;
  area            NUMBER;
BEGIN
  -- Perform some simple tests and assignments

  IF 2 + 2 = 4 THEN
    some_condition := TRUE;
  /* We expect this THEN to always be performed */
  END IF;
```

```
 /* This line computes the area of a circle using pi,
 which is the ratio between the circumference and diameter.
 After the area is computed, the result is displayed. */

 area := pi * radius**2;
 DBMS_OUTPUT.PUT_LINE('The area is: ' || TO_CHAR(area));
END;
/
```

7.4 NESTED BLOCKS

The **block**, which groups related declarations and statements, is the basic unit of a PL/SQL source program. It has an optional declarative part, a required executable part, and an optional exception-handling part. Declarations are local to the block and cease to exist when the block completes execution. Blocks can be nested. Because a block is an **executable statement**, it can appear in another block wherever an executable statement is allowed.

PL/SQL allows the **nesting** of blocks within blocks i.e., the Execution section of an outer block can contain inner blocks. Therefore, a variable which is accessible to an outer block is also accessible to all nested inner blocks. The variables declared in the inner blocks are not accessible to outer blocks. Based on their declaration we can classify variables into two types.

- **Local variables** - These are declared in a inner block and cannot be referenced by outside Blocks.
- **Global variables** - These are declared in a outer block and can be referenced by its itself and by its inner blocks.

Consider the following program.

We are creating two variables (x, y) in the outer block and using them in the inner block. We are also creating two variables (v_sum, v_avg) in the inner block and using them in the inner block. The variables created in the outer block are global variables and can be used in any part of the program. While variables declared in the inner block cannot be accessible in the outer block.

```
SET SERVEROUTPUT ON
DECLARE
        x NUMBER (10);
        y NUMBER (10);
```

```
BEGIN
    DECLARE
        v_sum NUMBER (10);
        v_avg NUMBER (10,2);
    BEGIN
        x := 10;
        y := 20;
        v_sum := x+y;
        DBMS_OUTPUT.PUT_LINE('sum := '|| v_sum);
        v_avg := (x+y)/2;
        DBMS_OUTPUT.PUT_LINE('average := '|| v_avg);
    END;
END;
```

7.5 DATA CONVERSION

Oracle provides conversion functions that will easily convert values of one data type to another. Two types of **conversion** are allowed in Oracle.

- **Implicit Conversion**
- **Explicit Conversion**

Oracle will **automatically** perform **implicit conversions** of data types for you, but it can lead to performance problems in your applications. Oracle can perform **explicit conversions** by using **conversion functions** discussed in Chapter 2. The following program illustrates the implicit conversion from NUMBER to VARCHAR2 and from VARCHAR2 to DATE in Oracle.

```
SET SERVEROUTPUT ON
/* To test implicit conversion in Oracle */
DECLARE
    x NUMBER (10);        -- declare number 1
    y NUMBER (10);        -- declare number 2
    z NUMBER (10);
    p VARCHAR2 (10);    -- declare p as character variable
    q VARCHAR2 (10);
    r VARCHAR2 (10);
    d DATE;
    cd VARCHAR2 (20);
```

```
BEGIN
        x := 10;
        y := 20;
        z := x+y;--adding 2 numeric values and storing result in z
        p := x;
        q := y;
        r := p+q;--adding 2 character values and storing result in r
        cd := '10-NOV-2011';
        d := cd; --converts character data into date type of format

        -- printing sum of two numbers
        DBMS_OUTPUT.PUT_LINE('sum of x and y := '|| z);

        -- printing sum of two VARCHAR2 data type values
        DBMS_OUTPUT.PUT_LINE('sum of p and q := '|| r);

        -- printing date in system format
        DBMS_OUTPUT.PUT_LINE('Meeting is on '|| d);
END;
/
```

7.5.1 Conversion Functions

Oracle has several built-in functions that are designed to convert information from one data type to another data type. The most commonly used conversion functions are:

Function Name	Description	Example
TO_CHAR()	The TO_CHAR() conversion function converts both numerical values and date values to data type VARCHAR2.	TO_CHAR(10101)
TO_DATE()	The TO_DATE() conversion function converts character data to data type DATE.	TO_DATE('31/07/2009')
TO_NUMBER	The TO_NUMBER conversion function converts character data to data type number.	TO_NUMBER('1010')

7.6 OPERATORS IN PL/SQL

To perform various mathematical, logical and comparison operations on variables and constants we need operators. An **operator** manipulates data items and returns a result. Syntactically, an operator appears before or after an operand or between two operands. Operators are represented by special characters or by keywords. For example, the multiplication operator is represented by an asterisk (*). Operators manipulate individual data items called **operands** or **arguments**. An **operand** can be a variable, constant, literal, operator, function invocation, or placeholder—or another expression.

PL/SQL operators can be divided into the following categories:

- Mathematical Operators
- Comparison/Relational Operators
- Logical/Boolean Operators
- Special Operators

7.6.1 Mathematical Operators

Mathematical Operators are used for computational purposes.

Operator	Description
**	Exponentiation Operator. Computes to the power of a given number.
*	Multiplication Operator.
/	Division Operator.
+	Addition Operator. Add two or more operands.
-	Subtraction Operator. Subtract two or more operands.
-	Negation Operator. Denotes a negative expression.
+	Plus Operator. Denotes a positive expression.

7.6.2 Comparison or Relational Operators

These operators are used to compare values.

Operator	Description
=	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
<>, !=, ~=, ^=	Not equal to

7.6.3 Logical or Boolean Operators

If we want to check for more than one condition in a single select query then we need to use Boolean operators. Boolean operators can return only True or False for a given condition.

Operator	Description
AND	If any condition is False, then it returns False else True.
OR	If any condition is True, then it returns True else False.
NOT	Negates the given condition.

Following is the truth table for logical operators.

X	Y	X AND Y	X OR Y	NOT X
TRUE	TRUE	TRUE	TRUE	FALSE

TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

7.6.4 Special Operators

Oracle provides some special operators which enhance the capabilities of SQL statements. The special operators are: IN, IS NULL, LIKE, BETWEEN and || (Concatenation).

Operator	Description
IN	Used to compare multiple values. Returns TRUE if value lies within the specified set. Can be used with the NOT boolean operator to negate the given condition.
BETWEEN	Used to check whether a value lies in a specified range.
LIKE	Used for pattern matching searches and returns TRUE if the value matches the pattern and FALSE if it does not.
IS NULL	Returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL.
||	Used to concatenate character strings.

7.7 SUMMARY

- PL/SQL is a procedural programming language that enhances the functionalities of SQL.
- We can write various types of queries with SQL statements but with PL/SQL we can group the related queries together to perform a particular task or an activity.
- PL/SQL is the only programming language that interfaces with the Oracle database natively and within the database environment.

- Every unit of PL/SQL must constitute a block. As a minimum there must be the delimiting words BEGIN and END around the executable statements.

- If PL/SQL variables or objects are defined for use in a block then you must also have a DECLARE section.

- PL/SQL blocks may be nested, nesting can occur wherever an executable statement could be placed (including the EXCEPTION section).

- The scope of an identifier is the region of a PL/SQL unit from which you can reference the identifier.

- The visibility of an identifier is the region of a PL/SQL unit from which you can reference the identifier without qualifying it.

- An identifier is local to the PL/SQL unit that declares it. If that unit has subunits, the identifier is global to them.

- Variables must be declared first before the usage.

- %TYPE attribute can be used to define a variable which is of the same type as a database column's type definition.

- When executing PL/SQL program, comments are ignored by the PL/SQL compiler.

- Adding comments promotes readability and maintainability of the code.

- PL/SQL supports two types of comments: Single-Line comments and Multi-Line comments.

- PL/SQL allows the nesting of blocks within blocks i.e., the Execution section of an outer block can contain inner blocks.

- Based on their declaration we can classify variables into two types: Local variables and Global variables.
- Two types of conversion, from one data type to another, are allowed in Oracle: Implicit Conversion and Explicit Conversion.

- An operator manipulates data items and returns a result. Operators are represented by special characters or by keywords.

- PL/SQL operators can be divided into the following categories

    - Mathematical Operators
    - Comparison/Relational Operators
    - Logical/Boolean Operators
    - Special Operators

7.8 REVIEW QUESTIONS

- Write the various categories of operators used in PL/SQL.
- How is explicit conversion different from implicit conversion?
- Discuss about the scope of variables in the nested block of PL/SQL.
- Discuss the rules about the block structure in PL/SQL.
- Write a short note on comments in PL/SQL.

7.9 LAB ASSIGNMENTS

1. Write an anonymous PL/SQL block to take the first name and last name from the user and print the variables using concatenation operator.

Assume records in the EMP table as shown below.

EMPNO	ENAME	HIREDATE	DEPTNO	GENDER	SALARY	COMM
111	Satish	19-DEC-2008	10	M	10,000	1000
222	Rashmi	01-JAN-1987	20	F	8000	550
333	Rishi	05-JUN-1976	10	M	7000	450
444	Anil	16-APR-1967	10	M	12,000	2000
555	Anita	-	30	F	-	1000
666	Nilesh	20-MAY-1987	20	M	13,000	-
777	Ruchi	11-JUN-2000	30	F	-	-
888	Sarika	-	-	F	-	-

- Retrieve ALL records from the EMP table.
- Retrieve the empno, ename and salary for all employees.

- Retrieve the deptno, ename, salary and comm for all employees.
- Retrieve the deptno, empno and total salary as salary + 10% of (salary+comm) for all employees.
- List the employees who have joined after 1st May 1987.
- List the male employees of department number 10 and 30.
- Display the employees who earned commission more than 1000 and belong to department 10 or have salary more than 10,000/- but do not belong to department 10.
- List employees with zero salary.
- List employees with either salary or commission unknown.
- Display the employees whose name contains 'SA'.
- Display the salary of employees as ZERO if unknown.

7.10 BIBLIOGRAPHY, REFERENCES AND FURTHER READING

- Database Management Systems, Third Edition by RamaKrishnan, Gehre. McGraw Hill
- Database System Concepts, Fifth Edition by Silberschatz, Korth, Sudarshan. McGraw Hill
- Murach's Oracle SQL and PL/SQL by Joel Murach. Shroff Publishers & Distributors
- Oracle Database 11g by Satish Asnani. PHI Learning Private Limited
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, Tata McGraw-Hill
- SQL, PL/SQL The programming language of Oracle, Bayross Ivan, BPB Publications
- Fundamentals of Database Systems, Elmasri Ramez and Navathe B. Shamkant, Pearson

7.11 ONLINE REFERENCES

O'Reilly "Mastering Oracle SQL"

http://oreilly.com/catalog/mastorasql/

Oracle Database PL/SQL language Reference 11g Release 2 (11.2), part number E25519-05

http://docs.oracle.com/cd/E11882_01/appdev.920/a96590/adg09dyn.htm

Chapter 5 Introducing PL/SQL

http://www.cs.kent.edu/~wfan/link/dbapre/dbatest/54905f.htm

Oracle SQL & PL/SQL

http://sql-plsql.blogspot.in/2007/05/oracle-plsql-cursors-with-parameters.html

http://sql-plsql.blogspot.in/2007/03/plsql-introduction.html

PL/SQL tutorial

http://plsql-tutorial.com/index.htm

http://www.academictutorials.com/pl-sql/introduction.asp

Wikipedia links

http://en.wikipedia.org/wiki/PL/SQL

8

INTERACTION WITH THE ORACLE SERVER

Unit Structure

8.0 OBJECTIVES

At the end of this chapter you will be able to:

- Invoke SELECT Statements in PL/SQL,
- Manipulate Data in the Server using PL/SQL,

- Understand the SQL Cursor concept,
- Use SQL Cursor Attributes to Obtain Feedback on DML,
- Save and Discard Transactions.

8.1 INTRODUCTION

No usage of PL/SQL is complete without presenting the ease of use involved in interacting with the Oracle database. Any data manipulation or change operation can be accomplished within PL/SQL without the additional overhead typically required in other programming environments. There is no ODBC interface, and no embedding is required for use of database manipulation with PL/SQL.

In this chapter we will invoke SELECT statements in PL/SQL and also manipulate data in the server. Then we will discuss the concept of cursor in SQL along with how to use cursor attributes in DML. Finally we shall have a look on how to save and discard transactions in a database.

8.2 INVOKING SELECT STATEMENT

The **SELECT** statement in SQL is used to extract data from the one or more tables or views in the database. However, the SQL SELECT statement cannot be used inside a PL/SQL block to retrieve records. In order to retrieve data from one or more tables in Pl/SQL we need to use the SELECT INTO statement.

The **SELECT INTO** statement retrieves values from one or more database tables (as the SQL SELECT statement does) and stores them in variables (which the SQL SELECT statement does not do). Using the INTO clause we can specify the variables or record in which to store the column values that the statement returns.

Syntax:

SELECT <column_name1>, <column_name2> **INTO** <var_name1>, <var_name2>

FROM table_name;

8.2.1 Restrictions in SELECT INTO clause:

(28) Columns selected in the query must be returned into local variables.
(29) The variables used must be matching with the columns in data type.
(30) It is used in the executable section (BEGIN ..... END) of a PL/SQL block.
(31) The SELECT query must return only one row.

Consider the following example:

To get the details of employees who work for the HR department and gets salary of 30000.

```
SET SERVEROUTPUT ON

DECLARE

    v_name emp_detail.ename%TYPE;

    v_age emp_detail.age%TYPE;

BEGIN

    SELECT ename, age INTO v_name, V_age

    FROM emp_detail WHERE empid =

        ( SELECT empid FROM emp

        WHERE dept = 'HR' AND salary = 30000);


DBMS_OUTPUT.PUT_LINE ('name: ' || v_name || 'Age: ' || v_age);

END;

/
```

8.3 DATA MANIPULATION IN THE SERVER USING PL/SQL

PL/SQL blocks can be used to **manipulate data** in the server using INSERT, DELETE and UPDATE statements. PL/SQL provides the following functionalities:

- Using SQL data manipulation languages in PL/SQL, data in the tables can be manipulated.
- Using SQL data transactional commands in PL/SQL, changes of data in the tables can be made permanent or revoked.
- All sorts of SQL functions can be used in PL/SQL.
- All operators that we are using in SQL statements can also be used in PL/SQL.
- We can declare variables and constants in PL/SQL to store results of a query during the process, which can be used later in PL/SQL block.

In the following program, a row is added, updated and deleted from the EMPLOYEE table using variables in PL/SQL to manipulate data in the database tables.

```
SET SERVEROUTPUT ON

DECLARE
        My_employee employee%ROWTYPE;
        My_lastname VARCHAR2 := 'SAMSON';
        My_firstname VARCHAR2 := 'DELILAH';
        My_salary NUMBER := 49500;
BEGIN
        SELECT *
        INTO my_employee
        FROM employee
        WHERE empid = 49594;
        UPDATE employee
        SET salary = my_employee.my_salary + 10000
        WHERE empid = my_employee.my_empid;

INSERT INTO employee (empid, lastname, firstname, salary)
        VALUES (emp_sequence.nextval, my_lastname, my_firstname,
        my_salary);

        My_empid := 59495;

        DELETE FROM employee
        WHERE empid = my_empid;
END;

/
```

8.4 SQL CURSOR CONCEPT

Oracle performs a set of tasks for executing any SQL statement.

- Reserves an area in memory called private SQL area.
- Populates this area with appropriate data.
- Frees the memory area when execution completes.

Cursor is a pointer to the private SQL area that stores information about processing a specific SELECT or DML statement. Within PL/SQL a SELECT statement cannot return more than one row at a time. So in order to process some group of rows for implementing certain logic to all the records of that group, we need to use cursors. The set of rows retrieved is called the ***active set***; its size depends on how many rows meet the query search condition.

For SQL statements, there are two types of cursors:

- *Implicit cursor*
- *Explicit cursor*

In this chapter we will discuss about implicit cursors. Explicit cursors have been explained in the later chapters.

8.4.1 Implicit Cursors

An **implicit cursor** is a session cursor that is constructed and managed by PL/SQL. PL/SQL opens an implicit cursor every time you run a SELECT or DML statement. You cannot control an implicit cursor, but you can get information from its attributes. An implicit cursor closes after its associated statement runs; however, its attribute values remain available until another SELECT or DML statement runs. **Cursor attributes return information about the state of the cursor.**

The syntax for the value of an implicit cursor attribute is **SQL*attribute*** (for example, SQL%FOUND). SQL*attribute* always refers to the most recently run DML or SELECT INTO statement.

The **implicit cursor attributes** are:

- **SQL%ISOPEN** Attribute: Is the Cursor Open?
- **SQL%FOUND** Attribute: Were Any Rows Affected?
- **SQL%NOTFOUND** Attribute: Were No Rows Affected?
- **SQL%ROWCOUNT** Attribute: How Many Rows Were Affected?

The table given below lists the cursor attributes and the values that they can return.

Cursor Attribute	Cursor Variable	Values for Cursor
%FOUND	SQL%FOUND	If no DML or SELECT INTO statement has run, NULL. If the most recent DML or SELECT INTO statement returned a row, TRUE. If the most recent DML or SELECT INTO statement did not return a row, FALSE.
%NOTFOUND	SQL%NOTFOUND	If no DML or SELECT INTO statement has run, NULL. If the most recent DML or SELECT INTO statement returned a row, FALSE. If the most recent DML or SELECT INTO statement did

		not return a row, TRUE.
%ROWCOUNT	SQL%ROWCOUNT	NULL if no DML or SELECT INTO statement has run; otherwise, a number greater than or equal to zero.
%ISOPEN	SQL%ISOPEN	Always FALSE. The Oracle database automatically opens and closes the implicit cursor associated with any DML or SELECT statement and so SQL%ISOPEN always returns FALSE.

Consider the following program where we will display the values of various CURSOR attributes.

```
SET SERVEROUTPUT ON

BEGIN

        UPDATE tbl_bank_account SET status = 'INACTIVE'

        WHERE branch = 'SAKI NAKA';

                IF SQL%FOUND THEN

DBMS_OUTPUT.PUT_LINE ('Accounts Found for branch Saki Naka');

        END IF;

                IF SQL%NOTFOUND THEN

                DBMS_OUTPUT.PUT_LINE ('No Accounts Found for
branch Saki Naka');

        END IF;

                IF SQL%ROWCOUNT > 0 THEN

                DBMS_OUTPUT.PUT_LINE (SQL%ROWCOUNT ' ||
account(s) inactivated');

        ELSE

                DBMS_OUTPUT.PUT_LINE ('No account(s) inactivated');

        END IF;

END;

/
```

8.5 SAVE AND DISCARD TRANSACTIONS

A **transaction** is a logical, atomic unit of work that contains one or more SQL statements. A transaction groups SQL statements so that they are either all **committed**, which means they are applied to the database, or all **rolled back**, which means they are undone from the database.

Transaction processing is a feature that lets multiple users work on the database concurrently, and ensures that each user sees a consistent version of data and that all changes are applied in the right order.

A database transaction consists of one or more statements. Specifically, a transaction consists of one of the following:

- One or more data manipulation language (DML) statements that together constitute an atomic change to the database
- One data definition language (DDL) statement

Now each and every transaction has a specific beginning and end point. A transaction begins when the user connects to the database and performs the first DML statement or when the last transaction has ended and a new DML query has started to execute. The COMMIT operation is automatically executed when you execute a DDL or DCL or the user exits normally by typing EXIT statement. In contrast, in order to guarantee that the changes made by the DML statements are actually saved in the database, the PL/SQL code should explicitly contain a commit statement.

The three transaction specifications available in PL/SQL are **commit, roll back and savepoint**.

Transaction control is the management of changes made by DML statements and the grouping of DML statements into transactions. In general, application designers are concerned with transaction control so that work is accomplished in logical units and data is kept consistent.

Transaction control involves using the following statements:

26. The **COMMIT** statement ends the current transaction and makes all changes performed in the transaction permanent. COMMIT also erases all savepoints in the transaction and releases transaction locks.
27. The **ROLLBACK** statement reverses the work done in the current transaction; it causes all data changes since the last COMMIT or ROLLBACK to be discarded. The ROLLBACK TO SAVEPOINT statement undoes the changes since the last savepoint but does not end the entire transaction.

28. The **SAVEPOINT** statement identifies a point in a transaction to which you can later roll back.

The session in the following table illustrates the use of the above statements

Time	Session	Explanation
t0	COMMIT;	This statement ends any existing transaction in the session.
t1	SET TRANSACTION NAME 'sal_update';	This statement begins a transaction and names it sal_update.
t2	UPDATE employees SET salary = 7000 WHERE last_name = 'Sharma';	This statement updates the salary for Sharma to 7000.
t3	SAVEPOINT after_sharma_sal;	This statement creates a savepoint named after_sharma_sal, enabling changes in this transaction to be rolled back to this point.
t4	UPDATE employees SET salary = 12000 WHERE last_name = 'Naik';	This statement updates the salary for Naik to 12000.
t5	SAVEPOINT after_naik_sal;	This statement creates a savepoint named after_naik_sal, enabling changes in this transaction to be rolled back to this point.
t6	ROLLBACK TO SAVEPOINT after_sharma_sal;	This statement rolls back the transaction to t3, undoing the update to Naik's salary at t4. The sal_update transaction has *not* ended.
t7	UPDATE employees SET salary = 11000 WHERE last_name = 'Mehta';	This statement updates the salary for Mehta to 11000 in transactionsal_update.
t8	ROLLBACK;	This statement rolls back all changes in transaction sal_update, ending the transaction.

Time	Session	Explanation
t9	SET TRANSACTION NAME 'sal_update2';	This statement begins a new transaction in the session and names itsal_update2.
t10	UPDATE employees SET salary = 7050 WHERE last_name = 'Lal';	This statement updates the salary for Lal to 7050.
t11	UPDATE employees SET salary = 10950 WHERE last_name = 'Yadav';	This statement updates the salary for Yadav to 10950.
t12	COMMIT;	This statement commits all changes made in transaction sal_update2, ending the transaction. The commit guarantees that the changes are saved in the online redo log files.

Consider the following program that shows the use of COMMIT, SAVEPOINT and ROLLBACK statements.

```
DROP TABLE emp_name;
CREATE TABLE emp_name AS
  SELECT employee_id, last_name, salary
  FROM employees;

CREATE UNIQUE INDEX empname_ix
ON emp_name (employee_id);

DECLARE
  emp_id        employees.employee_id%TYPE;
  emp_lastname  employees.last_name%TYPE;
  emp_salary    employees.salary%TYPE;

BEGIN
  SELECT employee_id, last_name, salary
  INTO emp_id, emp_lastname, emp_salary
  FROM employees
  WHERE employee_id = 120;

  SAVEPOINT my_savepoint1;

  UPDATE emp_name
  SET salary = salary * 1.1
```

```
 WHERE employee_id = emp_id;

 DELETE FROM emp_name
 WHERE employee_id = 130;

 SAVEPOINT my_savepoint2;

 INSERT INTO emp_name (employee_id, last_name, salary)
 VALUES (emp_id, emp_lastname, emp_salary);

 COMMIT;

EXCEPTION
 WHEN DUP_VAL_ON_INDEX THEN
   ROLLBACK TO my_savepoint2;
   DBMS_OUTPUT.PUT_LINE('Transaction rolled back.');
END;
/
```

8.5.1 Implicit Rollbacks

Before running an INSERT, UPDATE, or DELETE statement, the database marks an implicit savepoint (unavailable to you). If the statement fails, the database rolls back to the savepoint. Usually, just the failed SQL statement is rolled back, not the whole transaction. If the statement raises an unhandled exception, the host environment determines what is rolled back.

The database can also roll back single SQL statements to break deadlocks. The database signals an error to a participating transaction and rolls back the current statement in that transaction.

Before running a SQL statement, the database must parse it, that is, examine it to ensure it follows syntax rules and refers to valid schema objects. Errors detected while running a SQL statement cause a roll back, but errors detected while parsing the statement do not.

8.5.2 Ending Transactions

You should explicitly commit or roll back every transaction. If you do not commit or roll back a transaction explicitly, then Oracle determines its final state. For example,

- A user issues a COMMIT or ROLLBACK statement *without* a SAVEPOINT clause.

  In a **commit**, a user explicitly or implicitly requested that the changes in the transaction be made permanent. Changes made by the transaction are permanent and visible to other users only after a transaction commits.

- A user runs a **DDL** command such as CREATE, DROP, RENAME, or ALTER.

  The database issues an implicit **COMMIT** statement before and after every DDL statement. If the current transaction contains DML statements, then Oracle Database first commits the transaction and then runs and commits the DDL statement as a new, single-statement transaction.

- A user exits normally from most Oracle Database utilities and tools, causing the current transaction to be implicitly committed.
- A client process terminates abnormally or executes a ROLLBACK statement, causing the transaction to be implicitly or explicitly rolled back.

8.6 SUMMARY

- The SELECT statement in SQL is used to extract data from the one or more tables or views in the database.
- The SELECT INTO statement retrieves values from one or more database tables (as the SQL SELECT statement does) and stores them in variables (which the SQL SELECT statement does not do).
- PL/SQL blocks can be used to manipulate data in the server using INSERT, DELETE and UPDATE statements.
- Oracle performs a set of tasks for executing any SQL statement.
  - Reserves an area in memory called private SQL area.
  - Populates this area with appropriate data.
  - Frees the memory area when execution completes.
- Cursor is a pointer to the private SQL area that stores information about processing a specific SELECT or DML statement.
- For SQL statements, there are two types of cursors:
  - Implicit cursor
  - Explicit cursor
- An implicit cursor is a session cursor that is constructed and managed by PL/SQL.
- You cannot control an implicit cursor, but you can get information from its attributes.
- Cursor attributes return information about the state of the cursor.
- A transaction is a logical, atomic unit of work that contains one or more SQL statements.
- Transaction processing is a feature that lets multiple users work on

the database concurrently, and ensures that each user sees a consistent version of data and that all changes are applied in the right order.

- Transaction control is the management of changes made by DML statements and the grouping of DML statements into transactions.
- Transaction control involves using the following statements:
  - The COMMIT statement ends the current transaction and makes all changes performed in the transaction permanent.
  - The ROLLBACK statement reverses the work done in the current transaction.
  - The SAVEPOINT statement identifies a point in a transaction to which you can later roll back.
- Before running an INSERT, UPDATE, or DELETE statement, the database marks an implicit savepoint (unavailable to you). If the statement fails, the database rolls back to the savepoint.
- You should explicitly commit or roll back every transaction. If you do not commit or roll back a transaction explicitly, then Oracle determines its final state.

8.7 REVIEW QUESTIONS

- Explain the concept of implicit cursor in PL/SQL.
- What is a transaction? Explain COMMIT, ROLLBACK and SAVEPOINT in transaction.
- What is a cursor? Discuss the different attributes of a cursor?

8.8 LAB ASSIGNMENT

Consider the following table schema and write a PL/SQL block performing the following:

EMPID	ENAME	DEPT	DESG	SALARY
VARCHAR2	VARCHAR2	VARCHAR2	VARCHAR2	NUMBER

- Print the EMPID, ENAME, DEPT, DESG and SALARY of the employee whose name is “Prakash”. If the record is not found then print “There is no employee with name Prakash”.
- Print the details of the employee getting lowest salary.
- Print the details of the employee getting highest salary.
- Print the average salary of the employees belonging to “Finance” department.

- Delete all the records of employees from the department “HR” getting salary less than 10,000/- and also print the total number of records deleted.
- Modify the salary of the employees in the “Admin” department to give them a hike of 25%.

8.9 BIBLIOGRAPHY, REFERENCES AND FURTHER READING

- Database Management Systems, Third Edition by RamaKrishnan, Gehre. McGraw Hill
- Database System Concepts, Fifth Edition by Silberschatz, Korth, Sudarshan. McGraw Hill
- Murach's Oracle SQL and PL/SQL by Joel Murach. Shroff Publishers & Distributors
- Oracle Database 11g by Satish Asnani. PHI Learning Private Limited
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, Tata McGraw-Hill
- SQL, PL/SQL The programming language of Oracle, Bayross Ivan, BPB Publications
- Fundamentals of Database Systems, Elmasri Ramez and Navathe B. Shamkant, Pearson

8.10 ONLINE REFERENCES

O'Reilly "Mastering Oracle SQL"

http://oreilly.com/catalog/mastorasql/

Oracle Database PL/SQL language Reference 11g Release 2 (11.2), part number E25519-05

http://docs.oracle.com/cd/E11882_01/appdev.920/a96590/adg09dyn.htm

Chapter 5 Introducing PL/SQL

http://www.cs.kent.edu/~wfan/link/dbapre/dbatest/54905f.htm

Oracle SQL & PL/SQL

http://sql-plsql.blogspot.in/2007/05/oracle-plsql-cursors-with-parameters.html

http://sql-plsql.blogspot.in/2007/03/plsql-introduction.html

PL/SQL tutorial

http://plsql-tutorial.com/index.htm

http://www.academictutorials.com/pl-sql/introduction.asp

Wikipedia links

http://en.wikipedia.org/wiki/PL/SQL

UNIT - IV

9

CONTROL STRUCTURES

Unit Structure

9.1 OBJECTIVE

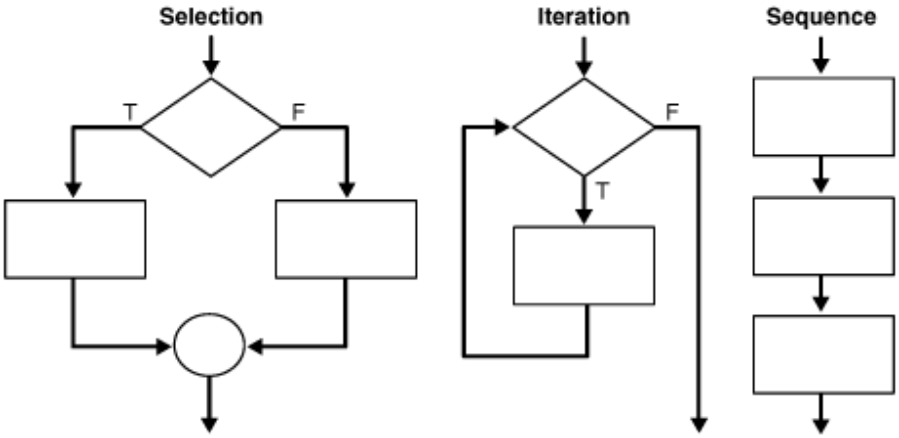
After completing this chapter, you will be able to:

- Understand how to use the conditional and looping structures using PLSQL.
- Understand how to utilize the control structures in queries.

- ❖ Aware of the syntax and examples for manipulating the database using queries with the help of various control and looping structures.
- ❖ Understand the Continue statement for updating the processes after a successful call.

9.2 CONTROL STRUCTURE

The Control Structures are used to decide the execution flow of the program depending on the programmer defined conditions. The control structures are the necessary commands over the execution of the PL/SQL program. Any computer program can be written using the basic control structures shown in Figure. They can be combined in any way necessary to deal with a given problem.



In selection structure it tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition can be any variable or expression that returns a BOOLEAN value (TRUE or FALSE). Also after the particular condition we can execute a complete block of statements. The selection statements include **IF and CASE** Statements.

The iteration structure executes a sequence of statements repeatedly as long as a condition holds true. Also here they give us the facility to increment or decrement of counters to achieve the proper results.

The sequence structure simply executes a sequence of statements in the order in which they occur. They are the simplest form of control structures which executes on by one.

9.2.1 IF and CASE Statements

A) IF Statements:

The 'IF' statements executes a sequence of statements, depending on the value of a condition. When the condition is satisfied, the necessary option will be executed. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF.
We can form the Condition using relational operators given in following table. These operators help us to define different kind of conditions or combination of conditions using following operator Symbols.

The IF & CASE statements are used in various kinds of operations on the database. These can be used to sort the data as well as to find out the exact entry from the table cell depending on its value. These two operations can used together if the need arise in operation.

Operator	Meaning
>	Greater than Operator
>=	Greater than or equal to Operator
<	Less than Operator
<=	Less than or equal to Operator
=	Equal to Operator
<>, !=, ~=, ^=	Not equal to Operator
LIKE	This Operator return true if the character pattern matches the given value.
BETWEEN..AND	These Operator returns true if the value is in the given range.
IN	This Operator returns true if the value is in the list.
IS NULL	This Operator return true if the value is NULL.

B) IF-THEN Statement:

This statement uses the simple condition to evaluate, then it executes the statement or a block of statements if the condition returns true. The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF).

The general format of an IF statement is:

```
IF condition THEN
        ex_statements
END IF;
```

Example:

```
SQL> DECLARE
  2  a NUMBER(6);
  3  b NUMBER(6);
  4  BEGIN
  5  a := 24;
  6  b := 34;
  7  IF a < b THEN
  8  DBMS_OUTPUT.PUT_LINE(a || 'is less than' || b);
  9  END IF;
 10  END;
 11  /

24 is less than 34
PL/SQL procedure successfully completed.
```

Example:

```
SQL> DECLARE
  2    a VARCHAR(12);
  3    b VARCHAR(12);
  4   BEGIN
  5    a := 'YASHASHREE';
  6    b := 'YASHASHREE';
  7    IF a LIKE b THEN
  8    DBMS_OUTPUT.PUT_LINE(a || 'is Same as ' || b);
  9    END IF;
 10     END;
 11   /

YASHASHREE  is Same as YASHASHREE

PL/SQL procedure successfully completed.a
```

C) IF-THEN-ELSE Statement:

In second type of **IF** statement it adds the keyword ELSE followed by an alternative sequence of statements. In this if the condition returns true then first block of statement executes and if the condition returns false then second block of statement executes.

The general format of an IF-THEN-ELSE statement is:

```
IF condition THEN
        true_ ex_statement;
        ELSE
        false_ ex_statement;
END IF;
```

Example:

```
SQL> DECLARE
  2    marks NUMERIC;
  3  BEGIN
  4    marks := '45';
  5    IF marks>35 THEN
  6      DBMS_OUTPUT.PUT_LINE('PASS');
  7    ELSE
  8      DBMS_OUTPUT.PUT_LINE('FAIL');
  9    END IF;
 10  END;
 11  /

PASS
PL/SQL procedure successfully completed.
```

Example:

```
SQL> DECLARE
  2    a VARCHAR(12);
  3    b VARCHAR(12);
  4   BEGIN
  5    a := 'YASHASHREE';
  6    b := 'yashashree';
  7    IF a LIKE b THEN
  8    DBMS_OUTPUT.PUT_LINE(a || 'is Same as ' || b);
  9   ELSE
 10    DBMS_OUTPUT.PUT_LINE(a || 'is not Same as ' || b);
 11  END IF;
 12
 13    END;
 14   /

YASHASHREE    is not Same as yashashree

PL/SQL procedure successfully completed.
```

D) NESTED IF-THEN- ELSE statements:

We can also put IF statements inside other IF statements. This statement gives us ability to check more than one condition in

a series and execute the appropriate block of statements depending on the conditions.
The general format of an IF-THEN-ELSE statement is:

```
IF <condition1>
THEN
  ...
ELSIF <condition2>
THEN
  ...
ELSIF <condition2>
THEN
  ...
END IF;
```

Example :

```
SQL>          DECLARE
 2    marks NUMERIC;
 3  BEGIN
 4      marks := '67';
 5      IF  marks >= 75 THEN
 6     DBMS_OUTPUT.PUT_LINE('YOU GOT DISINCTION');
 7     ELSIF  marks >= 60 AND marks<75 THEN
 8     DBMS_OUTPUT.PUT_LINE('YOU GOT FIRST CLASS');
 9     ELSIF marks >= 50 AND marks<60 THEN
10     DBMS_OUTPUT.PUT_LINE('YOU GOT SECOND CLASS ');
11     ELSIF marks >= '40' AND marks<50 THEN
12     DBMS_OUTPUT.PUT_LINE('YOU GOT PASS CLASS ');
13     ELSE
14    DBMS_OUTPUT.PUT_LINE('Sorry. You are  fail..... ');
15
16    END IF;
17  END;
18   /

YOU GOT FIRST CLASS
PL/SQL procedure successfully completed.
```

Example :

```
SQL> DECLARE
  2      D VARCHAR(10):= TO_CHAR(SYSDATE,'DY');
  3    BEGIN
  4    IF  D= 'SAT' THEN
  5   DBMS_OUTPUT.PUT_LINE('ENJOY YOUR WEEKEND');
  6    ELSIF  D= 'SUN' THEN
  7   DBMS_OUTPUT.PUT_LINE('ENJOY YOUR WEEKEND');
  8    ELSE
  9  DBMS_OUTPUT.PUT_LINE('HAVE A NICE DAY ');
```

```
10     END IF;
11  END;
12  /

HAVE A NICE DAY
      PL/SQL procedure successfully completed.
```

9.3: CASE STATEMENT :

The CASE statement is a compact way to evaluate a single condition and choose between many alternative actions. This statement is very useful when we have multiple conditions as well as the expected input is also multiple and changeable according to the situation. It makes sense to use CASE when there are three or more alternatives to choose from. The CASE Statements in PL/SQL has two forms

i) Simple CASE :

In Simple CASE we have to specify a SELECTOR, which determines which group of actions to get executed. It simply executes if the match is found otherwise it executes the default statement block.

ii) Searched CASE :

In Searched Case, the SELECTOR is not present, it has search conditions that are evaluated in order to determine which group of actions to take place.

A) Simple CASE Syntax:

```
                    CASE SELECTOR
WHEN Expr1 THEN ex_statements 1;
WHEN Expr2 THEN ex_statements2;
                         :
ELSE   ex_statements n;
   END CASE;
```

Example

To compare the IF and CASE statements, consider the code in example that outputs descriptions of school grades. Note the five Boolean expressions. In each instance, we test whether the same variable, grade, is equal to one of five values: 'A', 'B', 'C', 'D', or 'F'. You can rewrite the code in above example using the CASE statement, as shown in following Example.

Example Using the CASE-WHEN Statement

```
      SQL> DECLARE
2    grade CHAR(1);
3  BEGIN
```

```
 4   grade := 'D';
 5   CASE grade
 6   WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
 7     WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
 8     WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
 9     WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
10     WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
11      ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
12    END CASE;
13  END;
14  /

Fair
PL/SQL procedure successfully completed.
```

Example:

```
SQL>  DECLARE
 2       Product VARCHAR(20);
 3     BEGIN
 4       Product := 'PEN';
 5       CASE Product
 6       WHEN 'PEN' THEN DBMS_OUTPUT.PUT_LINE('PRICE IS
         20 RS.');
 7       WHEN 'PENCIL' THEN DBMS_OUTPUT.PUT_LINE('PRICE
         IS  10 RS.');
 8     WHEN 'ERASER' THEN DBMS_OUTPUT.PUT_LINE('PRICE
         IS  5 RS.');
 9       WHEN 'SHARPNER' THEN DBMS_ OUTPUT.
         PUT_LINE('PRICE IS  10 RS.');
10        WHEN 'NOTEBOOK' THEN DBMS_OUTPUT.PUT_LINE('PRICE
         IS  25 RS.');
11      ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
12       END CASE;
13      END;
14      /

PRICE IS  20 RS.
PL/SQL procedure successfully completed.
```

B) Searched CASE Syntax:

```
                        CASE
                        WHEN  SearchCondition THEN
ex_statements 1;
                        WHEN  SearchCondition THEN
ex_statements 2;
                                :
                        END StatementN;
                        END CASE;
```

Example :

```
SQL> DECLARE
  2    grade CHAR(1):='C';
  3  apprisal VARCHAR(20);
  4  id number:=124;
  5  attnd NUMBER:=150;
  6  min_days CONSTANT NUMBER:=200;
  7  BEGIN
  8    apprisal:=
  9    CASE
 10      WHEN grade = 'F'  OR attnd<min_days
 11      THEN 'Poor'
 12      WHEN grade = 'A' THEN 'Excellent'
 13      WHEN grade = 'B' THEN 'Very Good'
 14      WHEN grade = 'C' THEN 'Good'
 15      WHEN grade = 'D' THEN 'Fair'
 16      WHEN grade = 'F' THEN 'Poor'
 17      ELSE  'No such grade'
 18      END;
 19       DBMS_OUTPUT.PUT_LINE ('Result for student '||id||' is
         '||apprisal);
 20     END;
 21  /

Result for student 124  is Poor
PL/SQL procedure successfully completed.
```

9.4 LOOP STATEMENT :

This is used to repeatedly execute a set of statements. This is the simplest form of looping structures. The loop statements also execute continuously until the exit condition is not reached.

```
LOOP
Statements;
END LOOP;
```

Loop… End Loop has no termination point. So unless we terminate loop using EXIT command (discussed next) it becomes an infinite loop.

9.5 EXIT STATEMENT :

This is used to exit out of a Loop. This is mainly used with LOOP statement, as there is no other way of terminating the LOOP.
The following is the syntax of EXIT command.

```
EXIT [WHEN condition];
```

If EXIT is used alone, it will terminate the current loop as and when it is executed.

If EXIT is used with WHEN clause, then the current loop is terminated only when the condition given after WHEN is satisfied.

We can put EXIT statements anywhere inside a loop, but not outside a loop. To complete a PL/SQL block before it reaches its normal end, use the RETURN statement.

Example:

```
SQL> DECLARE
  2  c NUMBER(6);
  3  BEGIN
  4  c := 1;
  5  LOOP
  6  DBMS_OUTPUT.PUT_LINE('aI:' || c);
  7  c := c + 1;
  8  IF c > 5 THEN
  9  EXIT;
 10  END IF;
 11  DBMS_OUTPUT.PUT_LINE('bI:' || c);
 12  END LOOP;
 13  END;
 14  /

aI:1
bI:2
aI:2
bI:3
aI:3
bI:4
aI:4
bI:5
aI:5

PL/SQL procedure successfully completed.
```

Example :

```
SQL> DECLARE
  2  num NUMBER(6);
  3  BEGIN
  4  num := 4;
  5  LOOP
  6  DBMS_OUTPUT.PUT_LINE('Table of 4:' || num);
  7  num := num + 4;
  8  IF num > 40 THEN
  9  EXIT;
 10  END IF;
 11  END LOOP;
 12  END;
 13  /
Table of 4:4
```

```
Table of 4:8
Table of 4:12
Table of 4:16
Table of 4:20
Table of 4:24
Table of 4:28
Table of 4:32
Table of 4:36
Table of 4:40

PL/SQL procedure successfully completed.
```

9.6 LABELING A PL/SQL LOOP

We are also able to give or apply the labels to PLSQL loops. The optional label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. When we nest the labeled loops, ending label names are used to improve readability.

Simply label the enclosing loop to complete. Use the label in an EXIT statement, as shown in following Example. Every enclosing loop up to and including the labeled loop is exited.

Example Using EXIT with Labeled Loops

```
SQL> DECLARE
  2   a      PLS_INTEGER := 0;
  3   b      PLS_INTEGER := 0;
  4   c      PLS_INTEGER;
  5  BEGIN
  6   <<outer >>
  7   LOOP
  8     b := b + 1;
  9     c := 0;
 10      <<inner >>
 11     LOOP
 12       c := c + 1;
 13       a := a +b * c;
 14       EXIT inner WHEN (c > 5);
 15       EXIT outer WHEN ((b *c) > 15);
 16     END LOOP inner;
 17   END LOOP outer;
```

```
 18      DBMS_OUTPUT.PUT_LINE('The sum of products is
equals to: ' || TO_CHAR(a));
 19  END;
 20  /
sum of products is equals to: 166
PL/SQL procedure successfully completed.
```

9.7: WHILE STATEMENT

The while statement execute series of statements as long as the given condition is true. When the given condition is reached the loop terminates automatically

```
WHILE condition LOOP
Statements;
END LOOP;
```

As long as the condition is true then statements will be repeatedly executed. Once the condition is false then loop is terminated.

Example

```
SQL> DECLARE
  2  a NUMBER(7);
  3  BEGIN
  4  a := 1;
  5  WHILE a <= 5
  6  LOOP
  7  DBMS_OUTPUT.PUT_LINE('Initial:' || a);
  8  a := a + 1;
  9  DBMS_OUTPUT.PUT_LINE('After:' || a);
 10  END LOOP;
 11  END;
 12  /
Initial:1
After:2
Initial:2
After:3
Initial:3
After:4
Initial:4
After:5
Initial:5
After:6
PL/SQL procedure successfully completed.
```

Example :

```
SQL>  DECLARE
  2    num NUMBER(4);
  3    BEGIN
  4    num := 10;
  5    WHILE num <= 100
```

```
 6   LOOP
 7   DBMS_OUTPUT.PUT_LINE('Table of 10 :' || num);
 8   num := num + 10;
 9   END LOOP;
10    END;
11    /
Table of 10 :10
Table of 10 :20
Table of 10 :30
Table of 10 :40
Table of 10 :50
Table of 10 :60
Table of 10 :70
Table of 10 :80
Table of 10 :90
Table of 10 :100

PL/SQL procedure successfully completed.
```

9.8: FOR STATEMENT

The FOR loops iterate over a specified range of integers. The number of iterations is known before the loop is entered. A double dot (..) serves as the range operator. The range is evaluated when the FOR loop is first entered and is never re-evaluated. If the lowerrange equals the upperrange, the loop body is executed once. Following is the syntax for the **FOR** loop.

```
FOR counter IN [REVERSE] lowerrange ..
upperrange LOOP
Statements;
END LOOP;
```

Steps

The following is the sequence in which FOR will take the steps.

1. Counter is set to lowerrange.
2. If counter is less than or equal to upperrange then statements are executed otherwise loop is terminated.
3. Counter is incremented by one and only one. It is not possible to increment counter by more than one.
4. Repeats step2.

Example:

```
SQL> DECLARE
  2     num1 NUMBER(4);
  3  BEGIN
  4     num1 :=19;
  5     FOR num IN 1..10 LOOP
  6      DBMS_OUTPUT.PUT_LINE('Table of 19 :' || num1);
  7     num1:= num1 + 19;
```

```
  8      END LOOP;
  9      END;
 10    /
Table of 19 :19
Table of 19 :38
Table of 19 :57
Table of 19 :76
Table of 19 :95
Table of 19 :114
Table of 19 :133
Table of 19 :152
Table of 19 :171
Table of 19 :190

PL/SQL procedure successfully completed.
```

Example :

```
SQL>  CREATE  TABLE  temp(num  NUMERIC,  str
VARCHAR(20));
Table created.

SQL> BEGIN
  2     FOR i IN 1..10 LOOP
  3        IF MOD(i,2) = 0 THEN
  4       INSERT INTO temp VALUES(i,'number is even');
  5        ELSE
  6      INSERT INTO temp VALUES(i,'number is odd');
  7        END IF;
  8      END LOOP;
  9    END;
 10  /

PL/SQL procedure successfully completed.
SQL> SELECT * FROM temp;

       NUM       STR
      --------     --------------------
         1          number is odd
         2          number is even
         3          number is odd
         4          number is even
         5          number is odd
         6          number is even
         7          number is odd
         8          number is even
         9          number is odd
        10          number is even

10 rows selected.
```

If REVERSE option is used the following steps will take place:

1. Counter is set to upper range.
2. If counter is greater than or equal to lower range then statements are executed otherwise loop is terminated.
3. Counter is decremented by one.
4. Go to step 2.

Example

```
SQL>  DECLARE
  2     n NUMBER := 10;
  3     BEGIN
  4     DBMS_OUTPUT.PUT_LINE('RESULT');
  5     FOR i IN REVERSE 1..n LOOP
  6     DBMS_OUTPUT.PUT_LINE(i);
  7     END LOOP;
  8     END;
  9  /
RESULT
10
9
8
7
6
5
4
3
2
1


PL/SQL procedure successfully completed.
```

9.9: CONTINUE STATEMENT

To exit the current iteration of a loop, PL/SQL provides the following statements:

- CONTINUE
- Continue-When(Condition)

A) CONTINUE Statement:

You can put CONTINUE statements anywhere inside a loop, but not outside a loop. To complete a PL/SQL block before it reaches its normal end, use the RETURN statement .

- Continue statement support in only Oracle 11g.

A CONTINUE statement cannot cross a subprogram or method boundary.
When a CONTINUE statement is encountered, the current iteration of the loop completes immediately and control passes to the next iteration of the loop.

Example :

```
SQL>BEGIN
  2     FOR a IN 1 .. 10 LOOP
  3     IF MOD(a,2) = 0 THEN
  4     CONTINUE;
  5     DBMS_OUTPUT.PUT_LINE(' Even Number := ' || a);
  6     END IF;
  7     DBMS_OUTPUT.PUT_LINE(' Odd Number := ' || a);
  8     END LOOP;
  9     END;
```

```
Odd Number := 1
Even Number := 2
Odd Number := 3
Even Number := 4
Odd Number := 5
Even Number := 6
Odd Number := 7
Even Number := 8
Odd Number := 9
Even Number := 10

PL/SQL procedure successfully completed.
```

B) CONTINUE-WHEN statement
Syntax
Continue-When(Condition)
Once the condition in the When clause is evaluated and if found true, the current iteration of the loop completes and control passes to the next iteration.

Example :

```
SQL> BEGIN
  2    FOR a IN 1 .. 10 LOOP
  3    CONTINUE WHEN MOD(a,2) = 0;
  4    DBMS_OUTPUT.PUT_LINE('Odd Num := ' ||a);
```

```
    5   END LOOP;
    6   END;
    7   /
Odd Num := 1
Odd Num := 3
Odd Num := 5
Odd Num := 7
Odd Num := 9
PL/SQL procedure successfully completed.
```

9.10 QUESTIONS

1. Explain the PLSQL control Structure with its types and Syntax.
2. What are the different Operators used in IF statement?
3. Explain IF-THEN-ELSE Statement with help of Example.
4. Write a short example to demonstrate use of Case Statement in PLSQL.
5. Explain Searched CASE statement in PLSQL with Help of Example.
6. Write Short note on EXIT and Continue Statement in PLSQL.
7. How to Label PLSQL loop?
8. Write a short example to demonstrate use of WHILE loop in PLSQL.
9. Write a short example to demonstrate use of FOR loop in PLSQL.

Practice Questions:

10. Write a PL/SQL block of code for reverse number(e.g. 123=321)
11. If there are no transaction taken place in the last 365 days then mark the account status as inactive and then record the account number, opening date and the type of account in new table.
12. Write a PL/SQL block of code for area of a Triangle two times with different values. Store the values in a table.
13. Write a PL/SQL block of code for area of a Circle three times with different values. Store the values in a table.
14. Write a PL/SQL block of code that will accept an ID number of a student check if student's presence is less than 75% then declared not eligible for exam. (Create Student table with proper fields.)

9.11 FURTHER READING

- Murach's Oracle SQL and PLSQL by Joel Murach, Murach and Associates.
- Oracle Database 11g PL/SQL Programming Workbook, ISBN: 9780070702264,

  By: Michael McLaughlin,John Harper, TATAMCGRAW-HILL
- Oracle PL/SQL Programming, Fifth Edition By Steven Feuerstein, Bill Pribyl
- Oracle 11g: SQL Reference Oracle press
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, TATAMCGRAW-HILL
- Oracle database 11g: hands on SQL/PL SQL by Satish Asnani (PHI) EEE edition

10

COMPOSITE DATA TYPE & CURSORS

Unit Structure

10.1 OBJECTIVES

After completing this chapter, we will be able to:

- Learn and understand the Composite Data types.
- Understand the structure and working of Collections.
- Implement Collection Methods
- Understand the PLSQL Records
- Understand the Cursor implementation
- Implement the Cursor in FOR loop

10.2 INTRODUCTION TO COMPOSITE DATA TYPES:

Composite datatypes can be compileled as the joint venture of the existing data type. Composite datatypes are the datatypes that are build with the combination of the other available datatypes of PL/SQL. We can use this as the collection of the datatypes used to represent the one unique structure that can be used in the PL/SQL block.

10.3 COLLECTION:

A collection is referred as a sequence of multiple elements. It is an ordered group of elements, all of the same type. It is a general concept that includes lists, arrays, and other datatypes used in classic programming algorithms. Each element is addressed by a unique subscript.

PL/SQL offers following collection types:

- **Associative arrays**: These are also known as **index-by tables**; that let us look up elements using arbitrary numbers and strings for subscript values. (They are similar to **hash tables** in other programming languages.)
- **Nested tables**: These hold a random number of elements. They use sequential numbers as subscripts. We can define equivalent SQL types, allowing nested tables to be stored in database tables and manipulated through SQL.
- **Varrays** (Variable-size arrays): These hold a fixed number of elements (although we can change the number of elements at runtime). They use sequential numbers as subscripts. We can define equivalent SQL types, allowing Varrays to be stored in database tables. They can be stored and retrieved through SQL, but with less flexibility than nested tables.

10.4: INDEX-BY TABLES

The first type of collection is known as index-by tables. These behave in the same way as arrays except that have no upper bounds, allowing them to constantly extend. As the name implies, the collection is indexed using BINARY_INTEGER values, which do not need to be consecutive. The collection is extended by assigning values to an element using an index value that does not currently exist.

```
SQL> DECLARE
  2    TYPE table_type IS TABLE OF NUMBER(10)
  3      INDEX BY BINARY_INTEGER;
  4
  5    v_tab  table_type;
  6    v_idx  NUMBER;
  7  BEGIN
  8    -- Initialise the collection.
  9    << load_loop >>
 10    FOR i IN 1 .. 5 LOOP
 11      v_tab(i) := i;
 12    END LOOP load_loop;
 13
 14    -- Delete the third item of the collection.
 15    v_tab.DELETE(3);
 16
 17    -- Traverse sparse collection
 18    v_idx := v_tab.FIRST;
 19    << display_loop >>
 20    WHILE v_idx IS NOT NULL LOOP
 21      DBMS_OUTPUT.PUT_LINE('The number ' || v_tab(v_idx));
 22      v_idx := v_tab.NEXT(v_idx);
 23    END LOOP display_loop;
 24  END;
 25  /

The number 1
The number 2
The number 4
The number 5

PL/SQL procedure successfully completed.
```

10.5: COLLECTION METHODS

The collection in PLSQL comes with huge support by various methods. A variety of methods exist for collections, but not all are relevant for every collection type. The methods are specific for specific collections.

A) EXISTS:- If n^{th} element is present in a collection, it returns true.

Example:

```
IF sub.EXISTS(a) THEN sub(a) := new_sub;
END IF;
```

B) COUNT:- This method help to count the number of elements in a collection.

Example:
IF subs.COUNT = 24 THEN ...

C) LIMIT:- This method is used in varrays, LIMIT returns the bound value or maximum number of elements the varray can contain.

Example:
IF subs.LIMIT = 24 THEN ...

D) FIRST:- This method returns the first member of the collection.

Example:
IF sub.FIRST = sub.LAST THEN ...

E) LAST:- This method returns the last member of the collection.

Example:
IF sub.FIRST = sub.LAST THEN ...

F) PRIOR:- This method returns the preceding index number of n^{th} element.

Example:
n := subs.PRIOR(subs.FIRST);

G) PRIOR:- This method returns the next or succeeding index number of n^{th} element.

Example:
a := subs.NEXT(a);

H) EXTEND:- This method is used to increase the size of nested table or varray. It either appends one null element to a collection, n null elements or n copies of ith element of a collection.

Example :
subs.EXTEND(5,1); // It appends 5 copies of element 1.

I) TRIM:- this method is used to decrease the size of collection by removing one element from end of collection or n elements.

Example:
subs.TRIM(3);

J) DELETE:- This method deletes the collection elements. It either deletes all elements, n^{th} element from an array, all elements in the range m, n.

Example:
subs.DELETE(2);

Example on collection methods:

```
 1  DECLARE
 2    TYPE my_office IS TABLE OF VARCHAR2 (100);
 3
 4    office_staff      names_t := names_t ( );
 5    clerk             names_t := names_t ( );
 6    manager           names_t := names_t ( );
 7  BEGIN
 8    office_staff.EXTEND (4);
 9    office_staff (1) := 'Takshak';
10     office_staff (2) := 'Viraj';
11     office_staff (3) := 'Sonal';
12     office_staff (4) := 'pooja';
13
14     clerk.EXTEND;
15     clerk(clerk.LAST) := 'Sonal';
16     clerk.EXTEND;
17     clerk(clerk.LAST) := 'Viraj';
18
19     manager: = office_staff MULTISET EXCEPT clerk;
20
21     FOR l_row IN 1 .. manager.COUNT
22     LOOP
23       DBMS_OUTPUT.put_line (manager (l_row));
24     END LOOP;
25  END;
```

10.6 HOW TO USE INDEX BY TABLE OF RECORDS?

Example : Declare an index-by table variable to hold the employee records in cursor

```
SQL> CREATE TABLE EMP (EMPNO NUMBER(4) NOT NULL,
  2                    ENAME VARCHAR2(10),
  3                    JOB VARCHAR2(9),
  4                    MGR NUMBER(4),
  5                    HIREDATE DATE,
  6                    SAL NUMBER(7, 2),
  7                    COMM NUMBER(7, 2),
  8                    DEPTNO NUMBER(2));
Table created.

SQL> INSERT INTO EMP VALUES (736, 'Sonali', 'CLERK',    790,
TO_DATE('17-DEC-2000', 'DD-MON-YYYY'), 800, NULL, 20);
1 row created.
```

```
SQL> INSERT INTO EMP VALUES (749, 'Tushar', 'SALESMAN', 769,
TO_DATE('20-FEB-2001', 'DD-MON-YYYY'), 1600, 300, 30);
1 row created.

SQL> INSERT INTO EMP VALUES (752, 'Priya',  'SALESMAN', 769,
TO_DATE('22-FEB-2000', 'DD-MON-YYYY'), 1250, 500, 30);
1 row created.

SQL> INSERT INTO EMP VALUES (756, 'Tanu', 'MANAGER',  783,
TO_DATE('2-APR-2001',  'DD-MON-YYYY'), 2975, NULL, 20);
1 row created.

SQL> INSERT INTO EMP VALUES (765, 'Sara', 'SALESMAN',
769,TO_DATE('28-SEP-2001', 'DD-MON-YYYY'), 1250, 1400, 30);
1 row created.

SQL> INSERT INTO EMP VALUES (769, 'Sana', 'MANAGER',
783,TO_DATE('1-MAY-2001', 'DD-MON-YYYY'), 2850, NULL, 30);
1 row created.

SQL> INSERT INTO EMP VALUES (778, 'Tejas', 'MANAGER',
783,TO_DATE('9-JUN-2001','DD-MON-YYYY'), 2450, NULL, 10);
1 row created.

SQL> INSERT INTO EMP VALUES (778, 'Akhi', 'ANALYST',
756,TO_DATE('09-DEC-2002','DD-MON-YYYY'), 3000, NULL, 20);
1 row created.

SQL> INSERT INTO EMP VALUES (783, 'Amol', 'PRESIDENT',
NULL,TO_DATE('17-NOV-2001', 'DD-MON-YYYY'), 5000, NULL, 10);
1 row created.

SQL> INSERT INTO EMP VALUES (784, 'Sandip', 'SALESMAN',
769,TO_DATE('8-SEP-2001', 'DD-MON-YYYY'), 1500, 0, 30);
1 row created.

SQL> DECLARE
  2          CURSOR all_emps IS
  3            SELECT *
  4            FROM emp
  5            ORDER BY ename;
  6
  7        TYPE emp_table IS TABLE OF emp%ROWTYPE
  8                    INDEX BY BINARY_INTEGER;
  9       emps emp_table;
 10        emps_max BINARY_INTEGER;
 11     BEGIN
 12        emps_max := 0;
```

```
13         FOR emp IN all_emps LOOP
14           emps_max := emps_max + 1;
15           emps(emps_max).empno    := emp.empno;
16           emps(emps_max).ename    := emp.ename;
17           emps(emps_max).JOB      := emp.JOB;
18            emps(emps_max).HIREDATE := emp.HIREDATE;
19            emps(emps_max).DEPTNO   := emp.DEPTNO;
20       END LOOP;
21     END;
22  /
```

PL/SQL procedure successfully completed.

10.7 INTRODUCTION TO PL/SQL RECORD:

The PLSQL **record** is a group of related data items stored in **fields**, each with its own name and datatype. The record is special type of PLSQL variable that can hold a table row, or some columns from a table row. The fields of record correspond to table columns.

The %ROWTYPE attribute lets us declare a record that represents a row in a database table, without listing all the columns. Our code keeps working even after columns are added to the table. If we want to represent a subset of columns in a table, or columns from different tables, we can define a view or declare a cursor to select the right columns and do any necessary joins, and then apply %ROWTYPE to the view or cursor.

The important thing while using %TYPE or %ROWTYPE is the table name and the column name must already exist in database.

A) %TYPE and %ROWTYPE:

The %TYPE and %ROWTYPE attributes are used to define variables in PL/SQL , as it is defined within the database. If the datatype or precision of a column changes, the program automatically picks up the new definition from the database without having to make any code changes.

The %TYPE and %ROWTYPE constructs the variable, provide data independence, reduces maintenance costs, and allows programs to adapt as the database changes to meet new business needs.

- **%TYPE**

The %TYPE attribute is used to declare a *field* with the same type as that of a specified table's column. % TYPE provides the data type of a variable or a database column to that variable. This is very useful because we do not have to declare the parameter data

type and also, when the column data type changes in the table, we do not come to procedure and change the data type.

Syntax :

record_name record_type_name;

Example :

```
SQL> DECLARE
  2    v_name  student.name%TYPE;
  3  BEGIN
  4    SELECT name INTO v_name FROM student WHERE
ROWNUM = 1;
  5    DBMS_OUTPUT.PUT_LINE('Name = ' || v_name);
  6  END;
  7  /

Name = abc
PL/SQL procedure successfully completed.
```

- **%ROWTYPE**

The %ROWTYPE attribute is used to declare a *record* with the same types as found in the specified database table, view or cursor. % ROWTYPE provides the record type that represents a entire row of a table or view or columns selected in the cursor. We normally use %ROWTYPE to retrieve the record which contains all of the columns from a specified database table.

Syntax :

record_name table_name%ROWTYPE;

Example :

```
SQL> DECLARE
  2   v_stud student%ROWTYPE;
  3  BEGIN
  4     v_stud.rollno := 6;
  5     v_stud.name := 'xyz';
  6     v_stud.mark1 := 35;
  7     v_stud.mark2 := 56;
  8     v_stud.mark3 := 67;
  9    DBMS_OUTPUT.PUT_LINE('rollno : '||v_stud.rollno);
 10    DBMS_OUTPUT.PUT_LINE('name : '||v_stud.name);
 11    DBMS_OUTPUT.PUT_LINE('mark1 : '||v_stud.mark1);
 12    DBMS_OUTPUT.PUT_LINE( 'mark2 : '||v_stud.mark2);
 13    DBMS_OUTPUT.PUT_LINE( 'mark3 : '||v_stud.mark3);
 14  END;
 15  /
rollno : 6
name : xyz
mark1 : 35
```

```
mark2 : 56
mark3 : 67

PL/SQL procedure successfully completed.
```

C) Using of %type :

```
create table student(
  2   rollno number(4) primary key, name varchar(4),
  3  mark1 number(4),
  4   mark2 number(4),
  5  mark3 number(4)
  6  );

SQL> declare
  2       v_rollno student.rollno%type := &rollno;
  3       v_name student.name%type := '&name';
  4       v_mark1 student.mark1%type := &mark1;
  5       v_mark2 student.mark2%type := &mark2;
  6       v_mark3 student.mark3%type := &mark3;
  7  begin
  8       insert into student(rollno , name,mark1 , mark2 , mark3)
  9       values(v_rollno, v_name, v_mark1 , v_mark2 , v_mark3);
 10        dbms_output.put_Line('Inserted Successfully');
 11  end;
 12  /

Enter value for rollno: 4
old   2:       v_rollno student.rollno%type := &rollno;
new   2:       v_rollno student.rollno%type := 4;
Enter value for name: Sana
old   3:       v_name student.name%type := '&name';
new   3:       v_name student.name%type := 'Sana';
Enter value for mark1: 24
old   4:       v_mark1 student.mark1%type := &mark1;
new   4:       v_mark1 student.mark1%type := 24;
Enter value for mark2: 25
old   5:       v_mark2 student.mark2%type := &mark2;
new   5:       v_mark2 student.mark2%type := 25;
Enter value for mark3: 45
old   6:       v_mark3 student.mark3%type := &mark3;
new   6:       v_mark3 student.mark3%type := 45;
Inserted Successfully
```

10.8 WRITING INSERT AND UPDATE WITH PL/SQL RECORDS :

PLSQL gives us flexibility to write INSERT, UPDATE, and DELETE statements directly in PL/SQL programs, without any special notation:

```
SQL> CREATE TABLE I1( a INTEGER, b INTEGER);
Table created.

SQL> INSERT INTO I1 VALUES(1, 3);
1 row created.

SQL> INSERT INTO I1 VALUES(2, 4);
1 row created.

SQL> DECLARE
  2      x I1.a%TYPE:=64;
  3      y I1.b%TYPE:=54;
  4  BEGIN
  5  INSERT INTO I1 VALUES(x,y);
  6  END;
  7  /
PL/SQL procedure successfully completed.

SQL> SELECT * FROM I1;
         A          B
---------- ----------
         1          3
         2          4
        64         54

SQL> DECLARE
  2      x I1.a%TYPE:=64;
  3      y I1.b%TYPE:=54;
  4  BEGIN
  5  UPDATE I1 SET a = x WHERE b < y;
  7  END;
  8  /
PL/SQL procedure successfully completed.

SQL> SELECT * FROM I1;
         A          B
---------- ----------
        64          3
        64          4
        64         54

SQL> DECLARE
  2      x I1.a%TYPE:=64;
  3      y I1.b%TYPE:=54;
  4  BEGIN
  5  DELETE FROM I1 WHERE a = x;
  6  END;
  7  /
```

PL/SQL procedure successfully completed.

SQL> SELECT * FROM I1;
no rows selected

10.9: CURSOR

A cursor in PL/SQL means a specific private SQL area where information for the specific statement is kept for processing. PL/SQL uses both **implicit** and **explicit** cursors. PL/SQL implicitly declares a cursor for all SQL data manipulation statements on a set of rows, including queries that return only one row. For queries that return more than one row, we can explicitly declare a cursor to process the rows individually.

In the simplest form, a **cursor** is defined as the pointer into a table in the database. This simplifies the task of finding proper values among the huge set of database tables.

In PL/SQL block , SELECT statement cannot return more than one row at a time. So Cursor use to some group of rows(more than one row) for implementing certain logic to all the records of group are show.

10.10: CLASSIFICATION OF CURSORS

Cursors can be classified as:

- **Implicit Cursor or Internal Cursor** – These are managed by Oracle itself or it is the Internal Process of oracle for itself.
- **Explicit Cursor or User-defined Cursor** – these are used to manage for User/Prgrammer or External Processing.

A) Implicit Cursor:

Oracle uses implicit cursors for its internal processing. Even if we execute a SELECT statement Oracle reserves a private SQL area in memory called cursor.

Implicit cursor variables

Cursor Attribute	Cursor Variable	Description
%ISOPEN	SQL%ISOPEN	The Oracle engine automatically opens the cursor If cursor open **return true** otherwise **return false.**
%FOUND	SQL%FOUND	If selected return one or more than one row INSERT, UPDATE, DELETE opration affect If affect **return true** otherwise **return false.**
%NOTFOUND	SQL%NOTFOUND	If selected return one or more than one row INSERT, UPDATE, DELETE opration not affect If not affect the row **return true** otherwise **return false.**
%ROWCOUNT	SQL%ROWCOUNT	It return the number of rows affected by an insert, update, delete or select statement.

```
SQL> DECLARE  var_rows number(5);
  2  BEGIN
  3    UPDATE student
  4    SET mark1 = mark1 + 10;
  5    IF SQL%NOTFOUND THEN
  6      dbms_output.put_line('None of the marks where updated');
  7    ELSIF SQL%FOUND THEN
  8      var_rows := SQL%ROWCOUNT;
  9      dbms_output.put_line('Marks for ' || var_rows || 'students are
updated');
 10    END IF;
 11  END;
 12  /
```

Marks for 6 students are updated
PL/SQL procedure successfully completed.

❖ **Drawbacks of Implicit Cursors**

When we use implicit cursor, if our query returns only a single row, we can still decide to use an explicit cursor. The implicit cursor has the following drawbacks:

- Implicit cursors are less efficient than an explicit cursor.
- Implicit cursors are more vulnerable to data errors.
- Implicit cursors gives us less programmatic control.

B) Explicit Cursor:

The Cursors which are declared by user are called Explicit Cursor. The user has to declare the cursor, open cursor to reserve the memory are and populate data, fetch the records from the active data set one at a time, apply logic and last close the cursor.

-
- How to use Explicit Cursor?

There are four steps for using an Explicit Cursor.

- We have to DECLARE the cursor in the declaration section.
- OPEN the cursor in the Execution Section.
- FETCH the data from cursor into PL/SQL variables or records in the Execution Section.
- CLOSE the cursor in the Execution Section before we end the PL/SQL Block.

Cursor Declaration :

CURSOR c_student IS SELECT RollNo,mark1,mark2,mark3 FROM student;

In above syntax we have created a cursor with name c_student which is associated with student table.
Once we declared the cursor, we can open it as:

OPEN c_student;

Then we can fetch rows from it as:

FETCH c_student INTO var_rollno,var_mark1,var_mark2,var_mark3 ;

After finishing the use of cursor we can close it as:

CLOSE c_student;

Explicit cursor variables

Cursor Attribute	Cursor Variable	Description
%ISOPEN	c%ISOPEN	Oracle engine automatically open the cursor If cursor open **return true** otherwise **return false.**
%FOUND	c%FOUND	If selected return one or more than one row INSERT, UPDATE, DELETE opration affect If affect **return true** otherwise **return false.**
%NOTFOUND	c%NOTFOUND	If selected return one or more than one row INSERT, UPDATE, DELETE opration not affect If not affect the row **return true** otherwise **return false.**
%ROWCOUNT	c%ROWCOUNT	Return the number of rows affected by an insert, update, delete or select statement.

```
SQL> create table student(
  2   rollno number(4) primary key, name varchar(4),
  3  mark1 number(4),
  4   mark2 number(4),
  5  mark3 number(4)
  6  );

Table created.

SQL> insert into student values(1,'Hitesh',56,78,89);
1 row created.

SQL> insert into student values(2,'Suresh',55,44,66);
1 row created.

SQL> insert into student values(3,'Kamal',56,78,89);
1 row created.

SQL>  insert into student values(4,'Varun',55,44,66);
1 row created.

SQL> insert into student values(5,'Anu',56,78,89);
1 row created.

SQL> insert into student values(6,'Dinu',55,44,66);
```

```
1 row created.
SQL> select * from student;
ROLLNO NAME   MARK1 MARK2 MARK3
       -- -------- ---- ---------- ---------- ----------
       1 Hitesh         56         78         89
       2 Suresh          55         44         66
       3 Kamal          56         78         89
       4 Varun          55         44         66
       5 Anu            56         78         89
       6 Dinu           55         44         66

6 rows selected.

SQL> create table student_performance
  2  (
  3  Rollno number(4),
  4  Total number(4),
  5  Average number(4),
  6  Grade varchar(15)
  7  );

Table created.

SQL> Declare
 var_total number(4);
 var_average number(4);
 var_grade varchar(15);
 var_rollno number(4);
 var_mark1 number(4);
 var_mark2 number(4);
 var_mark3 number(4);
 cursor c_student is
 select RollNo,mark1,mark2,mark3 from student;
 begin
 open c_student;
 loop
 fetch c_student into var_rollno,var_mark1,var_mark2,var_mark3 ;
 exit when c_student%notfound;
 var_total:=var_mark1+var_mark2+var_mark3;
 var_average:=var_total/3;
 if var_average<35 then
 var_grade:='fail';
 else
```

```
 var_grade:='pass';
 end if;
 insert into student_performance
values(var_rollno,var_total,var_average,var_grade);
 end loop;
 close c_student;
 end;
 /
```

PL/SQL procedure successfully completed.

SQL> select * from student_performance;

ROLLNO	TOTAL	AVERAGE	GRADE
1	223	74	pass
2	165	55	pass
3	223	74	pass
4	165	55	pass
5	223	74	pass
6	165	55	pass

6 rows selected.

10.11 USING CURSOR IN FOR LOOP :

When we use FOR LOOP, we need not declare a record or variables to store the cursor values, need not open, fetch and close the cursor. These functions are accomplished by the FOR LOOP automatically.

General Syntax for using FOR LOOP:

```
FOR record_name IN cusror_name
LOOP
   process the row...
END LOOP;
```

Example

```
SQL> declare
  2   cursor s_stud (rno NUMBER) is
  3        select * from student where rollno = rno;
  4   begin
  5       for r_stud in s_stud(5) loop
  6           update student
  7           set mark2=mark2+5
  8           where rollno = r_stud.rollno;
```

```
 9          DBMS_OUTPUT.put_line('Student RollNo: '||
       r_stud.rollno||' Student Name: '||
            r_stud.name);
10        end loop;
11    end;
12    /
Student RollNo: 5 Student Name:iop
```

PL/SQL procedure successfully completed.

10.12 THE %NOTFOUND AND %ROWCOUNT ATTRIBUTES

-
- **A) %NOTFOUND**

The %NOTFOUND attribute works opposite of %FOUND. It returns TRUE if the cursor is unable to fetch another row because the last row was fetched. If the cursor is unable to return a row because of an error, the appropriate exception is raised. If the cursor has not yet been opened, a reference to the %NOTFOUND attribute raises the INVALID_CURSOR exception. We can evaluate the %NOTFOUND attribute of any open cursor, because we reference the cursor by name.

- **B) %ROWCOUNT**

The %ROWCOUNT is a cursor attribute. The value of %ROWCOUNT is set after the FETCH command is executed or an INSERT, UPDATE or DELETE implicit cursor is used. The %ROWCOUNT attribute (for EXPLICIT and IMPLICIT cursors) does not return the total number of rows for a query prior to the first fetch. It does return:

- The number of rows fetched “so far” following a fetch.
- The number of rows affected by a INSERT, UPDATE, and DELETE.

The attribute can be used in procedural statements but not in SQL statements.

Statement	%ROWCOUNT value
FETCH	Number of rows returned by the fetched cursor, incremented 1 time for each successful fetch.
SELECT INTO	1, even if TOO_MANY_ROWS is raised
UPDATE	Number of rows effected
DELETE	Number of rows effected
INSERT	Number of rows effected

Example :

```
SQL> DECLARE
  2  TYPE item_record IS RECORD
  3  ( id NUMBER, title VARCHAR2(60));
  4  item ITEM_RECORD;
  5  CURSOR c IS
  6  SELECT rollno, name
  7  FROM student
  8  WHERE rollno = 1;
  9  BEGIN
 10  OPEN c;
 11  LOOP
 12  FETCH c INTO item;
 13  IF c%NOTFOUND THEN
 14  IF c%ROWCOUNT = 0 THEN
 15  dbms_output.put_line('No Data Found');
 16  END IF;
 17  EXIT;
 18  ELSE
 19  dbms_output.put_line('Name of student :'||item.title);
 20  END IF;
 21  END LOOP;
 22  END;
 23  /
Name of student :abc

PL/SQL procedure successfully completed.
```

10.13 FOR UPDATE CLAUSE AND WHERE CURRENT CLAUSE

A) FOR UPDATE clause

The FOR UPDATE clause is an optional part of a SELECT statement. Cursors are read-only by default. The FOR UPDATE clause specifies that the cursor should be updatable, and enforces a check during compilation that the SELECT statement meets the requirements for an updatable *cursor*. For more information about updatability, see Requirements for updatable cursors and updatable ResultSets.

Syntax

FOR
{
READ ONLY | FETCH ONLY |
UPDATE [OF *Simple-column-Name* [, *Simple-column-Name*]*]
}

Simple-column-Name refers to the names visible for the table specified in the FROM clause of the underlying query.

Note: The use of the FOR UPDATE clause is not mandatory to obtain an updatable JDBC ResultSet. As long as the statement used to generate the JDBC ResultSet meets the requirements for updatable cursor, it is sufficient for the JDBC Statement that generates the JDBC ResultSet to have concurrency mode ResultSet.CONCUR_UPDATABLE for the ResultSet to be updatable.

The optimizer is able to use an index even if the column in the index is being updated.

Example :

```
SQL> create table ftbl (a number, b varchar2(10));
Table created.

SQL> insert into ftbl values (5,'five');
1 row created.

SQL> insert into ftbl values (6,'six');
1 row created.

SQL> insert into ftbl values (7,'seven');
1 row created.

SQL> insert into ftbl values (8,'eight');
1 row created.

SQL> insert into ftbl values (9,'nine');
1 row created.

SQL> create or replace procedure pincr as
  2    cursor c_ftbl is
  3      select a,b from f where length(b) = 5 for update;
  4      v_a ftbl.a%type;
  5      v_b ftbl.b%type;
  6  begin
  7    open c_ftbl;
```

```
 8   loop
 9     fetch c_ftbl into v_a, v_b;
10     exit when c_ftbl%notfound;
11     update ftbl set a=v_a+5 where current of c_ftbl;
12   end loop;
13  close c_ftbl;
14 end;
15 /
```

Procedure created.
SQL> exec pincr;
PL/SQL procedure successfully completed.

B) WHERE CURRENT Clause:

The WHERE CURRENT OF clause is a clause in some UPDATE and DELETE statements. It allows we to perform positioned updates and deletes on updatable cursors. For more information about updatable cursors, see SELECT statement

Syntax

WHERE CURRENT OF *cursor-Name*

Example

```
SQL> create table testc (  n number(3),  c varchar(50));
Table created.
SQL> insert into testc values (1,  'one');
1 row created.

SQL> insert into testc values (10, 'ten');
1 row created.

SQL> insert into testc values (15, 'one five');
1 row created.

SQL> insert into testc values (99, 'nine nine');
1 row created.

SQL> insert into testc values (42, 'four two');
1 row created.

SQL> declare
  2   cursor cur_test is
  3     select n, c from testc for update;
  4   n number(3);
  5   c varchar(50);
  6 begin
  7   open cur_test;
  8   loop
```

```
 9      fetch cur_test into n, c;
10      exit when cur_test%notfound;
11      if n>12 then
12        update testc set n=n*2, c=upper(c) where current of
cur_test;
13      end if;
14    end loop;
15  end;
16  /
```

PL/SQL procedure successfully completed.

10.14 QUESTIONS

1 Write a short note on Collections in PLSQL.

2 Explain Index By Tables with help of Example.

3 Write and explain Collection methods in PLSQL.

4 How to use INDEX BY Table of Records?

5 Write a short note on PL/SQL Records and its types.

6 How to Insert and Update with PL/SQL Records?

7 What are cursors in PLSQL? Write short note on Classification of Cursors.

8 Explain cursor with FOR loop.

9 Write a short note on %NOTFOUND and %ROWCOUNT Attributes.

10 Explain For UPDATE Clause and WHERE CURRENT Clause.

11 How to use WHERE CURRENT Clause? Explain with help of Example.

Practice Questions:

12 Create Table the employee with 6 fields and 10 records. Apply the various collection methods of PLSQL on the records of the table.

13 Update the above table records using the PLSQL record.

14 Use the %NOTFOUND and %ROWCOUNT attributes on the same table.

15 The bank manager of Ghansoli branch decides to activate all those accounts, which were previously marked as inactive for performing no transactions in last 365 days. Write a PL/SQL block to update the status of accounts. Display an appropriate message based on the number of rows affected by the update fired (use SQL%ROWCOUNT).

10.15 FURTHER READING

- Murach's Oracle SQL and PLSQL by Joel Murach, Murach and Associates.
- Oracle Database 11g PL/SQL Programming Workbook, ISBN: 9780070702264,

  By: Michael McLaughlin,John Harper, TATAMCGRAW-HILL
- Oracle PL/SQL Programming, Fifth Edition By Steven Feuerstein, Bill Pribyl
- Oracle 11g: SQL Reference Oracle press
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, TATAMCGRAW-HILL
- Oracle database 11g: hands on SQL/PL SQL by Satish Asnani (PHI) EEE edition

11

EXCEPTIONS HANDLING

Unit Structure

11.1 OBJECTIVE

After completing this chapter, you will be able to:

- ❖ Understand the Errors and Handling Errors in PLSQL
- ❖ Understand the Structure of Exception Handling
- ❖ Implement the different types of Exception Handling Code
- ❖ Understand the PLSQL Functions like SQLCODE and SQLEERM
- ❖ Using the RAISE_APPLICATION_ERROR ( )

11.2 INTRODUCTION TO EXCEPTION

- In simple words the Exception means the problem which may cause to interrupt the program execution.

- The Exception can be defined as an error situation, which arises during program execution. When an error occurs exception is raised, normal execution is stopped and control transfers to exception-handling part (If it is written in the code).
- Exceptions are defined as the condition that can cause the application into inconsistent state.
- A warning or error condition is called an **Exception**. Exceptions can be internally defined or user defined. Some common internal exceptions have predefined names, such as ZERO_DIVIDE and STORAGE_ERROR.
- We can define exceptions of our own in the declarative part of any PL/SQL block, subprogram, or package. For example, we might define an exception named low_balance to flag overdrawn bank accounts. Unlike internal exceptions, user-defined exceptions *must* be given names.
- To handle raised exceptions, we have to write separate block called *Exception Handlers*. After an exception handler is start its work, the current block stops executing and the enclosing block resumes with the next statement. If there is no enclosing block, control returns to the current environment.
- When an error occurs, an exception is raised. That means normal execution stops and control transfers to the exception handling part of our PL/SQL block or subprogram. Internal exceptions are raised automatically by the run-time system. User-defined exceptions must be raised explicitly by *RAISE* statements, which can also raise predefined exceptions.
- Using exceptions for error handling has several advantages. Without exception handling, every time we issue a command, we must check for execution errors.

```
BEGIN
  SELECT ...
    -- -- check for 'name not found' error
  SELECT ...
    -- -- check for 'name not found' error
  SELECT ...
    -- -- check for 'name not found' error
```

- The Error handling is not clearly different from normal processing; nor is it stout. If we neglect to code a check, the error goes undetected and is likely to cause other apparently unrelated errors.
- Using exceptions, we can handle errors with very ease without the need to code multiple checks, as follows:

```
BEGIN
  SELECT ...
  SELECT ...
  SELECT ...
  ...
EXCEPTION
  WHEN NAME_NOT_FOUND THEN  -- catches all ' name not
found ' errors
```

- The Exceptions brings readability in code by separate Error-Handling blocks.The Exceptions also improve reliability. We need not worry about checking for an error at every point it might occur. We just have to add an Exception Handler to our PL/SQL block. If the exception is ever raised in that block (or any sub-block), we can make sure it will be handled.

- **PL/SQL Exception message consists of three parts.**
  1) Type of Exception
  2) An Error Code
  3) A message
  By Handling the exceptions we can ensure that the PL/SQL block does not exit unexpectedly.

11.3 CODING STRUCTURE OF EXCEPTION HANDLING.

The coding of Exception handling section in PLSQL is quite simple and easy to understand. The General Syntax for coding the exception section

```
DECLARE
  Declaration section // Here we can declare any PLSQL
variables etc
BEGIN
  Exception section // This is the section which we have to
monitor for errors
EXCEPTION
    WHEN First_Exception THEN
      -The statements to handle the errors
    WHEN Second_Exception THEN
      - The statements to handle the errors
    WHEN Others THEN
      - The statements to handle the errors
END;
/
```

We can use normal, general PL/SQL statements in the Exception block. When an Exception is raised, Oracle searches for an appropriate exception handler in the exception section. For example in the above example, if the error raised is 'First_Exception ', then the error is handled according to the statements under it. Since, We cannot assume all the errors at once while coding and it is not possible to determine all the possible runtime errors during testing our code, the 'WHEN Others' exception is used to manage the exceptions that are not explicitly handled. Only one exception can be raised in a Block and the control does not return to the execution section after the error is handled.

If there exists, nested PL/SQL blocks as given in the following:

```
DELCARE
  Declaration section // Here we can declare any PLSQL variables etc
BEGIN
   DECLARE
    Declaration section // Here we can declare any PLSQL variables etc
   BEGIN
    Execution section // This is the section which we have to monitor for outer errors
   EXCEPTION
    Exception section // Here we have to write the PLSQL inner Exception Handling code
   END;
EXCEPTION
  Exception section  // Here we have to write the PLSQL inner Exception Handling code
END;
/
```

In the above case, if the exception is raised in the inner block it should be handled in the exception block of the inner PL/SQL block otherwise the control moves to the Exception block of the next upper PL/SQL block. If none of the blocks handle the exception the program ends abruptly with an error.

- Example of PLSQL Exception Handling block :

```
EXCEPTION
WHEN NO_DATA_FOUND THEN
v_msg := 'No company for id ' || TO_CHAR (v_id);
v_err := SQLCODE;
v_prog := 'fixdebt';
```

```
INSERT INTO errlog VALUES (v_err, v_msg, v_prog,
SYSDATE, USER);
WHEN OTHERS THEN
v_err := SQLCODE;
v_msg := SQLERRM;
v_prog := 'fixdebt';
INSERT INTO errlog VALUES (v_err, v_msg, v_prog,
SYSDATE, USER);
RAISE;
```

11.4 RULES FOR PL/SQL EXCEPTIONS

- The Exceptions declared in one block are considered local to that block and global to all its sub-blocks. Because a block can reference only local or global exceptions, enclosing blocks cannot reference exceptions declared in a sub-block.
- We should not declare an exception more than once in the similar block. We can declare the same exception in two different separate blocks.
- If we re-declare a global exception in a sub-block, the local declaration prevails. So, the sub-block cannot reference the global exception unless it was declared in a labeled block, in which case the following syntax is valid:

Block_name_label.name_of_exception

The following example illustrates the scope rules:

```
DECLARE
  balance EXCEPTION;
  customer_id NUMBER;
BEGIN
  DECLARE   // --- here the inner block begins
    balance EXCEPTION;  // -- this declaration prevails
    customer_id NUMBER;
  BEGIN
    ...
    IF ... THEN
      RAISE balance;  // -- this is not handled because raised in
outer block
    END IF;
  END;  //  -- Here inner block ends
EXCEPTION
  WHEN balance THEN  // -- handle raised exception
    ...
END;
/
```

The enclosing block does not handle the raised exception because the declaration of *balance* in the sub-block prevails.

Though they share the same name, the two *balance* exceptions are different, just as the two customer_id variables share the same name but they are different variables. Therefore, the RAISE statement and the WHEN clause refer to different exceptions. To have the enclosing block handle the raised exception, we must remove its declaration from the sub-block or define an OTHERS handler.

11.5 PRAGMA EXCEPTION_INIT( ) :

Associating a PL/SQL Exception with a Number: **Pragma EXCEPTION_INIT** to handle error conditions (typically ORA-messages) that have no predefined name, we must use the OTHERS handler or the pragma EXCEPTION_INIT. A **pragma** is a compiler directive that is processed at compile time and not at run time. In PL/SQL, the pragma EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number. This allows to refer to any internal exception by name and to write a specific handler for it. When we see an **error stack**, or sequence of error messages, the one on top is the one that we can trap and handle. We code the pragma EXCEPTION_INIT in the declarative part of a PL/SQL block, subprogram or package using the syntax as …..

PRAGMA EXCEPTION_INIT(Name of Exception, -Oracle_error_number);

Where 'Name of Exception' is the name of a previously declared exception and the number is a negative value corresponding to an ORA- error number. The pragma must appear somewhere after the exception declaration in the same declarative section, as shown in the following example:

```
DECLARE
  problem_found EXCEPTION;
  PRAGMA EXCEPTION_INIT(problem_found, -57);
BEGIN
  ... // -- here is  Some code that causes an ORA-00057 error
EXCEPTION
  WHEN problem_found THEN
    //  -- here we have to write the code to handle the error
END;
/
```

11.6 CLASSIFICATION OF PLSQL EXCEPTION.

There are 3 types of Exceptions.
a) The Named System Exceptions
b) The Unnamed System Exceptions
c) The Named Programmer-defined Exceptions

A) The Named system exception

The Named system exceptions are exceptions that have been already given names by PL/SQL. They are named in the STANDARD package in PL/SQL and do not need to be defined by the programmer.

"Named system exceptions are not declared explicitly, that are raised implicitly when a predefined Oracle error occurs, also that are caught by referencing the standard name within an exception-handling routine."

Oracle has a standard set of exceptions already named as follows:

Exception Name	Oracle Error	Explanation
DUP_VAL_ON_INDEX	ORA-00001	We tried to execute an INSERT or UPDATE statement that has created a duplicate value in a field restricted by a unique index.
TIMEOUT_ON_RESOURCE	ORA-00051	We are waiting for a resource and we timed out.
TRANSACTION_BACKED_OUT	ORA-00061	The remote portion of a transaction has rolled back.
INVALID_CURSOR	ORA-01001	We tried to reference a cursor that does not yet exist. This may have happened because we've executed a FETCH cursor or CLOSE cursor before OPENing the cursor.
NOT_LOGGED_ON	ORA-01012	We tried to execute a call to Oracle before logging in.
LOGIN_DENIED	ORA-01017	We tried to log into Oracle with an invalid username/password combination.

NO_DATA_FOUND	ORA-01403	We tried one of the following: We executed a SELECT INTO statement and no rows are returned. We referenced an uninitialized row in a table. We read past the end of file with the UTL_FILE package.
TOO_MANY_ROWS	ORA-01422	We tried to execute a SELECT INTO statement and more than one row was returned.
ZERO_DIVIDE	ORA-01476	We tried to divide a number by zero.
INVALID_NUMBER	ORA-01722	We tried to execute an SQL statement that tried to convert a string to a number, but it was unsuccessful.
STORAGE_ERROR	ORA-06500	We ran out of memory or memory was corrupted.
PROGRAM_ERROR	ORA-06501	This is a generic "Contact Oracle support" message because an internal problem was encountered.
VALUE_ERROR	ORA-06502	We tried to perform an operation and there was a error on a conversion, truncation, or invalid constraining of numeric or character data.
CURSOR_ALREADY_OPEN	ORA-06511	We tried to open a cursor that is already open.

The syntax for the named system exception in a procedure is:

```
CREATE [OR REPLACE] PROCEDURE
name_of_procedure
   [ (param1 [,param2]) ]
IS
   [declaration_section]
BEGIN
   executable_section // The code of this section will be
monitored for errors
```

```
EXCEPTION
   WHEN exception_name1 THEN
      [statements]    // Exception handling code
   WHEN exception_name2 THEN
      [statements]  // Exception handling code
   WHEN exception_name_n THEN
      [statements]    // Exception handling code
   WHEN OTHERS THEN
      [statements] // Exception handling code
END [procedure_name];
/
```

The syntax for the Named System exception in a function is:

```
CREATE [OR REPLACE] FUNCTION function_name
   [ (param1 [,param2]) ]
   RETURN return_datatype
IS | AS
   [declaration_section]
BEGIN
   executable_section // The code of this section will be
monitored for errors
EXCEPTION
   WHEN exception_name1 THEN
      [statements] // Exception handling code
   WHEN exception_name2 THEN
      [statements]    // Exception handling code
   WHEN exception_name_n THEN
      [statements]   // Exception handling code
   WHEN OTHERS THEN
      [statements]   // Exception handling code
        END [function_name];
        /
```

For Example: Suppose a NO_DATA_FOUND exception is raised in a procedure, then we can write a code to handle the exception as shown below.

```
BEGIN
  --Execution section    // The code of this section will
be monitored for errors
EXCEPTION
WHEN NO_DATA_FOUND THEN
```

```
 dbms_output.put_line ('Sorry the given statement
dose not return any row …');
 END;
/
```

Here is an example of a procedure that uses a Named System Exception:

Example:

```
CREATE OR REPLACE PROCEDURE new_customer  (cust_id IN
NUMBER, cust_name IN VARCHAR2)
IS
BEGIN
   INSERT INTO customer (cid,csname ) VALUES ( cust_id,
cust_name );
EXCEPTION
   WHEN DUP_VAL_ON_INDEX THEN
      raise_application_error (-20001,'We have tried to insert a
duplicate customer' id);
   WHEN OTHERS THEN
      raise_application_error (-20002,'An error has occurred
inserting a customer.');
END;
/
```

In this example, we are trying to trap the Named System Exception called **DUP_VAL_ON_INDEX**.

Example: NO_DATA_FOUND EXCEPTION

```
SQL> DECLARE
  2    uid   all_users.username%TYPE := 10;
  3    uname  all_users.username%TYPE;
  4  BEGIN
  5    SELECT username
  6    INTO   uname
  7    FROM   all_users
  8    WHERE  user_id = uid;
  9
 10    DBMS_OUTPUT.put_line('uname=' || uname);
 11  EXCEPTION
 12    WHEN NO_DATA_FOUND THEN
 13      DBMS_OUTPUT.put_line('No users have a user_id=' || uid);
 14  END;
 15  /
```

```
No users have a user_id=10
PL/SQL procedure successfully completed.
```

Example: ZERO_DIVIDE EXCEPTION

```
SQL> declare
  2  n number;
  3  begin
  4  n:=10/0;
  5  exception
  6  when ZERO_DIVIDE then
  7  dbms_output.put_line('zero divide error');
  8  end;
  9  /

zero divide error
PL/SQL procedure successfully completed.
```

B) The Unnamed System Exceptions

In simple terms we can refer these exceptions as exception without proper, standard system names. Those system exception for which oracle does not provide a name is known as unnamed system exception. These exceptions do not occur frequently. These Exceptions have a code and an associated message.

There are two ways to handle unnamed system exceptions:

- Using the WHEN OTHERS exception handler, or
- Associating the exception code to a name and using it as a named exception.

We can assign a name to unnamed system exceptions using a Pragma called EXCEPTION_INIT. EXCEPTION_INIT will associate a predefined Oracle error number to a programmer_defined exception name.

We have to follow some steps to use unnamed system exceptions are also we have to keep some points in our mind while using them as…

• They are raised implicitly.
• If they are not handled in WHEN OTHERS they must be handled explicitly.
• To handle the exception explicitly, they must be declared using

Pragma EXCEPTION_INIT as given above and handled referencing the user-defined exception name in the exception section.

The general syntax to declare unnamed system exception using EXCEPTION_INIT is as follows.

Syntax:

```
DECLARE
  exception_name EXCEPTION;
  PRAGMA
  EXCEPTION_INIT (exception_name, Err_code);
BEGIN
Execution section
EXCEPTION
 WHEN exception_name THEN
    handle the exception
END;
/
```

Consider the customer and the order table from sql joins. Here cid is a primary key in customer table and a foreign key in order table. If we try to delete cid from the product table while it has child records in oid table an exception will be thrown with oracle code number -1517. We can provide a name to this exception and handle it in the exception section as given below.

```
DECLARE
 Child_record_exception EXCEPTION;
 PRAGMA
  EXCEPTION_INIT (Child_record_exception, -1517);
BEGIN
 Delete FROM customer where cid= 345;
EXCEPTION
  WHEN Child_record_exception
  THEN dbms_output.put_line('Child records are present for
this product_id.');
END;
/
```

C) The Named Programmer-defined exception.

The database programming sometimes becomes critical due to large number of tables present in the database and also due to relations among the tables. So sometimes, it is not possible to handle all the unwanted exceptions using Named and Un-named exceptions. That’s why we need to name and trap our own exceptions - ones that aren't defined already by PL/SQL. These are called Named Programmer-defined exceptions.

Points to ponder while using Named Programmer-defined exceptions:

• Named Programmer-defined exceptions should be explicitly declared in the declaration section.
• They should be explicitly raised in the execution section.
• They should be handled by referencing the user-defined exception name in the exception section.

The syntax for the Named Programmer-Defined Exception in a procedure is:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
   [ (param1 [,param2]) ]
IS
   [declaration_section]
   exception_name EXCEPTION; // programmer defined
exception
BEGIN
   executable_section
   RAISE exception_name ; // rasing the programmer
defined exception
EXCEPTION
   WHEN exception_name THEN
      [statements]          // handling the programmer defined
exception
   WHEN OTHERS THEN
      [statements]
END [procedure_name];
/
```

The syntax for the Named programmer-defined exception in a function is:

```
CREATE [OR REPLACE] FUNCTION function_name
   [ (param1 [,param2]) ]
   RETURN return_datatype
IS | AS
   [declaration_section]
   exception_name EXCEPTION;
BEGIN
   executable_section
   RAISE exception_name ;
EXCEPTION
   WHEN exception_name THEN
      [statements]
   WHEN OTHERS THEN
      [statements]
END [function_name];
/
```

Here is an example of a procedure that uses a Named Programmer-Defined Exception:

Example:

```
CREATE OR REPLACE PROCEDURE new_customer(cut_id IN
NUMBER, CP_in IN VARCHAR)
IS
   no_customer EXCEPTION;
BEGIN
   IF CP_in = 'NULL' THEN
      RAISE no_customer;
ELSE
   INSERT INTO customerer (cid, CP )VALUES (cust_id, CP_in );
END IF;
EXCEPTION
    WHEN no_customer THEN
      raise_application_error (-20001,'We must have customer
Product in order to submit  customer.');
   WHEN OTHERS THEN
      raise_application_error (-20002,'An error has occurred
inserting an Supplier.');
END;
/
```

In this example, we have declared a Named programmer-defined Exception called **no_customer** in our declaration statement with the following code:

no_ customer EXCEPTION;

We've then raised the exception in the executable section of the code:

```
IF CP_in = 'NULL' THEN
     RAISE no_ customer;
```

Now if the **CP_in** variable contains a NULL, our code will jump directly to the Named programmer-defined exception called **no_ customer**.

Finally, we can tell the procedure what to do when the **no_ customer** exception is encountered by including code in the WHEN clause:

```
WHEN no_ customer THEN
  raise_application_error (-20001,'We must have Customer
Product in order to submit customer.');
```

11.7 WHEN OTHERS CLAUSE:

The WHEN OTHERS, clause is used to trap all the remaining exceptions that not been handled by our Named System Exceptions and Named Programmer-Defined Exceptions. In simple words we can say that WHEN OTHERS, clause is used to handle all the unknown exception or default exception handling block.
The syntax for the WHEN OTHERS clause in a procedure is:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
   [ (param1 [,param2]) ]
IS
   [declaration_section]
BEGIN
   executable_section // section to monitor for errors
EXCEPTION
   WHEN exception_name1 THEN
      [statements]       // handling the exception
   WHEN exception_name2 THEN
      [statements]    // handling the exception
   WHEN exception_name_n THEN
      [statements]    // handling the exception

   WHEN OTHERS THEN
      [statements]    // handling the all uncaught exception
END [procedure_name];
/
```

The syntax for the WHEN OTHERS clause in a function is:

```
CREATE [OR REPLACE] FUNCTION function_name
   [ (param1 [,param2]) ]
   RETURN return_datatype
IS | AS
   [declaration_section]
BEGIN
   executable_section  // Section to monitor for errors
EXCEPTION
   WHEN exception_name1 THEN
      [statements]    // handling the exception
   WHEN exception_name2 THEN
      [statements]       // handling the exception
   WHEN exception_name_n THEN
      [statements]    // handling the exception
```

```
    WHEN OTHERS THEN
        [statements]      // handling the all uncaught exception
END [function_name];
/
```

Here is an example of a procedure that uses a WHEN OTHERS clause:

```
CREATE OR REPLACE PROCEDURE new_customer(cust_id IN
NUMBER, CP_in IN VARCHAR)
IS
   no_cust EXCEPTION;
BEGIN
   IF CP_in = 'NULL' THEN
      RAISE no_cust;
ELSE
   INSERT INTO customer(cid, cp ) VALUES ( cust_id, CP_in );
END IF;
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
      raise_application_error (-20001,'We have tried to insert a
duplicate cid.');
   WHEN no_cust THEN
      raise_application_error (-20001,'We must have CP in order to
submit the customer.');
   WHEN OTHERS THEN
      raise_application_error (-20002,'An error has occurred
inserting an customer.');
END;
/
```

In this example, if an exception is encountered that is not a **DUP_VAL_ON_INDEX** or a **no_cust**, it will be trapped by the WHEN OTHERS clause.

11.8 THE SQLCODE FUNCTION

The SQLCODE function returns the error number associated with the most recently raised error exception. This function should only be used within the Exception Handling section of our code:

```
EXCEPTION
   WHEN exception_name1 THEN
      [statements]
   WHEN exception_name2 THEN
      [statements]
   WHEN exception_name_n THEN
      [statements]
   WHEN OTHERS THEN
      [statements]  // Here we can use the SQLCODE
function
END [procedure_name];
/
```

We could use the SQLCODE function to raise an error as follows:

Example :

```
EXCEPTION
 WHEN OTHERS THEN
raise_application_error(-20001,'An error was encountered -
'||SQLCODE||' -ERROR- '||SQLERRM);
END;
/
```

Or we could log the error to a table as follows:

Example:

```
EXCEPTION
  WHEN OTHERS THEN
     err_code := SQLCODE;
     err_msg := substr(SQLERRM, 1, 200);
     INSERT INTO audit_table (error_number,
error_message)
     VALUES (err_code, err_msg);
END;
/
```

11.9 THE SQLERRM FUNCTION

The SQLERRM function returns the error message associated with the most recently raised error exception. This function should only be used within the Exception Handling section of our code:

```
EXCEPTION
   WHEN exception_name1 THEN
      [statements]
```

```
   WHEN exception_name2 THEN
      [statements]
   WHEN exception_name_n THEN
      [statements]
   WHEN OTHERS THEN
      [statements]  // Here we can use the SQLERRM
function
END [procedure_name];
/
```

We could use the SQLERRM function to raise an error as follows:

Example:

```
EXCEPTION
  WHEN OTHERS THEN
    raise_application_error(-20001,'An error was encountered -
'||SQLCODE||' -ERROR- '||SQLERRM);
END;
/
```

Or we could log the error to a table as follows:

Example:

```
EXCEPTION
  WHEN OTHERS THEN
    err_code := SQLCODE;
    err_msg := substr(SQLERRM, 1, 200);
    INSERT INTO audit_table (error_number,
error_message)
    VALUES (err_code, err_msg);
END;
/
```

11.10 THE RAISE_APPLICATION_ERROR ( )

The **RAISE_APPLICATION_ERROR** is a built-in procedure in oracle which is used to display the user-defined error messages along with the error number whose range is in between -20000 and -20999.

Whenever a message is displayed using RAISE_APPLICATION_ERROR, all previous transactions which are not committed within the PL/SQL block are rolled back automatically (i.e. change due to INSERT, UPDATE, or DELETE statements).

RAISE_APPLICATION_ERROR raises an exception but does not handle it.

RAISE_APPLICATION_ERROR is used for the following reasons,
a) Used to create a unique id to the user-defined exception.
b) Used to make the user-defined exception look like an Oracle error.

The General Syntax to use this procedure is:

RAISE_APPLICATION_ERROR (error_number, error_message);

• The Error number must be between -20000 and -20999
• The Error_message is the message we want to display when the error occurs.

Steps to be followed to use RAISE_APPLICATION_ERROR procedure:
1. Declare a user-defined exception in the declaration section.
2. Raise the user-defined exception based on a specific business rule in the execution section.
3. Finally, catch the exception and link the exception to a user-defined error number in RAISE_APPLICATION_ERROR.

Using the following example we can display a error message using RAISE_APPLICATION_ERROR.

```
DECLARE
  large_quantity EXCEPTION;
  CURSOR product_quantity is
  SELECT p.pname as name, sum(o.total_unit) as units
  FROM orders o, product p
  WHERE o.pid = p.pid;
  quantity orders.total_unit%type;
  up_limit CONSTANT orders.total_unit%type := 20;
  message VARCHAR2(50);
BEGIN
  FOR product_rec in product_quantity LOOP
    quantity := product_rec.units;
     IF quantity > up_limit THEN
        RAISE large_quantity;
     ELSIF quantity < up_limit THEN
      message:= 'The number of unit is below the discount
limit.';
     END IF;
     dbms_output.put_line (message);
  END LOOP;
 EXCEPTION
```

```
    WHEN large_quantity THEN
        raise_application_error(-2100, 'The number of unit is
above the discount limit.');
    END;
/
```

11.11 QUESTIONS

1. Explain Exception. Explain the syntax of exception handling in PL/SQL.
2. Explain the parts of exception message in PL/SQL.
3. Write short note on scope rules of PL/SQL exception.
4. List and explain types of PL/SQL exception in short.
5. List and explain NAMED System exceptions.
6. What UNNAMED exception? Explain with examples.
7. Explain in detail programmer defined exception with examples.
8. Explain use of WHEN OTHER clause with example.
9. Write short note on SQLCODE and SQLERRM functions.
10. Explain in detail use of RAISE_APPLICATION_ERROR ( ).

Practice Questions:

11. Write a PL/SQL block of code which shows the use of exception handling.
12. Write a PL/SQL block of code which shows the use of user define exception.

11.12 FURTHER READING

- Murach's Oracle SQL and PLSQL by Joel Murach, Murach and Associates.
- Oracle Database 11g PL/SQL Programming Workbook, ISBN: 9780070702264,
  By: Michael McLaughlin,John Harper, TATAMCGRAW-HILL
- Oracle PL/SQL Programming, Fifth Edition By Steven Feuerstein, Bill Pribyl
- Oracle 11g: SQL Reference Oracle press
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, TATAMCGRAW-HILL
- Oracle database 11g: hands on SQL/PL SQL by Satish Asnani (PHI) EEE edition

❖❖❖❖

UNIT - V

12

STORED PROCEDURE

Unit Structure

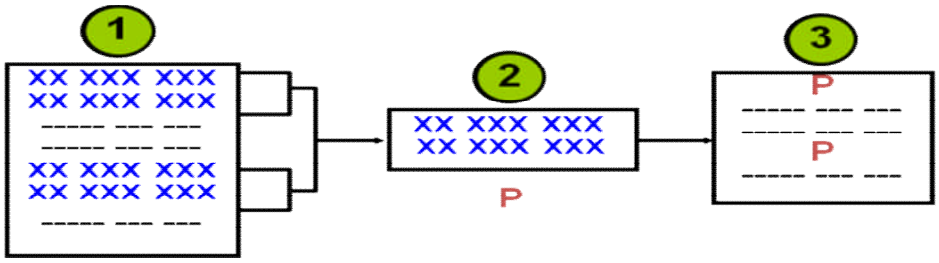
12.1 OBJECTIVES:

After completing this chapter, you will be able to:

- Create a Modularized and Layered Subprogram
- Understand the advantages of modular program constructs
- Understand the Creation and use of Stored Procedure
- Understand the Functions and its parameters
- Understand the Difference between Procedures & Functions
- Utilizing the stored procedures in critical PLSQL queries

12.2 CREATING A MODULARIZED AND LAYERED SUBPROGRAM DESIGN

The modularized and layered subprogram design is nothing but the appropriate arrangement of PLSQL blocks or procedures while writing the code. This arrangement gives the readability to code as well as ease to handle and modify the code for developer. With this code anybody can have the sense to understand the code and its execution hierarchy.



12.2.1 Modularize code into subprograms.

1. Locate code sequences repeated more than once.
2. Create subprogram P containing the repeated code.
3. Modify original code to invoke the new subprogram.

12.2.2 Create subprogram layers for your application.

1. Data access subprogram layer with SQL logic
2. Business logic subprogram layer, which may or may not use data access layer

PL/SQL is a block-structured language. The PL/SQL code block helps modularize code by using:

1. Anonymous blocks
2. Procedures and functions
3. Packages
4. Database triggers

12.3 THE BENEFITS OF USING MODULAR PROGRAM CONSTRUCTS:

The PLSQL modular programs Subprograms support reusability. Once tested, a subprogram can be reused in any number of applications. We can call PL/SQL subprograms from many different environments, so we can use it in any new language or API to access the database.

Subprograms promote maintainability. We can change the internals of a subprogram without changing other subprograms that call it. Subprogram plays a major role in other maintainability features, such as packages and object types.

The PLSQL modular programs allow us to extend the PL/SQL language. Procedures act like new statements. Functions act like new expressions and operators. Subprogram allows us to break a program down into manageable, well-defined modules. We can use stepwise refinement approach to problem solving.

Model subprograms allow us to distinguish the definition of procedures and functions unless the main program is tested. You can design applications with the top down approach without worrying about implementation details.

When we use PL/SQL subprograms to define an API, we can make our code even more reusable and maintainable by grouping the subprograms into a PL/SQL package.

We can summarize the benefits of PLSQL modular program construct as follows:

1. ***Easy to maintain***: Because the code is well arranged so it very easy to maintain for the developers.
2. ***Better data security and integrity***: The code is separated in layered paradigm so that it helps to improve the security and the data integrity.
3. ***Better performance***: Due to separation and sequential arrangement the code suppose to give improved performance.
4. ***Better code clarity***: The code is layered and every layer contains the self explanatory code which improves the code readability and clarity.

12.4 WHAT IS A STORED PROCEDURE?

A stored procedure or in simple a subroutine or a proc or a subprogram is a *named PL/SQL block* which performs one or more specific task. The stored procedures are written in advance and compiled before its use. This improves the speed of execution. This is similar to a procedure or functions in other programming languages. A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure. The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block. A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.

12.5 COMPARISON OF ANONYMOUS BLOCKS AND SUB PROGRAMS IN PL SQL:

- Anonymous is unnamed PL/SQL block, cannot save in database, cannot allow any mode of parameter.
- Stored programs are saved into database and we can recall them whenever program requires it, it accepts the mode of parameter like in, in out, out.
- An anonymous block is a PL/SQL block that appears in our application and is not named. A stored procedure or a named block is a PL/SQL block that oracle stores in the database and can be called by name from any application.
- Anonymous blocks are not stored in the database so they cannot be called from other blocks; whereas stored subprograms are stored in the database they can be called from other blocks many times.
- Anonymous blocks are compiled each time they are executed, where as stored subprograms compile only one time when they are created.

12.5.1 Stored Program Units (The Procedures & The Functions)

A stored procedure and functions are PL/SQL program unit which has a name, which can take parameters, and can return values, which are stored in the data dictionary and also they can be called by many users.

The term stored procedure is sometimes used generically for both stored procedures and stored functions. The only difference between procedures and functions is that functions always return a single value to the caller, while procedures do not return a value to the caller.

12.5.2 Naming Procedures and Functions

A PLSQL procedure or function must be named because it is stored in the database. Each publicly-visible procedure or function in a schema must have a unique name, and the name must be a legal PL/SQL identifier.

If we plan to call a stored procedure using a stub generated by SQL*module, then the stored procedure name must also be a legal identifier in the calling host 3GL language, such as Ada or C.
The point here to remember that, the name of PLSQL procedure is the unique identifier in database system. So we are not able to make more than one procedure with similar name.

- **CREATE PROCEDURE :**

The general format of a create procedure statement is :

```
CREATE [OR REPLACE] PROCEDURE procedure_name
   [ (param1 [,param2]) ]
IS
   [declaration_section]   // Application variables
BEGIN
   [executable_section]  // Application Logic
EXCEPTION
  [  exception_section]  // Exception handling statements
END [procedure_name];
/
```

The following is a simple example of a procedure:

Example 1: (*Simple procedure without parameter***)**

```
SQL> set serveroutput on
SQL> CREATE OR REPLACE PROCEDURE DEMO AS
  2  BEGIN
  3  DBMS_OUTPUT.PUT_LINE('HELLO WORLD');
  4  END;
  5  /
```

Procedure created.

Example 2 : (*Simple procedure with parameters***)**
Note:(*This procedure can be called Using PLSQL block as shown below point 3rd of CALL procedure section***.)**

```
SQL> CREATE OR REPLACE PROCEDURE Square(sq_num INT,
sq OUT INT) AS
  2  BEGIN
  3  sq:= sq_num*sq_num;
  4   DBMS_OUTPUT.PUT_LINE('Square of entered number is ' ||
sq );
  5  END;
  6  /
```

Procedure created.

- **CALL PROCEDURE :**

You can call procedure in three ways-

1.Using EXECUTE

Example 1 Executl :

```
SQL> EXECUTE DEMO
HELLO WORLD
```

PL/SQL procedure successfully completed.

2.Using CALL
Example 1 Call:

```
SQL> call HELLO();
Hello World
Call completed.
```

3.Using PL/SQL block
Example 1 call :

```
SQL> begin
  2  HELLO();
  3  end;
  4  /
Hello World
PL/SQL procedure successfully completed.
```

Example 2 call :

```
SQL> Declare
  2  my_num int;
  3  Begin
  4  SQUARE(4,my_num);
  5  END;
  6  /
Square of entered number is 16
PL/SQL procedure successfully completed.
```

12.6 PROCEDURE PARAMETERS

The Parameter modes define the behavior of formal parameters. The three parameter modes, IN (That is always Default), OUT, and IN OUT, can be used with any subprogram wherever necessary. However, avoid using the OUT and IN OUT modes with functions. The purpose of a function is to take no arguments and return a single value. There are three types of parameters that can be declared:

IN	**OUT**	**IN OUT**
It is the default.	It must be specified.	It must be specified.
Passes values to a subprogram.	Returns values to the caller.	Passes initial values to a subprogram; returns updated values to the caller.

Formal parameter acts like a constant.	Formal parameter acts like an uninitialized variable.	Formal parameter acts like an initialized variable.
Formal parameter cannot be assigned a value.	Formal parameter cannot be used in an expression; must be assigned a value.	Formal parameter should be assigned a value.
Actual parameter can be a constant, initialized variable, literal, or expression.	Actual parameter must be a variable.	Actual parameter must be a variable.

Example 1:

12.6.1 IN and OUT MODE

```
SQL> CREATE OR REPLACE
  2  PROCEDURE SUM_AB (A IN INT, B IN INT, C OUT INT) IS
  3 BEGIN
  4  C := A + B;
  5  END;
  6  /

Procedure created.

SQL> DECLARE
  2  R INT;
  3  BEGIN
  4  SUM_AB(23,29,R);
  5  DBMS_OUTPUT.PUT_LINE('SUM IS: ' || R);
  6  END;
  7  /
SUM IS: 52

PL/SQL procedure successfully completed.
```

Example 2:

```
SQL>  CREATE OR REPLACE  PROCEDURE CONCAT
          ( subchars1 IN STRING, subchars2 IN STRING, ConcChar OUT
STRING ) IS
```

```
  2    BEGIN
  3    ConcChar:= subchars1 + subchars2;
  4     DBMS_OUTPUT.PUT_LINE('After Concatination of characters word
is ' || ConcChar );
  5    END;
  6   /

Procedure created.

SQL> DECLARE
  2    my_char STRING(10);
  3    BEGIN
  4    CONCAT('SONA','LI', my_char);
  5    END;
  6    /

After Concatination of characters word is SONALI
PL/SQL procedure successfully completed.
```

12.6.2 IN OUT MODE

Example 1 :

```
SQL> set serveroutput on
SQL> CREATE OR REPLACE
  2  PROCEDURE DOUBLEN (N IN OUT INT) IS
  3  BEGIN
  4  N := N * 2;
  5  END;
  6  /

Procedure created.
SQL> DECLARE
  2  R INT;
  3  BEGIN
  4  R := 7;
  5  DBMS_OUTPUT.PUT_LINE('BEFORE CALL R IS: ' || R);
  6  DOUBLEN(R);
  7  DBMS_OUTPUT.PUT_LINE('AFTER CALL R IS: ' || R);
  8  END;
  9  /
BEFORE CALL R IS: 7
AFTER CALL R IS: 14
PL/SQL procedure successfully completed.
```

Example 2 :

```
SQL>  CREATE OR REPLACE PROCEDURE DIVISION
                        (my_num1 int,my_num2 IN OUT INT) IS
  2    BEGIN
  3    my_num2 := my_num1 / my_num2;
  4    END;
  5   /
```

Procedure created.

```
SQL>  DECLARE
  2    num1 INT;
  3    num2 INT;
  4    BEGIN
  5    num1 := 180;
  6    num2 := 18;
  7     DBMS_OUTPUT.PUT_LINE('BEFORE IN OUT DIVISION NUMBER
IS: ' || num2);
  8    DIVISION(num1,num2);
  9       DBMS_OUTPUT.PUT_LINE('AFTER CALL to DIVSION IN OUT
NUMBER IS: ' || num2);

 10    END;
 11    /
```

```
BEFORE IN OUT DIVISION NUMBER IS: 18
AFTER CALL to DIVSION IN OUT NUMBER IS: 10
```

PL/SQL procedure successfully completed.

- **HOW TO VIEW SOURCE CODE OF PROCEDURE?**

```
SQL>  SELECT  TEXT  FROM  USER_SOURCE  WHERE
NAME='HELLO';
TEXT
------------------------------------------------------------
PROCEDURE HELLO IS
BEGIN
DBMS_OUTPUT.PUT_LINE('Hello World');
END;
```

- **DROP PROCEDURE**

If we're interested in getting rid of a procedure totally, we can DROP it. The general
format of a DROP is:

DROP PROCEDURE procedure_name;

```
SQL>DROP PROCEDURE DEMO;
```

Procedure dropped.

- **VIEW LIST OF ALL PROCEDURE**

```
SQL> SELECT OBJECT_NAME
  2    FROM USER_OBJECTS
  3    WHERE OBJECT_TYPE = 'PROCEDURE';

OBJECT_NAME
----------------------------------------
DISPLAYINFO
DEMO
DOUBLEN
SUM_AB
DISP_AB

5 rows selected.
```

12.7 EXAMPLE OF PROCEDURE

```
SQL> create table student(
  2   rollno number(4) primary key, name varchar(4),
  3  mark1 number(4),
  4   mark2 number(4),
  5  mark3 number(4)
  6  );

Table created.

SQL> insert into student values(1,'abc',56,78,89);

1 row created.

SQL> insert into student values(2,'pqr',55,44,66);

1 row created.

SQL> insert into student values(3,'xyz',56,78,89);

1 row created.

SQL>  insert into student values(4,'qwe',55,44,66);

1 row created.

SQL> insert into student values(5,'iop',56,78,89);

1 row created.

SQL> insert into student values(6,'tgb',55,44,66);
```

1 row created.

SQL> select * from student;

ROLLNO	NAME	MARK1	MARK2	MARK3
1	abc	56	78	89
2	pqr	55	44	66
3	xyz	56	78	89
4	qwe	55	44	66
5	iop	56	78	89
6	tgb	55	44	66

==========

```
CREATE OR REPLACE PROCEDURE displayinfo(
            p_rollno IN student.rollno%TYPE,
            o_name OUT student.name%TYPE,
            o_mark1 OUT  student.mark1%TYPE,
            o_mark2 OUT  student.mark2%TYPE,
            o_mark3 OUT  student.mark3%TYPE)
IS
BEGIN
  SELECT name,mark1,mark2,mark3
  INTO o_name,o_mark1,o_mark2 ,o_mark3
  FROM  student WHERE rollno = p_rollno;
  END;
/
```

==========

```
DECLARE
   o_name  student.name%TYPE;
   o_mark1 student.mark1%TYPE;
   o_mark2 student.mark2%TYPE;
   o_mark3 student.mark3%TYPE;
    total NUMERIC;
BEGIN
   displayinfo(1,o_name,o_mark1,o_mark2 ,o_mark3);
   DBMS_OUTPUT.PUT_LINE('name :  ' || o_name);
   DBMS_OUTPUT.PUT_LINE('mark1 :  ' || o_mark1);
   DBMS_OUTPUT.PUT_LINE('mark2 :  ' || o_mark2);
   DBMS_OUTPUT.PUT_LINE('mark3 :  ' || o_mark3);
   total:=o_mark1+o_mark2 +o_mark3 ;
   DBMS_OUTPUT.PUT_LINE('Total :  ' || total);
 END;
```

PL/SQL procedure successfully completed.
name : abc
mark1 : 56
mark2 : 78
mark3 : 89
Total : 223

12.8 STORED FUNCTIONS

A **stored function** (also called a **user function** or **user defined function**) is a set of PL/SQL statements you can call by name. Stored functions are very similar to procedures, except that a function returns a value to the environment in which it is called. User functions can be used as part of a SQL expression. Use the CREATE FUNCTION statement to create a standalone stored function.

Functions are special types of procedures that have the capability to return a value. It is very oblivious question of when to use what, either functions or procedures. If we're interested in the "results" of the code, then we use a function, and return those results. If we are interested in the "side effects" (like table updates, etc.) and not about the "result", then use a procedure. Usually it doesn't affect the code all that much if we use a procedure or a function.

The general format of a create function statement is :

```
CREATE [OR REPLACE] FUNCTION function_name
   [ (param1 [,param2]) ]
   RETURN return_datatype
IS | AS
   [declaration_section] // Application variables
BEGIN
  [ executable_section]   // Application logic
EXCEPTION
 [ exception_section]   // Exception handling Code
END [function_name];
/
```

EXAMPLE 1 :

```
SQL> set serveroutput on
SQL> CREATE OR REPLACE FUNCTION ADD2 (X INT, Y INT)
RETURN INT IS
  2  BEGIN
  3  RETURN (X + Y);
  4  END;
```

```
  5  /
```
Function created.

Example 2:

```
SQL> CREATE OR REPLACE FUNCTION STRINGRETURN ( myname STRING) RETURN STRING IS
  2  BEGIN
  3  RETURN (myname);
  4  END;
  5  /

Function created.
```

CALL FUNCTION :

Call for Example 1:

```
SQL> BEGIN
   DBMS_OUTPUT.PUT_LINE ('RESULT IS:' || ADD2 (25,50));
   END;
   /
RESULT IS:75
```
PL/SQL procedure successfully completed.

Call for Example 2:

```
SQL>  BEGIN
1    DBMS_OUTPUT.PUT_LINE ('MY NAME IN FUNCTION IS: ' ||
                              STRING_RETURN('YASHASHREE'));
  3   END;
  4   /
MY NAME IN FUNCTION IS: YASHASHREE

PL/SQL procedure successfully completed.
```

12.9 FUNCTION PARAMETERS

The Parameter modes define the behavior of formal parameters. The three parameter modes, IN (That is always Default), OUT, and IN OUT, can be used with any subprogram wherever necessary. However, avoid using the OUT and IN OUT modes with functions. The purpose of a function is to take no arguments and return a single value.. There are three types of parameters that can be declared:

1. **IN** - The parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or function.

2. **OUT** - The parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.

3. **IN OUT** - The parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

SOURCE CODE OF FUNCTION

```
SQL> SELECT TEXT FROM USER_SOURCE WHERE NAME='ADD2';
TEXT
----------------------------------------------------------
FUNCTION ADD2 (X INT, Y INT) RETURN INT IS
BEGIN
RETURN (X + Y);
END;
```

DROP FUNCTION

If you're interested in getting rid of a function totally, you can DROP it.
The general format of a DROP is:
DROP FUNCTION function_name;

```
SQL>DROP FUNCTION ADD2;
Procedure dropped.
```

LIST OF ALL FUNCTION

```
SQL> SELECT OBJECT_NAME
  2      FROM USER_OBJECTS
  3      WHERE OBJECT_TYPE = 'FUNCTION';

OBJECT_NAME
-----------------------------------------------
DISPLAY1
ADD2
5 rows selected.
```

12.10 EXAMPLE OF FUNCTION

```
SQL> CREATE OR REPLACE FUNCTION display(
  2      p_rollno IN student.rollno%TYPE,
  3      o_name OUT student.name%TYPE,
  4      o_mark1 OUT  student.mark1%TYPE,
  5      o_mark2 OUT  student.mark2%TYPE,
  6      o_mark3 OUT  student.mark3%TYPE)
```

```
 7  RETURN NUMBER
 8  IS total NUMBER;
 9  BEGIN
10    SELECT name,mark1,mark2,mark3
11    INTO o_name,o_mark1,o_mark2 ,o_mark3
12    FROM  student WHERE rollno = p_rollno;
13    total:=o_mark1+o_mark2 +o_mark3 ;
14   RETURN(total);
15    END;
16  /
```

Function created.

```
SQL> DECLARE
  2     o_name  student.name%TYPE;
  3     o_mark1 student.mark1%TYPE;
  4     o_mark2 student.mark2%TYPE;
  5     o_mark3 student.mark3%TYPE;
  6      total NUMERIC;
  7  BEGIN
  8    total:= display(1,o_name,o_mark1,o_mark2 ,o_mark3);
  9     DBMS_OUTPUT.PUT_LINE('name :  ' || o_name);
 10     DBMS_OUTPUT.PUT_LINE('mark1 :  ' || o_mark1);
 11     DBMS_OUTPUT.PUT_LINE('mark2 :  ' || o_mark2);
 12     DBMS_OUTPUT.PUT_LINE('mark3 :  ' || o_mark3);
 13     DBMS_OUTPUT.PUT_LINE('Total :  ' || total);
 14   END;
 15  /
name :  abc
mark1 :  56
mark2 :  78
mark3 :  89
Total :  223
```

PL/SQL procedure successfully completed.

12.11 DIFFERENCE BETWEEN PROCEDURES & FUNCTIONS

Procedures are traditionally the workhorse of the coding world and functions are traditionally the smaller, more specific pieces of code. In general, if you need to update the chart of accounts, you would write a procedure. If you need to retrieve the

organization code for a particular GL account, you would write a function.

Here are a few more differences between a procedure and a function:

- A function MUST return a value.
- A procedure cannot return a value.
- Procedures and functions can both return data in OUT and IN OUT parameters
- The return statement in a function returns control to the calling program and returns the results of the function
- The return statement of a procedure returns control to the calling program and cannot return a value
- Functions can be called from SQL, procedure cannot
- Functions are considered expressions, procedure are not

12.12 QUESTIONS

1. How to create Modularized and Layered Subprogram?
2. What is a Stored Procedure?
3. Explain Difference between anonymous blocks and sub programs in PL SQL.
4. How to create and call Stored Procedure Explain with help of Example.
5. Write short note on PROCEDURE PARAMETERS.
6. Give the example for using PARAMETERS in Procedures.
7. What are stored functions? Write its Creation and calling Example.
8. Write short note on Function PARAMETERS.
9. Give a brief example of Stored Function.
10. Explain the Difference between Procedures & Functions.

12.13 FURTHER READING

- Murach's Oracle SQL and PLSQL by Joel Murach, Murach and Associates.
- Oracle Database 11g PL/SQL Programming Workbook, ISBN: 9780070702264,

  By: Michael McLaughlin,John Harper, TATAMCGRAW-HILL

- Oracle PL/SQL Programming, Fifth Edition By Steven Feuerstein, Bill Pribyl
- Oracle 11g: SQL Reference Oracle press
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, TATAMCGRAW-HILL
- Oracle database 11g: hands on SQL/PL SQL by Satish Asnani (PHI) EEE edition

❖❖❖❖

13

PACKAGE

Unit Structure :

13.1 OBJECTIVES

After completing this chapter, you will be able to:

- ❖ Learn and understand the Complete Structures of Packages
- ❖ Understand the advantages of Packages
- ❖ Understand the Components of packages
- ❖ Overload Subprograms in PL/SQL etc.

13.2 WHAT IS PACKAGE?

The PLSQL package is nothing but logical grouping of functions and stored procedures that can called and referenced by the single name.

The package is an encapsulated collection of related program objects for example, procedures, functions, variables, constants, cursors, and exceptions stored together in the database. Also a package is a schema object that groups logically related

PL/SQL types, variables, and subprograms. Using packages is an alternative to creating procedures and functions as standalone schema objects.

Packages have two parts, a specification and a body; sometimes the body is unnecessary. The specification is the interface to the package. It declares the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. The body defines the queries for the cursors and the code for the subprograms. We can think of the specification as an interface and of the body as a black box. We can debug, enhance, or replace a package body without changing the package spec (specification). To create package specs, we have to use the SQL statement **CREATE PACKAGE**. A **CREATE PACKAGE BODY** statement defines the package body. The spec holds public declarations, which are visible to stored procedures and other code outside the package. We must declare subprograms at the end of the spec after all other items.

The body holds implementation details and private declarations, which are hidden from code outside the package. Following the declarative part of the package body is the optional initialization part, which holds statements that initialize package variables and do any other one-time setup steps. The AUTHID clause determines whether all the packaged subprograms execute with the privileges of their definer or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker.

A call spec lets we map a package subprogram to a Java method or external C function. The call spec maps the Java or C name, parameter types, and return type to their SQL counterparts.

13.3 CONTENTS OF PL/SQL PACKAGE:

The following things are contained in a PL/SQL package:

- ***Declarations of cursor with the text of SQL queries****:* Reusing exactly the same query text in multiple locations is faster than retyping the same query each time with slight differences. It is also easier to maintain if we need to change a query that is used in many places.
- Get and Set methods for the package variables, if we want to avoid letting other procedures read and write them directly.
- ***Procedures and functions declaration that call each other****:* We do not need to worry about compilation order for packaged procedures and functions, making them more

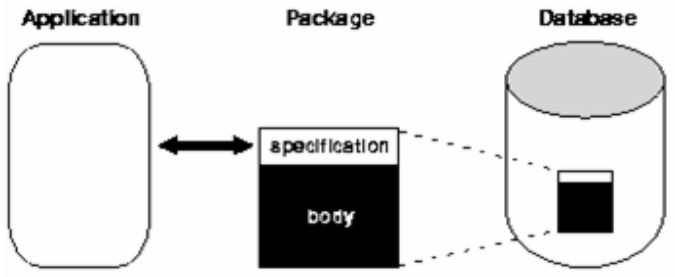
convenient than standalone stored procedures and functions when they call back and forth to each other.

- ***Exceptions declarations****:* Normally, we need to be able to reference these from different procedures, so that we can handle exceptions within called subprograms. The naming and declaration should be in the proper block and scope.
- ***Declarations for overloaded procedures and functions****:* We can create multiple variations of a procedure or function, using the same names but different sets of parameters.
- Variables that we want to remain available between procedure calls in the same session. We can treat variables in a package like global variables.
- Only the declarations in the package spec are visible and accessible to applications. Implementation details in the package body are hidden and inaccessible. We can change the body (implementation) without having to recompile calling programs.
- ***Type declarations for PL/SQL collection types***: To pass a collection as a parameter between stored procedures or functions, we must declare the type in a package so that both the calling and called subprogram can refer to it.

13.4 INTRODUCING TO PL/SQL PACKAGE:

PL/SQL package is a group of related stored functions, procedures, types, cursors and etc. PL/SQL package is like a library once written stored in the Oracle database and can be used by many applications. A package has two parts:

- A package specification is the public interface of your applications. The public here means the stored function, procedures, type … are accessible by other applications.
- A package body contains the code that implements the package specification.



PL/SQL Package

13.5 ADVANTAGES OF PACKAGE:

1. All related function and procedure can be grouped together in a single unit called packages and also all the things can be referred by using single name.
2. Packages are reliable to granting privileges.
3. All function and procedure within a package can share variable among them.
4. Package enables to perform "overloading" of functions and procedures.
5. Package improve performance by loading the multiple object into memory at once, therefore , subsequent calls to related program do not required physical I/O.
6. Package is reduce the traffic because all block execute all at once

13.6 COMPONENTS OF PACKAGES

Specification: It contains the list of various functions, procedure names which will be a part of the package.

Body: This contains the actual PL/SQK statement code implementing the logics of functions and procedures declared in "specification".

Defining Package Specification

```
CREATE or REPLACE PACKAGE <Package Name>
{is,as}
PROCEDURE [Schema..] <ProcedureName>
  (<argument> {IN,OUT,IN OUT} <Data Type>,..);
FUNCTION [Schema..]  <Function Name>
  (<argument> IN <Data Type>,..)
  RETURN <Data Type>);
```

Creating Package Body

```
CREATE or REPLACE PACKAGE BODY <Package Name>
{is,as}
PROCEDURE [Schema..] <ProcedureName>
  (<argument> {IN,OUT,IN OUT} <Data Type>,..)
{IS, AS}
<variable> declarations;
<constant> declarations;

BEGIN
<PL/SQL subprogram body> // Application logic goes here
```

```
EXCEPTION
<PL/SQL Exception block> // Exception handling code goes
here

END;
FUNCTION [Schema..] <FunctionName>(<argument> IN
<Data Type>,..)
  return <Data Type> {IS,AS}
  <variable> declarations;
  <constant> declarations;

BEGIN
  <PL/SQL subprogram body> // Application code goes here

EXCEPTION
  <PL/SQL Exception block> // Exception handling code
goes here

END;

END;
/
```

Example 1 :

Defining Package Specification

```
SQL> create or replace package pkg_demo
  2    as
  3      function cArea (r NUMBER) return NUMBER;
  4      procedure pPrint (str1 VARCHAR2 :='hello',
  5        str2 VARCHAR2 :='world',
  6        str3 VARCHAR2  :='!' );
  7   end;
  8   /

Package created.
```

Defining Package Body

```
SQL> create or replace package body pkg_demo
  2     as
  3       function cArea (r NUMBER)return NUMBER
  4       is
  5          pi NUMBER:=3.14;
  6       begin
  7         return (pi * r *r);
  8       end;
  9       procedure pPrint(str1 VARCHAR2 :='hello',
 10                    str2 VARCHAR2 :='world',
```

```
11        str3 VARCHAR2  :='!' )
12         is
13         begin
14            DBMS_OUTPUT.put_line(str1||','||str2||str3);
15         end;
```

Empname	empage	salary	Job_title	Hiredate	empid
Amol	29	29000	Manager	23/11/1998	1
Suresh	27	26000	Developer	16/10/1999	2
Kapil	28	28000	Designer	05/08/2000	3
Pappu	22	17000	Designer	18/06/2001	4
Prashant	25	18000	Tester	11/10/2002	5
KomaL	23	18500	Executive	23/10/2003	6

```
16     end;
17     /
```

Package body created.

Example 2 :
To use following Example we need to create table as shown in following digram:
Table : **emp**

```
SQL> CREATE SEQUENCE empid;

Sequence created.

SQL> CREATE OR REPLACE PACKAGE emp_constraints
AS  -- spec
 2     TYPE emprectype IS RECORD (empid INT, salary
REAL);
 3    CURSOR draw_salary RETURN emprectype;
 4   PROCEDURE hire_emp (
 5  empname  VARCHAR2,
 6  empage VARCHAR2,
 7  salary    NUMBER,
 8  Hiredate date,
 9  job    VARCHAR2,empid int );
10
11   PROCEDURE fire_employee (emp_id NUMBER);
12  END emp_constraints ;
13  /

Package created.
```

```
SQL> CREATE OR REPLACE PACKAGE BODY emp_ constraints
AS  -- body
  2  CURSOR draw_salary RETURN emprectype IS
  3    SELECT empid, salary FROM emp ORDER BY salary DESC;
  4     PROCEDURE hire_emp (
  5  empname  VARCHAR2,
  6  empage VARCHAR2,
  7  salary    NUMBER,
  8  Hiredate date,
  9  job    VARCHAR2,
 10  empid int) IS
 11     BEGIN
 12     INSERT INTO emp VALUES
 13
(empname,empage,salary,SYSDATE,job,empid_seq.NEXTVAL);
 14     END hire_emp;
 15
 16     PROCEDURE fire_employee(emp_id NUMBER) IS
 17     BEGIN
 18       DELETE FROM emp WHERE eid = empid;
 19     END fire_employee;
 20  END emp_ constraints ;
 21  /

Package body created.
```

- ***Call Package Function(For Example 1)***

```
SQL> SELECT pkg_demo.cArea(2) FROM DUAL;

PKG_DEMO.CAREA(2)
-----------------
          12.56
```

- ***Call Package Procedure(For Example 1)***

```
SQL> call pkg_demo.pPrint();
hello,world!

Call completed.
```

- **Package Alter**

Package Alter Syntax

```
ALTER PACKAGE <Package Name> COMPILE BODY;
/
```

Package Alter Code:

```
SQL>ALTER PACKAGE pkg1 COMPILE BODY;

Package body Altered.
```

- **Package Drop**

Package Drop Syntax:

DROP PACKAGE <Package Name>;

Package Drop Code:

SQL>DROP PACKAGE pkg1;

Package dropped.

13.7 DATA DICTIONARY AND PL/SQL SOURCE CODE:

Oracle's **data dictionary** provides information that Oracle needs to perform its tasks. This information consists of *definition*, *allocated* and *used* storage size for database objects, default column values, integrity constraints, names of and privileges granted to users, auditing information and more.

The datadictionary is stored in a few tables owned by SYS (the so called dictionary base tables. Their content is exposed through the static dictionary views. These views and tables should not be written to, only selected. The base tables are stored in the system tablespace (which is always available when the database is open).

The data dictionary is updated when a DDL statement is executed. The following *views* are part of the data dictionary.

Find all views along with a comment in **dict**:

select * from dict;

13.8 OVERLOADING SUBPROGRAMS IN PL/SQL:

- 2 or more procedures or functions are called overloaded when
  a) They have the same names
  b) Different no of formal parameters defined
  c) Formal parameters differ in their datatypes/Subtypes and not in the same family
  d) They belong to same subprogram/package/PL SQL Block
- They are not overloaded when
  a) Any of the above points are not satisfied
  b) RETURN datatype alone differs and not the formal parameters

The main purpose of overloading procedures is that, when a PL SQL block is found to do a same operation with different inputs, we names them same and feed different parameters

Example :

```
SQL> create or replace package pkg_demo1
  2    as
  3      procedure pOvr(a in int, b in int);
  4      procedure pOvr (str1 VARCHAR2 :='hello',
  5                      str2 VARCHAR2 :='world',
  6                      str3 VARCHAR2  :='!' );
  7   end;
  8   /

Package created.

SQL> create or replace package body pkg_demo1
  2     as
  3        procedure pOvr(a in int, b in int)
  4       is
  5       c int;
  6       begin
  7           c := a + b;
  8          dbms_output.put_line(c);
  9       end;
 10        procedure pOvr(str1 VARCHAR2 :='hello',
 11                       str2 VARCHAR2 :='world',
 12                       str3 VARCHAR2  :='!' )
 13       is
 14       begin
 15          dbms_output.put_line(str1||','||str2||str3);
 16       end;
 17    end;
 18    /

Package body created.

SQL> call pkg_demo1.pOvr();
hello,world!

Call completed.

SQL> call pkg_demo1.pOvr(2,3);
5

Call completed
```

13.9 THE STANDARD PACKAGE:

How Package STANDARD Defines the PL/SQL Environment?

The package STANDARED provides some predefined functions, procedures in the PL/SQL that are handy when developing the PL/ASQL applications. It defines the PL/SQL environment. The package spec globally declares types, exceptions, and subprograms, which are available automatically to PL/SQL programs. This package can be utilized with bunch of PLSQL subprograms to perform the specific task. The best thing about it is that it can be used n number of times. For example, package STANDARD declares function ABS, which returns the absolute value of its argument, as follows:

FUNCTION ABS (t NUMBER) RETURN NUMBER;

The contents of the package STANDARD are directly visible to applications. We do not need to write references to its contents by prefixing the package name. For example, we might call ABS from a database trigger, stored subprogram, Oracle tool, as follows:

abs_diff := ABS(x - y);

If we redeclare ABS in a PL/SQL program, our local declaration overrides the global declaration. However, we can still call the built-in function by qualifying the reference to ABS, as follows:

abs_diff := STANDARD.ABS(x - y);

Most built-in functions are overloaded. For example, package STANDARD contains the following declarations:

FUNCTION TO_CHAR (right DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (left DATE, right VARCHAR2) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER, right VARCHAR2) RETURN VARCHAR2;

PL/SQL resolves a call to TO_CHAR by matching the number and datatypes of the formal and actual parameters.

13.10 PRODUCT-SPECIFIC PACKAGES:

The product specific packages helps programmer to use them at specific programming condition. Oracle and various Oracle tools are supplied with product-specific packages that help we build

PL/SQL-based applications. For example, Oracle is supplied with many utility packages, a few of which are highlighted below.

- **DBMS_ALERT Package :**

Package **DBMS_ALERT** allows us to use database triggers to alert an application when specific database values change. The alerts are transaction based and asynchronous (that is, they operate independently of any timing mechanism). For example, a company might use this package to update the value of its investment portfolio as new stock and bond quotes arrive.
We can also use DBS package in the exception handling mechanism to show the error message.

- **DBMS_OUTPUT Package :**

Package **DBMS_OUTPUT** enables us to display output from PL/SQL blocks and subprograms, which makes it easier to test and debug them. The procedure put_line outputs information to a buffer in the SGA. We display the information by calling the procedure get_line or by setting **SERVEROUTPUT ON** in Oracle. For example, suppose we create the following stored procedure:

```
CREATE PROCEDURE Salary_Statement (payroll OUT
NUMBER) AS
  CURSOR c1 IS SELECT salary, empage FROM emp;
BEGIN
  payroll := 0;
  FOR t1rec IN c1 LOOP
    t1rec.empage := NVL(t1rec.empage, 0);
    payroll := payroll + t1rec.salary + t1rec.empage;
  END LOOP;
  /* Display debug info. */
  dbms_output.put_line('Value of payroll: ' ||
TO_CHAR(payroll));
END;
/
```

When we issue the following commands, Oracle displays the value assigned by the procedure to parameter payroll:

```
SQL> SET SERVEROUTPUT ON
SQL> VARIABLE num NUMBER
SQL> CALL Salary_Statement (:num);
```

Value of payroll: 4523

- **DBMS_PIPE Package :**

Package DBMS_PIPE allows different sessions to communicate over named pipes. (A *pipe* is an area of memory used by one process to pass information to another.) We can use

the procedures pack_message and send_message to pack a message into a pipe, send it to another session in the same instance.

At the other end of the pipe, we can use the procedures receive_message and unpack_message to receive and unpack (read) the message. Named pipes are useful in many ways. For example, we can write routines in C that allow external programs to collect information, then send it through pipes to procedures stored in an Oracle database.

- **UTL_FILE Package :**

Package UTL_FILE allows our PL/SQL programs to read and write operating system (OS) text files. It provides a restricted version of standard OS stream file I/O, including open, put, get, and close operations.

When we want to read or write a text file, we call the function fopen, which returns a file handle for use in subsequent procedure calls. For example, the procedure put_line writes a text string and line terminator to an open file, and the procedure get_line reads a line of text from an open file into an output buffer.

- **UTL_HTTP Package :**

Package UTL_HTTP allows our PL/SQL programs to make hypertext transfer protocol (HTTP) callouts. It can retrieve data from the Internet or call Oracle Web Server cartridges. The package has two entry points, each of which accepts a URL (uniform resource locator) string, contacts the specified site and returns the requested data, which is usually in hypertext markup language (HTML) format.

13.11 POINTS TO PONDER FOR WRITING PACKAGES:

When we are writing the packages; we have to keep them as general as possible so they can be reused in future applications. We have to avoid writing packages that duplicate some feature already provided by Oracle. Package specs reflect the design of our application. So, we have to define them before the package bodies. Place in a spec only the types, items, and subprograms that must be visible to users of the package. By this, other developers cannot misuse the package by basing their code on irrelevant implementation details.

To reduce the need for recompiling when code is changed, place as few items as possible in a package spec. Changes to a package body do not require Oracle to recompile dependent

procedures. However, changes to a package spec require Oracle to recompile every stored subprogram that references the package.

Separating Cursor Specs and Bodies with Packages:

We can separate a cursor specification from its body for placement in a package. That way, we can change the cursor body without having to change the cursor spec. We code the cursor spec in the package spec by using this syntax:

CURSOR cursor_name [(param1 [, param2]...)] RETURN return_type;

In the following example, we use the %ROWTYPE attribute to provide a record type that represents a row in the database table employee

```
CREATE or replace PACKAGE emp_salary AS
  CURSOR t1 RETURN employee%ROWTYPE;  -- declare
cursor spec
  END emp_salary;
/


CREATE or replace PACKAGE BODY emp_salary AS
  CURSOR t1 RETURN employee%ROWTYPE IS
    SELECT * FROM employee WHERE emp_sal > 43000;
-- define cursor body
  END emp_ salary;
/
```

The cursor spec has no SELECT statement because the RETURN clause specifies the data type of the return value. However, the cursor body must have a SELECT statement and the same RETURN clause as the cursor spec. Also, the number and data types of items in the SELECT list and the RETURN clause must match.

Packaged cursors increase flexibility. For example, we can change the cursor body in the last example, without having to change the cursor spec.

From a PL/SQL block or subprogram, we use dot notation to reference a packaged cursor, as the following example shows:

```
DECLARE
  emp_rec  employee%ROWTYPE;
BEGIN
  OPEN emp_salary.t1;
  LOOP
```

```
    FETCH emp_salary.t1 INTO emp_rec; /* Do more
processing here... */
    EXIT WHEN emp_salary.t1%NOTFOUND;
  END LOOP;
  CLOSE emp_salary.t1;
END;
 /
```

The scope of a packaged cursor is not limited to a PL/SQL block. When we open a packaged cursor, it remains open until we close or disconnect it from the session.

13.12 QUESTIONS:

1. What are packages in PL/SQL? What are the advantages of Packages?

2. Explain the Components and specification of PL/SQL packages.

3. Give simple example of package specification and body element.

4. Write short note on Data Dictionary and PL/SQL Source Code.

5. How to overload Subprograms in PL/SQL

6. Explain the PL/SQL package BODY with the example.

7. Explain some package features with example.

8. Explain STANDARD package in PL/SQL.

9. Elaborate use of following product specific packages.

- DBMS_ALERT
- DBMS_OUTPUT
- DBMS_PIPE
- UTL_FILE
- UTL_HTTP

10. What points should be taken in consideration for writing PL/SQL packages?

11. How to separating Cursor Specs and Bodies from Packages?

Practice Questions:

12. Create a function that accepts emp_id and check if the emp_id exits in dept table display a message employee present and if not then display the message employee absent.

13. Create a package named myPack that will hold the function created in above exercise. Write the package specification and package body for the package myPack.

14. Write a PL/SQL block of code for a function which calculates square of a number. Use IN OUT parameter.

15. Write a PL/SQL block of code for a procedure which displays the information of a Student table. (Create Student table with proper fields.)

16. Write a PL/SQL block of code for a procedure which displays the message HELLO WORLD.

13.13 FURTHER READING

- Murach's Oracle SQL and PLSQL by Joel Murach, Murach and Associates.
- Oracle Database 11g PL/SQL Programming Workbook, ISBN: 9780070702264,

  By: Michael McLaughlin,John Harper, TATAMCGRAW-HILL
- Oracle PL/SQL Programming, Fifth Edition By Steven Feuerstein, Bill Pribyl
- Oracle 11g: SQL Reference Oracle press
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, TATAMCGRAW-HILL
- Oracle database 11g: hands on SQL/PL SQL by Satish Asnani (PHI) EEE edition

❖❖❖❖

Unit - VI

14

DYNAMIC SQL

Unit Structure

14.1 OBJECTIVES

After completing this chapter, you will be able to:

- Understand the Execution Flow of SQL with PL SQL
- Understand Dynamically Executing a PL/SQL Block
- Understand to use DBMS_SQL Package
- Manage and use Advantages of Native Dynamic SQL
- Understand Performing DML Using Dynamic SQL

14.2 THE EXECUTION FLOW OF SQL

All SQL statements in the database go through various stages. These stages are called as Execution flow of SQL:

- **Parse**: This stage is called as 'pre-execution'. In this stage the code is checked for "is this possible?" possibility including syntax, object existence, privileges and so on.
- **Bind**: In this stage we are able to get the actual values of any variables referenced in the statement
- **Execute**: In this stage we execute the statements in the SQL code.
- **Fetch**: In this stage the obtained results are returned to the user.

Some of the stages may not be relevant for all statements—for example; the fetch phase is applicable to queries but not DML. These stages can be appearing one by one or with one another. Every time the statement is parsed. Binding of values depends upon whether we are using the variables or not. Every time it is necessary to use the variables in SQL block. Execution stage happens every time. Fetch is also depend of the code of block, not necessary execute every time.

14.3 EXECUTION FLOW OF SQL IN PL/SQL SUBPROGRAMS

- When a SQL statement is included in a PL/SQL subprogram, the parse and bind phases are normally done at compile time, i.e. when the procedure, function or package body is CREATED.
- What if the text of the SQL statement is not known when the procedure is created? How could the Oracle server parse it?
  - It couldn't.
  - For example, suppose we want to DROP a table, but the user will enter the table name at execution time:

```
CREATE PROCEDURE drop_tabl (tbl_nam VARCHAR2)
IS BEGIN
   DROP TABLE tbl_nam; - - cannot be parsed
END;
/
```

14.4 DYNAMIC SQL:

In programming world the word Dynamic refers for runtime execution. The Dynamic SQL is used to write programs that mention SQL statements whose full text is not known until runtime. The complete query or the procedure is evaluated only at the run time, which gives results depending on the code. In this code some the code may refer the code which is precompiled or may refer

some code which is not part of current code. Instead the static SQL statements do not change from execution to execution. The full code of static SQL statements is known at compilation, which gives the following benefits:

- The total compilation checks that the necessary privileges are already given to access the database objects.
- The total compilation verifies that the SQL statements reference valid database objects.

The Static SQL execution is very much straight forward and self explanatory. The result and performance of static SQL is always good than dynamic SQL. Due to these advantages, we should use dynamic SQL only when we cannot use static SQL to achieve our results. The static SQL has limitations that can be avoided with dynamic SQL. Sometimes we may not know the complete code of the SQL statements that must be executed in a PL/SQL procedure. Our program may accept user input that defines the SQL statements to execute, or our program may need to complete some processing to determine the correct course of action. In such situation, use dynamic SQL.

For example, a duration calculating application in a data warehouse environment may not know the exact table name until runtime. These tables might be named according to the starting month and year of the quarter, for example INV_01_1997, INV_04_1997, INV_07_1997, INV_10_1997, INV_01_1998, and so on. We can use dynamic SQL in our duration calculating application to specify the table name at runtime.

When we want to run a complex query with a user-selectable sort order then we can use Dynamic SQL. Instead of coding the query twice, with different *ORDER BY* clauses, we can construct the query dynamically to include a specified *ORDER BY* clause.

14.5 DYNAMIC QUERIES EXECUTION:

While dealing with traditional database management system, there are several situations occur when we need to use dynamic SQL. We can use dynamic SQL to create applications that execute dynamic queries, whose full code is not known until runtime. Many types of programs need to use dynamic queries, including:

- The Code or Programs that allow users to input or choose query search or sorting criteria at runtime.
- The Code or Programs that allow users to input or choose optimizer hints at run time.
- The Code or Programs that query a database where the data definitions of tables are constantly changing.

- The Code or Programs that query a database where new tables are often created.

14.6 DYNAMICALLY EXECUTING A PL/SQL BLOCK :

There are multiple ways of executing normal PLSQL block. We can use the *EXECUTE IMMEDIATE* statement to execute anonymous PL/SQL blocks. We can add flexibility by constructing the block contents at runtime.

For example, suppose we want to write a program that takes an emp number and sends to a handler for the event. The name of the handler is in the form MY_EVENT_HANDLER_emp_number, where emp_number is the number of the employee. One approach is to implement the sender as a switch statement, where the code handles each event by making a static call to its appropriate handler. This code is not very extensible because the dispatcher code must be updated whenever a handler for a new employee is added.

```
CREATE OR REPLACE PROCEDURE my_emp_handler_1(e
number) AS BEGIN
  -- Code to process the event
  RETURN;
END;
/

CREATE OR REPLACE PROCEDURE my_emp_handler_2(e
number) AS BEGIN
  -- Code to process the event
  RETURN;
END;
/

CREATE OR REPLACE PROCEDURE my_emp_handler_3(e
number) AS BEGIN
  -- Code to process the event
  RETURN;
END;
/
```

```
SQL> CREATE OR REPLACE PROCEDURE
my_emp_adder (emp number, e number)
        IS
 2  BEGIN
 3    IF (emp = 1) THEN
 4      my_emp_handler_1 (e);
 5    ELSIF (emp = 2) THEN
```

```
 6     my_emp_handler_2(e);
 7    ELSIF (emp = 3) THEN
 8     my_emp_handler_3(e);
 9    END IF;
10  END;
11  /
```

Otherwise using native dynamic SQL, we can write a smaller, more flexible event dispatcher similar to the following:

```
SQL> CREATE OR REPLACE PROCEDURE my_emp_adder   (emp
NUMBER, e NUMBER)
  2          IS  BEGIN
 3    EXECUTE IMMEDIATE
 4      'BEGIN
 5        my_emp_handler_' || to_char(e) || '(:1);
 6      END;'
 7    USING e;
 8  END;
 9  /

Procedure created.
```

14.7 DYNAMIC SQL USING NATIVE DYNAMIC SQL

In this situation we are able to decide how to perform the following operations using native dynamic SQL:

- How and when to Execute DDL and DML operations.
- How and when to Execute single row and multiple row queries.

The database in this scenario is a company's human resources database (named **emp**) with the following data model:
A master table named offices contains the list of all company locations. The offices table has the following definition:

Column Name	Null?	Type
LOCATION	**NOT_NULL**	**VARCHAR2(200)**

Multiple emp_location tables contain the employee information, where location is the name of city where the office is located. For example, a table named emp_delhi contains employee information for the company's Delhi office, while a table named emp_nagpur contains employee information for the company's Nagpur office.

Each emp_location table has the following definition:

Column Name	Null?	Type
EMP_NO	NOT_NULL	NUMBER(4)
E_NAME	NOT_NULL	VARCHAR2(10)
JOB	NOT_NULL	VARCHAR2(9)
SAL	NOT_NULL	NUMBER(7,2)
DEPT_NO	NOT_NULL	NUMBER(2)

The following sections describe various native dynamic SQL operations that can be performed on the data in the employee database.

Note: *To execute following examples we need to create the proper database and table structure in advance otherwise the code will give compilation error.*

- **DML Operation Using Native Dynamic SQL**

The following native dynamic SQL procedure gives an increment to all employees with a particular job title:

```
SQL> CREATE OR REPLACE PROCEDURE
       salary_incr (incr_percent NUMBER, job VARCHAR2)
     IS
  2    TYPE loc_array_type IS
  3  TABLE OF VARCHAR2(40) INDEX BY binary_integer;
  4    dml_str VARCHAR2(200);
  5    loc_array   loc_array_type;
  6  BEGIN    -- fetch the list of office locations
  7    SELECT location BULK COLLECT INTO loc_array
  8            FROM offices;      -- for each location, give a raise to
    employees with the given 'job'
  9    FOR i IN loc_array.first..loc_array.last LOOP
 10       dml_str := 'UPDATE emp_' || loc_array(i)    || ' SET sal = sal
    * (1+(:incr_percent/100))'
 11        || ' WHERE job = :job_title';
 12    EXECUTE IMMEDIATE dml_str USING incr_percent, job;
 13    END LOOP
 14  END;
 15  /
SHOW ERRORS;      -- To view the syntax error if any
```

- **DDL Operation Using Native Dynamic SQL**

The EXECUTE IMMEDIATE statement can perform DDL operations. For example, the following procedure adds an office location.

```
SQL> CREATE OR REPLACE PROCEDURE add_loc (loc
VARCHAR2) IS
  2  BEGIN
  3      -- insert new location in master table
  4          INSERT INTO offices VALUES (loc);       -- create an
employee information table
  5      EXECUTE IMMEDIATE
  6      'CREATE TABLE ' || 'emp_' || loc ||
  7      '(
  8          empno   NUMBER(4) NOT NULL,
  9          ename   VARCHAR2(10),
 10          job     VARCHAR2(9),
 11          sal     NUMBER(7,2),
 12          deptno  NUMBER(2)
 13      )';
 14  END;
 15  /
```

The following procedure deletes an office location:

```
SQL> CREATE OR REPLACE PROCEDURE drop_loc (loc VARCHAR2)
IS
  2  BEGIN
  3      -- delete the employee table for location 'loc'
  4      EXECUTE IMMEDIATE 'DROP TABLE ' || 'emp_' || loc;    -- remove
           location from master table
  5      DELETE FROM offices WHERE location = loc;
  6  END;
  7  /
```

- **Single-Row Query Using Native Dynamic SQL**

The EXECUTE IMMEDIATE statement can perform dynamic single-row queries. We can specify bind variables in the USING clause and fetch the resulting row into the target specified in the INTO clause of the statement.

The following function retrieves the number of employees at a particular location performing a specified job:

```
SQL> CREATE OR REPLACE FUNCTION get_num_of_emp (loc
VARCHAR2, job VARCHAR2)
  2      RETURN NUMBER IS
  3      query_str VARCHAR2(1000);
  4      num_of_emp NUMBER;
  5  BEGIN
  6      query_str := 'SELECT COUNT(*) FROM '
  7          || ' emp_' || loc
  8          || ' WHERE job = :job_title';
  9      EXECUTE IMMEDIATE query_str
```

```
 10         INTO num_of_emp
 11         USING job;
 12      RETURN num_of_emp;
 13  END;
 14  /
```

- **Multiple-Row Query Using Native Dynamic SQL**

The OPEN-FOR, FETCH, and CLOSE statements can perform dynamic multiple-row queries. For example, the following procedure lists all of the employees with a particular job at a specified location

```
SQL> CREATE OR REPLACE PROCEDURE list_emp(loc VARCHAR2,
job VARCHAR2) IS
  2     TYPE cur_typ IS REF CURSOR;
  3     c   cur_typ;
  4     query_str   VARCHAR2(1000);
  5     emp_name    VARCHAR2(20);
  6     emp_num     NUMBER;
  7  BEGIN
  8     query_str := 'SELECT ename, empno FROM emp_' || loc
  9          || ' WHERE job = :job_title';     -- find employees who perform the
specified job
 10     OPEN c FOR query_str USING job;
 11     LOOP
 12        FETCH c INTO emp_name, emp_num;
 13        EXIT WHEN c%NOTFOUND;        -- processing goes here
 14     END LOOP;
 15     CLOSE c;
 16  END;
 17  /
```

14.8 USING DBMS_SQL PACKAGE

This is the standard SQL package provided by the oracle to Database programmer to bring flexibility in the coding. The DBMS_SQL package gives access to dynamic SQL and dynamic PL/SQL from within PL/SQL programs. "Dynamic" means that the SQL statements we execute with this package are not prewritten into our programs. They are, constructed at runtime as character strings and then passed to the SQL engine for execution.

The DBMS_SQL package provides an entity called a SQL cursor number. Because the SQL cursor number is a PL/SQL integer, we can pass it across call boundaries and store it. We can also use the SQL cursor number to obtain information about the SQL statement that we are executing.

We must use the DBMS_SQL package to execute a dynamic SQL statement when we don't know either of the following until run-time:

- SELECT list
- What placeholders in a SELECT or DML statement must be bound

In the following situations, we must use native dynamic SQL instead of the DBMS_SQL package:

- *The dynamic SQL statement retrieves rows into records.*

We want to use the SQL cursor attribute %FOUND, %ISOPEN, %NOTFOUND, or %ROWCOUNT after issuing a dynamic SQL statement that is an INSERT, UPDATE, DELETE, or single-row SELECT statement.

When we need both the DBMS_SQL package and native dynamic SQL, we can switch between them, using the following:

- DBMS_SQL.TO_REFCURSOR Function
- DBMS_SQL.TO_CURSOR_NUMBER Function

Note:
We can invoke DBMS_SQL subprograms remotely which improves ease of coding and realization of the result from any location.

- **When to use Native Dynamic SQL and the DBMS_SQL Package:**

The Oracle provides two methods for using dynamic SQL within PL/SQL: native dynamic SQL and the DBMS_SQL package. Native dynamic SQL allows us to place dynamic SQL statements directly into PL/SQL code. These dynamic statements include DML statements (including queries), PL/SQL anonymous blocks, DDL statements, transaction control statements, and session control statements.

To process most native dynamic SQL statements, we use the EXECUTE IMMEDIATE statement. To process a SELECT statement, use OPEN-FOR, FETCH, and CLOSE statements.

The DBMS_SQL package is a PL/SQL library that offers an API (Application programming Interface) to execute SQL statements dynamically. It has procedures to open a cursor, parse a cursor; supply binds, and so on. Programs that use the DBMS_SQL package make calls to this package to perform dynamic SQL operations.

14.9 ADVANTAGES OF NATIVE DYNAMIC SQL

Native dynamic SQL has following advantages over the DBMS_SQL package:

- **Very easy to Use**

Because native dynamic SQL is integrated with SQL, we can use it in the same way that we use static SQL within PL/SQL code. Native dynamic SQL code is typically more compact and readable than equivalent code that uses the DBMS_SQL package.

With the DBMS_SQL package we must call many procedures and functions in a strict sequence, making even simple operations require a lot of code. We can avoid this complexity by using native dynamic SQL instead.

Following table demonstrates the difference in the amount of code required to perform the same operation using the DBMS_SQL package and native dynamic SQL.

Example of Code Comparison of DBMS_SQL Package and Native Dynamic SQL

1 Code of DBMS_SQL Package

```
SQL> CREATE PROCEDURE insertion ( tname  VARCHAR2,
dno  NUMBER,  dname  VARCHAR2,
  2       location    VARCHAR2) IS
  3    mycur        INTEGER;   stmt_str       VARCHAR2(200);
  4    rows_processed  BINARY_INTEGER;
  5
  6  BEGIN
  7 stmt_str := 'INSERT INTO ' || tname || ' VALUES   (:deptno,
:dname, :loc)';
  8
  9    mycur := dbms_sql.open_cursor; -- opening cursor
 10     -- parse cursor
 11     dbms_sql.parse(mycur, stmt_str,  dbms_sql.native);
 12    -- supply binds
 13     dbms_sql.bind_variable    (mycur, ':deptno', dno);
 14     dbms_sql.bind_variable      (mycur, ':dname', dname);
 15     dbms_sql.bind_variable      (mycur, ':loc', location);
 16     -- execute cursor
 17      rows_processed :=     dbms_sql.execute(mycur);
 18      -- close cursor
 19      dbms_sql.close_cursor(mycur);
 20  END;
 21  /
```

2. Code of Native Dynamic SQL

```
SQL> CREATE PROCEDURE insertion1
    (  tname  VARCHAR2, dno NUMBER,  dname VARCHAR2,
  2       location    VARCHAR2) IS
  3    stmt_str    VARCHAR2(200);
  4
  5  BEGIN
  6    stmt_str := 'INSERT INTO ' ||     tname || ' values     (:deptno,
:dname, :loc)';
  7
  8    EXECUTE IMMEDIATE stmt_str
  9       USING
 10       dno, dname, location;
 11
 12  END;
 13  /

Procedure created.

SQL> SHOW ERRORS;
No errors.
```

14.10 EXECUTION SPEED OF NATIVE DYNAMIC SQL IS FASTER THAN DBMS_SQL:

The Native dynamic SQL in PL/SQL performs comparably to the performance of static SQL, because the PL/SQL interpreter has built-in support for it. Programs that use native dynamic SQL are much faster than programs that use the DBMS_SQL package. Typically, native dynamic SQL statements perform 1.5 to 3 times better than equivalent DBMS_SQL calls. Sometimes our performance gains may vary depending on our application.

Native dynamic SQL bundles the statement preparation, binding, and execution steps into a single operation, which minimizes the data copying and procedure call overhead and improves performance.

The DBMS_SQL package is based on a procedural API and incurs high procedure call and data copy overhead. Each time we bind a variable, the DBMS_SQL package copies the PL/SQL bind variable into its space used during execution. Each time we execute a fetch, the data is copied into the space managed by the DBMS_SQL package and then the fetched data is copied, one column at a time, into the appropriate PL/SQL variables, resulting in substantial overhead.

14.11 ADVANTAGES OF THE DBMS_SQL PACKAGE

The DBMS_SQL package provides the following advantages over native dynamic SQL:

1. We can Reuse SQL Statements using DBMS_SQL

The PARSE procedure in the DBMS_SQL package parses a SQL statement once. After the initial parsing, we can use the statement multiple times with different sets of bind arguments.

Native dynamic SQL prepares a SQL statement each time the statement is used, which typically involves parsing, optimization, and plan generation. Although the extra prepare operations incur a small performance penalty, the slowdown is typically outweighed by the performance benefits of native dynamic SQL.

2. Client-Side Program supports DBMS_SQL Package:

The DBMS_SQL package is supported in client-side programs, but native dynamic SQL is not. Every call to the DBMS_SQL package from the client-side program translates to a PL/SQL remote procedure call (RPC); these calls occur when we need to bind a variable, define a variable, or execute a statement.

3. Supports Multiple Row Updates and Deletes with a RETURNING Clause

The DBMS_SQL package supports statements with a RETURNING clause that update or delete multiple rows. Native dynamic SQL only supports a RETURNING clause if a single row is returned.

4. It Supports DESCRIBE

The DESCRIBE_COLUMNS procedure in the DBMS_SQL package can be used to describe the columns for a cursor opened and parsed through DBMS_SQL. This feature is similar to the DESCRIBE command in SQL*Plus. Native dynamic SQL does not have a DESCRIBE facility.

5. Supports SQL Statements Larger than 32KB

The DBMS_SQL package supports SQL statements larger than 32KB; native dynamic SQL does not.

14.12 PERFORMING DML USING DYNAMIC SQL:

The following example includes a dynamic INSERT statement for a table with three columns:

query := 'INSERT INTO dept_new VALUES (:dept_no, :dept_name, :loc)';

This example inserts a new row for which the column values are in the PL/SQL variables dept_no, dept_name, and location. Table shows sample code that accomplishes this DML operation using the DBMS_SQL package and native dynamic SQL.

Table DML Operation Using the DBMS_SQL Package and Native Dynamic SQL

1. **Code of DBMS_SQL DML Operation**

```
SQL> DECLARE
  2    stmt_str VARCHAR2(350);  my_cur NUMBER;
  3    deptid NUMBER := 101;  deptname VARCHAR2(20);
  4    location VARCHAR2(20);myresources VARCHAR2(20);
       rows_processed NUMBER;
  5  BEGIN
  6    stmt_str := 'INSERT INTO departments VALUES(:did,
:dname,

         :location,:resources)';
  7    my_cur := DBMS_SQL.OPEN_CURSOR;
  8    DBMS_SQL.PARSE(my_cur, stmt_str,
DBMS_SQL.NATIVE);
  9      -- supply binds
 10    DBMS_SQL.BIND_VARIABLE    (my_cur, ':did',
deptid);
 11    DBMS_SQL.BIND_VARIABLE    (my_cur, ':dname',
deptname);
 12    DBMS_SQL.BIND_VARIABLE    (my_cur, ':location',
location);
 13    DBMS_SQL.BIND_VARIABLE    (my_cur, ':resources',
myresources);
 14
 15      rows_processed := dbms_sql.execute(my_cur);
 16      -- execute
 17    DBMS_SQL.CLOSE_CURSOR(my_cur); -- close
 18  END;
 19 /
```

2. **Code of Native Dynamic SQL DML Operation:**

```
SQL> DECLARE
  2    stmt_str VARCHAR2(350);   deptid NUMBER := 102;
  3    deptname VARCHAR2(20);   location VARCHAR2(20);
4     myresource  VARCHAR2(20);

  5  BEGIN
```

```
 6    stmt_str := 'INSERT INTO departments VALUES
 7         (:did, :dname, :location,:resources)';
 8  EXECUTE IMMEDIATE stmt_str
 9      USING deptid, deptname, location ,myresource;
10  END;
11  /
```

14.13 USE OF DYNAMIC SQL DIFFERENT LANGUAGES:

The dynamic SQL is also supported in various database languages with their language specifications. We can call dynamic SQL from other languages as:

- If we use C/C++, we can call dynamic SQL with the Oracle Call Interface (OCI), or we can use the Pro*C/C++ pre-compiler to add dynamic SQL extensions to our C code.
- If we use COBOL, we can use the Pro*COBOL pre-compiler to add dynamic SQL extensions to our COBOL code.
- If we use Java, we can develop applications that use dynamic SQL with JDBC.

If we have an application that uses OCI, Pro*C/C++, or Pro*COBOL to execute dynamic SQL, we should consider switching to native dynamic SQL inside PL/SQL stored procedures and functions. The network round-trips required to perform dynamic SQL operations from client-side applications might decrease performance. Stored procedures can reside on the server, eliminating the network overhead. We can call the PL/SQL stored procedures and stored functions from the OCI, Pro*C/C++, or Pro*COBOL application.

14.14 QUESTIONS

1. State the execution flow of SQL in PL/SQL Subprograms.
2. How to execute PL/SQL Block Dynamically?
3. Write short note on Dynamic SQL.
4. How to execute Dynamic queries?
5. What is Native Dynamic SQL?
6. Write short note on DBMS_SQL Package.
7. State the Advantages of Native Dynamic SQL.
8. State the Advantages of DBMS_SQL Package.
9. Where we can use Dynamic SQL other than PLSQL?

Practice Questions:

10. Write a Simple example for DML Operation Using Native Dynamic SQL.
11. Write a Simple example for DDL Operation Using Native Dynamic SQL.
12. Write a Simple example for Multiple-Row Query Using Native Dynamic SQL.
13. Use the DBMS_SQL package in the above examples.

14.15 FURTHER READING

- Murach's Oracle SQL and PLSQL by Joel Murach, Murach and Associates.
- Oracle Database 11g PL/SQL Programming Workbook, ISBN: 9780070702264,

  By: Michael McLaughlin,John Harper, TATAMCGRAW-HILL
- Oracle PL/SQL Programming, Fifth Edition By Steven Feuerstein, Bill Pribyl
- Oracle 11g: SQL Reference Oracle press
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, TATAMCGRAW-HILL
- Oracle database 11g: hands on SQL/PL SQL by Satish Asnani (PHI) EEE edition

15

TRIGGERS

Unit Structure

15.1 OBJECTIVES

After completing this chapter, you will be able to:

- Understand the Fundamentals of Triggers
- Create and use Triggers

- ❖ Understand the types of Triggers
- ❖ Understand the Execution Hierarchy of PL/SQL Trigger
- ❖ Creating DML and DDL Triggers
- ❖ View , Alter and Drop the Triggers
- ❖ Understand the Cyclic Cascading in a Triggers
- ❖ Understand the Privileges Required to Use Triggers

15.2 DEFINING A TRIGGER:

The triggers play an important role while validating the SQL and PLSQL queries on automatic basis depending on the particular condition. It executes or fired like a definite event every time. A trigger can be defined as automatic code execution on a particular event of a database. A trigger is a PL/SQL block structure or a subprogram which is fired or executed when a DML statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically at predefined timing and event, when an associated DML statement is executed. Triggers are physically stored in database.

A trigger stored in the database can include SQL and PL/SQL or Java statements to run as a unit and can invoke stored procedures. However, procedures and triggers differ in the way that they are invoked. A procedure is explicitly run by a user, application, or trigger. Triggers are implicitly fired by Oracle when a triggering event occurs, no matter which user is connected or which application is being used. Triggers are useful in achieving security and auditing. It also maintains data integrity and referential integrity.

15.3 INSIDE THE TRIGGERS

The triggers give the dynamic approach to our SQL script. Before we create and use the trigger, the user SYS must run a SQL script commonly called DBMSSTDX.SQL. The proper name and location of this script depend on our operating system. Before starting with the triggers we must consider following points.

- We must have the CREATE TRIGGER system privilege, to create a trigger in our own schema on a table.
- To create a trigger in any schema on a table in any schema, or on another user's schema (*schema*. SCHEMA), we must have the CREATE ANY TRIGGER system privilege.
- To create a trigger on DATABASE, we must have the ADMINISTER DATABASE TRIGGER system privilege.

If the trigger issues SQL statements or calls procedures or functions, then the owner of the trigger must have the privileges necessary to perform these operations. These privileges must be granted directly to the owner rather than acquired through roles.
In short we must have maximum privileges to deal with the triggers.

15.4 THE DATABASE TRIGGERS & APPLICATION TRIGGERS:

There are various types of triggers. Triggers can be categorized as Database triggers and Application triggers on the basis of scope of usage of triggers. The Database triggers are activated on any event occurring in the database, while application trigger are restricted to an application. Database triggers can be created on top of table, view, schema or database. Timings for table or view can be before and after a DML operation, while those on schema and database
can be logging in and log off.

A. Triggers with DML:

The following are the three criteria which must be kept in mind before creating and using the DML trigger.

1. There can be three possible DML actions on data i.e. INSERT, UPDATE or DELETE. These are events for the DML triggers.
2. A simultaneous action can be performed either before or after an event. This serves as timing for DML triggers.
3. Whether the trigger action must be at DML statement level or at affected row level, decides the level of a trigger.

   After the timing, event and level are set, trigger body must be created to implement the triggering logic.

Important Note: *The size of a trigger cannot be greater than 32 KB.*

Syntax of Triggers:
The Syntax for creating a trigger is:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF}
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF column_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
```

```
[FOR EACH ROW]
WHEN (condition)
BEGIN
--- The SQL Code // application logic goes here
END;
/
```

Understanding the Syntax:

- ***CREATE [OR REPLACE] TRIGGER trigger_name*** *:-* This line creates a trigger with the given name or overwrites an existing trigger with the same name. It is a compulsory part of syntax.
- ***{BEFORE | AFTER | INSTEAD OF }*** - This line indicates at what time the trigger should get fired. i.e. For example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. Before and After cannot be used to create a trigger on a view.
- ***{INSERT [OR] | UPDATE [OR] | DELETE}*** - This line determines the triggering event. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.
- ***[OF col_name]*** - This statement is used with update triggers. This statement is used when we want to trigger an event only when a specific column is updated.
- ***[ON table_name]*** - This statement identifies the name of the table or view to which the trigger is associated.
- ***[REFERENCING OLD AS o NEW AS n]*** - This statement is used to reference the old and new values of the data being changed. By default, we reference the values as :old.column_name or :new.column_name. The reference names can also be changed from old (or new) to any other user-defined name. We cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.
- ***[FOR EACH ROW]*** - This statement is used to determine whether a trigger must fire when each row gets affected (i.e. a Row Level Trigger) or just once when the entire SQL statement is executed(i.e.statement level Trigger).
- ***WHEN (condition)*** - This statement is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

For Example: If we want to avoid the duplicate entry of the field other than primary field then we can create trigger to check that particular entry. The price of a product changes constantly. It is important to maintain the history of the prices of the products. We

can create a trigger to update the 'price_trace' table when the price of the product is updated in the 'product' table.

1) Create the 'product' table and 'price_trace ' table

```
SQL> CREATE TABLE price_trace
16    (product_id number(5), product_name varchar2(32),
17     supplier_name varchar2(32),
4  unit_price number(7,2) );

Table created.

SQL> CREATE TABLE product (product_id number(5),
product_name varchar2(32),
                supplier_name varchar2(32),  unit_price number(7,2)
        );

Table created.

SQL> insert into product values(1312,'Wooden_Door','Galaxy',950);

1 row created.

SQL> insert into product values(1313,'Plastic_Door','Tanmay',1950);

1 row created.

SQL> insert into product values(1314,'Metal_Door','Sun',11450);

1 row created.
```

2) Create the my_price_trace trigger and execute it.

```
SQL> CREATE or REPLACE TRIGGER  my_price_trace
  2  BEFORE UPDATE OF unit_price
  3  ON product
  4  FOR EACH ROW
  5  BEGIN
  6  INSERT INTO price_trace
  7             VALUES      (:old.product_id,      :old.product_name,
:old.supplier_name, :old.unit_price);
  8  END;
  9  /

Trigger created.
```

3) Lets update the price of a product.

```
        SQL> UPDATE product SET unit_price = 900 WHERE
product_id = 1312;
        1 row updated.
```

Once the above update query is executed, the trigger fires and updates the 'price_trace' table. We can view the result using following statements.

SQL> select * from price_trace;

PRODUCT_ID	PRODUCT_NAME	SUPPLIER_NAME	UNIT_PRICE
1312	Wooden_Door	Galaxy	950

4) If we **ROLLBACK** the transaction before committing to the database, the data inserted to the table is also rolled back.

15.5 CLASSIFICATION OF PL/SQL TRIGGERS:

There are two types of triggers based on the level it is triggered.

1) Trigger on Row level: - The Row level trigger fires automatically when any other query makes any change in the any single table row. The row level trigger is fired each time the table is affected by the triggering statement. For example, if an UPDATE statement updates multiple rows of a table, a row trigger is fired once for each row affected by the UPDATE statement. If a triggering statement affects no rows, a row trigger is not executed at all. Row triggers are useful if the code in the trigger action depends on data provided by the triggering statement or rows that are affected.

2) Trigger on Statement level: - The statement trigger is fired once on behalf of the triggering Statement, regardless of the number of rows in the table that the triggering statement affects (even if no rows are affected). For example, if a DELETE statement deletes several rows from a table, a statement-level DELETE trigger is fired only once, regardless of how many rows are deleted from the table.

Statement triggers are useful if the code in the trigger action does not depend on the data provided by the triggering statement or the rows affected. For example, if a trigger makes a complex security check on the current time or user, or if a trigger generates a single audit record based on the type of triggering statement, a statement trigger is used.

15.6 DIFFERENCE BETWEEN BEFORE & AFTER TRIGGERS:

The Before and After triggers are related with the timing of firing them or we can say that when to execute them. When defining a trigger, we can specify the trigger timing. Means, we can

specify whether the trigger action is to be executed before or after the triggering statement. BEFORE and AFTER apply to both statement and row triggers.

1) BEFORE Triggers: The BEFORE triggers executes the trigger action before the triggering statement. The BEFORE triggers are used when the trigger action should determine whether the triggering statement should be allowed to complete. By using a BEFORE trigger for this purpose, we can eliminate unnecessary processing of the triggering statement and its ultimate rollback in cases where an exception is raised in the trigger action. BEFORE triggers are also used to derive specific column values before completing a triggering INSERT or UPDATE statement.

2) AFTER Triggers: The AFTER triggers execute the trigger action after the triggering statement is executed. The AFTER triggers are used when we want the triggering statement to complete before executing the trigger action. If a BEFORE trigger is already present, an AFTER trigger can perform different actions on the same triggering statement.

We can have multiple triggers of the same type for the same statement for any given table. For example we may have two *BEFORE STATEMENT* triggers for *UPDATE* statements on the *EMPLOYEE* table. Multiple triggers of the same type permit modular installation of applications that have triggers on the same tables. We can design our own *AFTER ROW* trigger in addition to the Oracle-defined *AFTER ROW* trigger.

We can create as many triggers of the preceding different types as we need for each type of DML statement (INSERT, UPDATE, or DELETE). For example, suppose we have a table Payment, and we want to know when the table is being accessed and the types of queries being issued.

15.7 EXECUTION SEQUENCE OF PL/SQL TRIGGER:

There are some rules of execution of triggers. The following sequence is followed when a trigger is fired.
1) The BEFORE statement trigger executes / fires first.
2) Next BEFORE row level trigger fires, once for each row affected.
3) Then AFTER row level trigger fires once for each affected row. These events will alternates between BEFORE and AFTER row level triggers.

4) Finally the AFTER statement level trigger fires.

For example let’s create a table 'product_chk' which we can use to store messages when triggers are fired.

```
SQL> CREATE TABLE product_chk (Message varchar2(50),
Current_Date date  );
Table created.
```

Let's create a BEFORE and AFTER statement and row level triggers for the product table.

1) BEFORE UPDATE, Statement Level: This trigger will insert a record into the table 'product_chk' before a SQL update statement is executed, at the statement level.

```
SQL>  CREATE  or  REPLACE  TRIGGER
Before_Update_product
  2  BEFORE
  3  UPDATE ON product
  4  Begin
  5  INSERT INTO product_chk
  6  Values('Before update, statement level trigger', sysdate);
  7  END;
  8  /

Trigger created.
```

2**) BEFORE UPDATE, Row Level:** This trigger will insert a record into the table 'product_chk' before each row is updated.

```
SQL>  CREATE  or  REPLACE  TRIGGER
Before_Upddate_Row_product
  2   BEFORE
  3   UPDATE ON product
  4   FOR EACH ROW
  5   BEGIN
  6   INSERT INTO product_chk
  7   Values('Before update row level trigger',sysdate);
  8   END;
  9   /

Trigger created.
```

3) AFTER UPDATE, Statement Level: This trigger will insert a record into the table 'product_chk' after a SQL update statement is executed, at the statement level.

```
SQL>  CREATE or REPLACE TRIGGER
After_Update_product
  2   AFTER
```

```
  3  UPDATE ON product
  4  BEGIN
  5  INSERT INTO product_chk
  6  Values('After update, statement level trigger', sysdate);
  7  End;
  8  /
```

Trigger created.

4) AFTER UPDATE, Row Level: This trigger will insert a record into the table 'product_chk' after each row is updated.

```
SQL> CREATE or REPLACE TRIGGER
After_Update_Row_product
  2  AFTER
  3  insert On product
  4  FOR EACH ROW
  5  BEGIN
  6  INSERT INTO product_chk
  7  Values('After update, Row level trigger',sysdate);
  8  END;
  9  /

Trigger created.
```

Now let's execute a update statement on table item.

```
SQL> UPDATE product SET unit_price = 850
  2  WHERE product_id in (1312,1314);

2 rows updated.
```

We can check the data in 'product_chk' table to see the order in which the trigger is fired.

```
SQL>  SELECT * FROM product_chk;
```

Output:

```
MESSAGE                                          CURRENT_DATE
------------------------------------------------ --------------------
Before update, statement level                     29-AUG-12
Before update row level trigger                   29- AUG -12
Before update row level trigger                   29- AUG -12
After update, statement level triggers            29- AUG -12
```

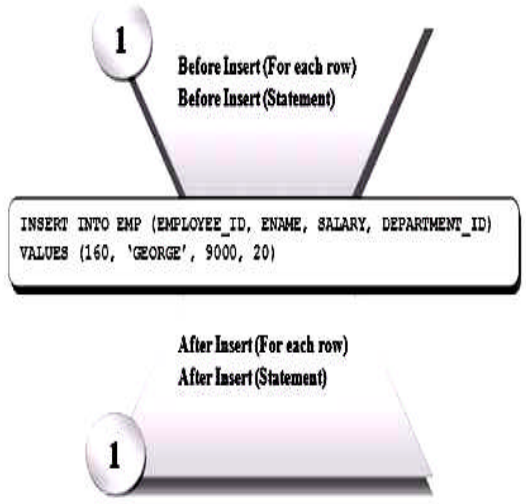
The above result shows 'before update' and 'after update' row level events have occurred twice, since two records were updated. But 'before update' and 'after update' statement level events are fired only once per SQL statement.

The above rules apply similarly for INSERT and DELETE statements.

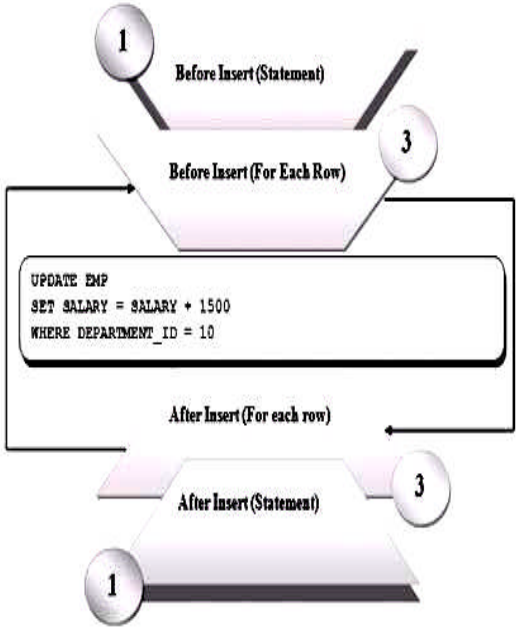
15.8 DIFFERENCE BETWEEN STATEMENT LEVEL AND ROW LEVEL TRIGGERS :

A) Statement-Level Triggers: These are the default triggers. These are fired only once when the triggering event occurs. These triggers does not have any effect that any row affected or not by the update or insert statement.

B) Row-Level Triggers: To fire these types of triggers FOR EACH ROW specification is required. These triggers are fired only when the rows affected by an event. If no rows are affected, it will not fire.



Trigger-Firing Sequence: Single-Row Manipulation



Trigger-Firing Sequence: Multi-Row Manipulation

(IMG Ref www.club-oracle.com)

Creating a DML Statement Trigger

we can create a statement level trigger as below. The trigger fires before the INSERT action on EMPLOYEE table. As per the trigger action, it inserts a record in Employee Log table, with current date, action and remarks.

```
SQL> create table employee(empid int,empname char(20),
  2         empsal number(7,2), jobtitle char(100));

Table created.
```

```
SQL> create table trace_emp(empid int,status char(2),
  2          actdate date,act char(20),remark char(100));

Table created.

SQL> CREATE OR REPLACE TRIGGER trace_employee
  2  BEFORE INSERT ON employee
  3  BEGIN
  4  INSERT INTO  trace_emp(empid, status, actdate, act, remark)
  5    VALUES(2, 'P', SYSDATE, 'CREATE', 'you are with the
triggers');
  6  END;
  7  /

Trigger created.

SQL> INSERT INTO EMPLOYEE (empid, empname, empsal,
jobtitle)
  2  VALUES(4, 'SONALI', 34500, 'manager');

1 row created.


SQL> SELECT * FROM trace_emp;

    EMPID     ST       ACTDATE        ACT          REMARK
 ----------  ------  ----------------  ----------  ----------------------------------------
       2          P        29-AUG-12     CREATE    you are with the triggers
```

Conditional Predicates to Detect the Trigger DML operation:

We can use conditional predicate with the DML trigger operations. For a specific timing, if all the events have to be tested instead of creating three different DML triggers, oracle provides DML predicates to be used in trigger body. The available predicates can be INSERTING, UPDATING or DELETING. For example, below trigger body shows the usage of DML predicates. Note the event specification and handling.

```
CREATE OR REPLACE TRIGGER my_trigg_trace_ emp
BEFORE INSERT OR UPDATE OR DELETE ON employee
BEGIN

IF INSERTING THEN
…
ELSIF UPDATING THEN
…
ELSIF DELETING
…
END IF;
```

```
END;
/
```

Output>>TRIGGER created.

15.9 BUILDING A DML ROW LEVEL TRIGGER

The DML row level trigger EMPLOYEE_MEMBERSHIP deletes the membership record for every employee record which gets deleted.

```
SQL> CREATE OR REPLACE TRIGGER EMPLOYEE_MEMBERSHIP
  2  BEFORE DELETE ON EMPLOYEE
  3  BEGIN
  4  FOR EACH ROW
  5   DELETE FROM EMP_MEMBERSHIP
  6   WHERE EMPID=:OLD.EMPID;
  7  END;
  8   /
```

Output>>TRIGGER created.

15.10 DDL TRIGGER CREATION:

Following example creates an AFTER statement trigger on any DDL statement CREATE. Such a trigger can be used to audit the creation of new data dictionary objects in our schema.

```
CREATE TRIGGER audit_db_field AFTER CREATE
  ON SCHEMA pl/sql_block
```

15.11 CALLING A PROCEDURE IN A TRIGGER BODY:

In following example we could create the check_salary trigger described in the preceding example by calling a procedure instead of providing the trigger body in a PL/SQL block. Assume we have defined a procedure check_salary in the hr schema, which verifies that an employee's salary is in an appropriate range. Then we could create the trigger check_salary as follows:

```
CREATE TRIGGER check_salary
BEFORE INSERT OR UPDATE OF empsal, jobtitle ON employees
  FOR EACH ROW
  WHEN (new. jobtitle <> 'DEVELOPER')
  CALL check_salary(:new.jobtitle, :new.empsal, :new.empname)
```

The procedure check_salary could be implemented in PL/SQL, C, or Java. Also, we can specify: OLD values in the CALL clause instead of: NEW values.

15.12 BUILDING A DATABASE EVENT TRIGGER:

Following example demonstrates the basic syntax for a trigger to log all errors. The hypothetical PL/SQL block does some special processing for a particular error (invalid logon, error number 1017). This trigger is an AFTER statement trigger, so it is fired after an unsuccessful statement execution, such as unsuccessful logon.

```
CREATE TRIGGER errlog AFTER SERVERERROR ON
DATABASE
  BEGIN
    IF (IS_SERVERERROR (1017)) THEN
      <special processing of logon error>
    ELSE
      <log error number>
    END IF;
  END;
/
```

15.13 CREATION OF A SCHEMA TRIGGER:

Following example creates a BEFORE statement trigger on the sample schema hr. When a user connected as hr attempts to drop a database object, the database fires the trigger before dropping the object.

```
CREATE OR REPLACE TRIGGER drop_resist_trigger
  BEFORE DROP ON hr.SCHEMA
  BEGIN
    RAISE_APPLICATION_ERROR ( num => -20000,  msg =>
'not able to drop object');
  END;
/
```

15.14 IDENTIFIERS (OLD AND NEW)

The OLD and NEW are identifiers which carry a record value before and after the DML event. The record values can be referred by prefixing a column value with the corresponding identifier. Below table shows the OLD and NEW values within each triggering event.

Event	OLD value	NEW value
INSERT	NULL	Current value

UPDATE	Old value of record	New value of record
DELETE	Record value before delete operation	NULL

Example:
The trigger below archives an employee record if salary is incremented by more than 2000. Note that the increment is checked by the WHEN clause condition.

```
SQL> CREATE OR REPLACE sal_incr
  2  BEFORE UPDATE OF empsal ON employee
  3  FOR EACH ROW
  4  WHEN(OLD.empsal  -  NEW.empsal > 2000)
  5  BEGIN
  6  INSERT INTO EMP_ARCHIVE (id, empid, OLD_SAL,
NEW_SAL, REVISED_DT)
      VALUES
  7  (SQ_ARC.NEXTVAL, :OLD.EMPID,:OLD.SALARY,
:NEW.SALARY, SYSDATE);
  8 END;
  9 /
```

15.15 INSTEAD OF TRIGGERS (VIEW TRIGGERS):

While database programming sometimes there may be situation that the triggers has two options and it must fired with alternate options. The INSTEAD OF trigger satisfy the condition. Triggers on views are known as INSTEAD OF triggers. They are known by their name because they skip the current triggering event action and perform alternate one. Other reason could be that only timing mode available in such triggers is INSTEAD OF. It is required for the complex view because it is based on more than one table. Any DML on complex view would be successful only if all key columns, not null columns are selected in the view definition. Alternatively, INSTEAD OF trigger can be created to synchronize the effect of DML across all the tables.

Instead of trigger is a row level trigger and can be used only with a view, and not with tables. For

Example, following view ORD_VU is created on top of ORDERS and WAREHOUSE tables. If
RET_LIMIT is attempted for update in the view, a record must be added to WAREHOUSE_HISTORY table and new value must be updated in the WAREHOUSE table.

```
SELECT O.ID, O.QTY, O.ORD_DATE, P.SITE_ID,
P.RET_LIMIT
FROM ORDERS O, WAREHOUSE P
WHERE O.SITE_ID=P.SITE_ID
CREATE OR REPLACE TRIGGER T_UPD_ORDERVIEW
INSTEAD OF UPDATE ON ORD_VU
BEGIN
  INSERT INTO WAREHOUSE_HISTORY
  (SITE_ID, OLD_RET_LIMIT, NEW_RET_LIMIT,
UPD_DATE, UPD_USER)
  VALUES
   (:OLD.SITE_ID, :OLD_RET_LIMIT, :NEW.RET_LIMIT,
SYSDATE, USER);
   UPDATE WAREHOUSE
  SET RET_LIMIT = :NEW.RET_LIMIT
  WHERE SITE_ID = :OLD.SITE_ID;
END;
/
```

15.16 LISTING OF TRIGGERS INFORMATION:

We can see all our user defined triggers by doing a select statement on USER_TRIGGERS. This will gives us clear idea about the trigger and its structure.
For example:

```
SELECT TRIGGER_NAME FROM USER_TRIGGERS;
```

Above statement produces the names of all triggers. We can also select more columns to get more detailed trigger information. We can do that at our own relaxation, and explore it on our own.

15.17 ALTERING A TRIGGER:

There is facility to change the trigger code and conditions. If a trigger seems to be getting in the way, and we don't want to drop it, just disable it for a little while, we can alter it to disable it. Note that this is not the same as dropping a trigger; after we drop a trigger, it is gone.

The general format of an alter would be something like this:

```
ALTER TRIGGER trigger_name [ENABLE|DISABLE];
```

For example, let's say that with all our troubles, we still need to modify the DOB of 'SONALI SAMBARE '. We cannot do this since we have a trigger on that table that prevents just that. So, we can disable it...

```
ALTER TRIGGER PERSON_DOB DISABLE;
```

Now, we can go ahead and modify the DOB :-)

```
UPDATE PERSON SET DOB = SYSDATE WHERE NAME = 'YASHASHREE SAMBARE';
```

We can then re-ENABLE the trigger.

```
ALTER TRIGGER PERSON_DOB ENABLE;
```

If we then try to do the same type of modification, the trigger kicks and prevents us from modifying the DOB.

- **Syntax for removing Triggers:**

For removing the trigger we have to use following syntax.

```
DROP TRIGGER trigger_name;
```

15.18 KNOWING INFORMATION ABOUT TRIGGERS:

We can use the data dictionary view 'USER_TRIGGERS' to obtain information about any trigger. The below statement shows the structure of the view 'USER_TRIGGERS'.

```
DESC USER_TRIGGERS;
```

```
NAME                                    Type
----------------------------------------------------------
TRIGGER_NAME                            VARCHAR2(30)
TRIGGER_TYPE                            VARCHAR2(16)
TRIGGER_EVENT                           VARCHAR2(75)
TABLE_OWNER                             VARCHAR2(30)
BASE_OBJECT_TYPE                        VARCHAR2(16)
TABLE_NAME                              VARCHAR2(30)
COLUMN_NAME                             VARCHAR2(4000)
REFERENCING_NAMES                       VARCHAR2(128)
WHEN_CLAUSE                             VARCHAR2(4000)
STATUS                                  VARCHAR2(8)
DESCRIPTION                             VARCHAR2(4000)
ACTION_TYPE                             VARCHAR2(11)
TRIGGER_BODY                            LONG
```

This view stores information about header and body of the trigger.

```
SELECT * FROM user_triggers WHERE trigger_name = 'Before_Update_product';
```

The above SQL query provides the header and body of the trigger 'Before_Update_Stat_product'.
We can drop a trigger using the following command.

DROP TRIGGER trigger_name;

15.19 CYCLIC CASCADING in a TRIGGER:

Sometimes the triggers may create some critical situation. This is an undesirable situation where more than one trigger enters into an infinite loop. While creating a trigger we should ensure the situation does not exist.

Let's consider we have two tables 'inv' and 'product'. Two triggers are created.

1) The INSERT Trigger, triggerA on table 'inv' issues an UPDATE on table 'product'.

2) The UPDATE Trigger, triggerB on table 'product' issues an INSERT on table 'inv'.

In such a situation, when there is a row inserted in table 'inv', triggerA fires and will update table 'product'. When the table 'product' is updated, triggerB fires and will insert a row in table 'inv'. This cyclic situation continues and will enter into a infinite loop, which will crash the database.

15.20 BOUNDARIES ON TRIGGER CONDITIONS:

Trigger conditions are subject to the following restrictions:

1) If we specify this clause for a DML event trigger, then we must also specify FOR EACH ROW. Oracle Database evaluates this condition for each row affected by the triggering statement.

2) We cannot specify trigger conditions for INSTEAD OF trigger statements.

3) We can reference object columns or their attributes, or varray, nested table, or LOB columns. We cannot invoke PL/SQL functions or methods in the trigger condition.

15.21 TRIGGER EXCEPTIONS:

Triggers become part of the transaction of a statement, which implies that it causes (or raises) any exceptions, the whole statement is rolled back.

Think of an exception as a flag that is raised when an error occurs. Sometimes, an error or exception is raised for a valid reason. For example, to prevent some action that improperly modifies the database. Let's say that our database should not allow anyone to modify their DOB (after the person is in the database, their DOB is assumed to be static). Anyway, we can create a trigger that would prevent us from updating the DOB:

```
CREATE OR REPLACE
TRIGGER change_resist_id
BEFORE UPDATE OF empid ON employee
FOR EACH ROW
BEGIN
RAISE_APPLICATION_ERROR (-20000,'CANNOT
CHANGE DATE OF BIRTH');
END;
/
```

Notice the format of the trigger declaration. We explicitly specify that it will be called BEFORE UPDATE OF DOB ON PERSON. The next thing we should notice is the procedure call RAISE APPLICATION ERROR, which accepts an error code, and an explanation string. This effectively halts our trigger execution, and raises an error, preventing our DOB from being modified. An error (exception) in a trigger stops the code from updating the DOB. When we do the actual update for example

UPDATE PERSON SET DOB = SYSDATE;

We end up with an error, which says we CANNOT CHANGE DATE OF BIRTH.

UPDATE PERSON SET DOB = SYSDATE;

```
UPDATE PERSON SET DOB = SYSDATE
*
ERROR at line 1:
ORA-20000: CANNOT CHANGE DATE OF BIRTH
ORA-06512: at "PARTICLE.PERSON_DOB", line 2
ORA-04088: error during execution of trigger
'PARTICLE.PERSON_DOB'
```

We should also notice the error code of ORA-20000. This is our -20000 parameter to RAISE APPLICATION ERROR.

15.22 PRIVILEGES REQUIRED TO USE TRIGGERS:

To work with triggers we have to satisfy some conditions. To create a trigger in our schema:

- We must have the CREATE TRIGGER system privilege
- One of the following must be true:
  - We own the table specified in the triggering statement
  - We have the ALTER privilege for the table specified in the triggering statement
  - We have the ALTER ANY TABLE system privilege

To create a trigger in another schema, or to reference a table in another schema from a trigger in our schema:

- We must have the CREATE ANY TRIGGER system privilege.
- We must have the EXECUTE privilege on the referenced subprograms or packages.

To create a trigger on the database, we must have the ADMINISTER DATABASE TRIGGER privilege. If this privilege is later revoked, we can drop the trigger but not alter it.

The object privileges to the schema objects referenced in the trigger body must be granted to the trigger owner explicitly (not through a role). The statements in the trigger body operate under the privilege domain of the trigger owner, not the privilege domain of the user issuing the triggering statement (this is similar to the privilege model for stored subprograms).

15.23 QUESTIONS:

1. Define Triggers. Explain the syntax of creating triggers in PL/SQL.
2. Explain the types of triggers.
3. Distinguish between BEFORE and AFTER Triggers.
4. Write the execution hierarchy of Triggers.
5. Distinguish between Statement Level and Row Level Triggers.
6. How to create DML statement Triggers.
7. Explain the conditional predicate to detect the Trigger DML operation.
8. Explain the creation of DML Row Level Trigger with example.
9. How to call procedure in Trigger body.
10. Explain the creation of Database Event Trigger with example.

11. Explain the creation of SCHEMA Trigger with example.
12. Explain Triggers on view.
13. How to View, Alter and Remove Triggers? Explain with examples.
14. Explain Cyclic Cascading in Triggers.
15. List and Explain the Restrictions on Trigger Conditions.
16. Write short note on Trigger Exceptions.
17. List and Explain Privileges Required to Use Triggers. Create a student view for student's personal information.
18. Create a view for Teacher and change it to select all teachers having subject I.T...

15.24 FURTHER READING

- Murach's Oracle SQL and PLSQL by Joel Murach, Murach and Associates.
- Oracle Database 11g PL/SQL Programming Workbook, ISBN: 9780070702264,

  By: Michael McLaughlin,John Harper, TATAMCGRAW-HILL
- Oracle PL/SQL Programming, Fifth Edition By Steven Feuerstein, Bill Pribyl
- Oracle 11g: SQL Reference Oracle press
- Oracle 11g: PL/SQL Reference Oracle Press.
- Expert Oracle PL/SQL, By: Ron Hardman,Michael McLaughlin, TATAMCGRAW-HILL
- Oracle database 11g: hands on SQL/PL SQL by Satish Asnani (PHI) EEE edition