CS405 - Database Programming using Oracle 11g

(#Highlighted by Aila Noor#)

Contents

Module-01: Conceptual Data Modeling & Entity Relationship Diagram (ERD) Review

1. Concept of ERD

Entity–relationship modeling was developed by Peter Chen and published in 1976; it serves as building block of relational database design. Entity relationship diagram is a graphical representation of the relationships between data in a database. It is the result of using systematic process and it just only visualize the business data instead of defining the business process. In very simple terms, ERD is a visual representation of data that describes how the data is related to each other.

2. Components of ERD

There are three main components of ERD and these are:

- Entity
- Attributes
- Relationships

3. Entity & Attributes

The word entity is rooted from the Latin word "en" which means being. Entity is name of place, person or thing about which something can be stored in a system. An entity can be a real-world object that can be easily identifiable. For example, in a school database, students, teachers, classes, and courses offered can be considered as entities. All these entities have some attributes or properties that give them their identity. An entity set is a collection of similar types of entities.

Entities are represented by means of their properties, called Attribute or Column. All attributes have values which are the qualities or data about Entities that is to be stored. An Attribute describes a property or characteristics of an entity. Continuing with the above example, a student entity may have name, class, and age as attributes. Attribute is the smallest storage unit of any database.

4. Relationships

Relationship represents how data is connected among entities in a given System. The association among the entities can also be termed as relationships. In our school example, the two entities e.g. student and course have an association or relation with each other as student enroll in a course. Interaction among entities is captured using relationships. In a database, relationships are created between different entities in order to remove the redundancy and ultimately improve the database performance.

Relationship define how data is connected among the entities in a given system or in other words how one entity is logically connected with another entity of the system. Relationships in

a database are said to be a combination of cardinality and optionality, where optional relationship is one in which there may or may not be a matching record in parent / child table and cardinality represents the concept of "how many" and normally it is 0 or more. Equation for creating relationship is as follow:

R ? ?? ??? ? ? h ?? ?? ? = ? ? ? ? ? ? ???? ?? ? + ? ? ??? ? ? ???? (?)

The concepts of cardinality and optionality are explained below in details.

Relationships are bi-directional in nature; Relationship between two entities A and B is as follow:

i. Relationship from A to B
ii. Relationship from B to A

5. Optionality

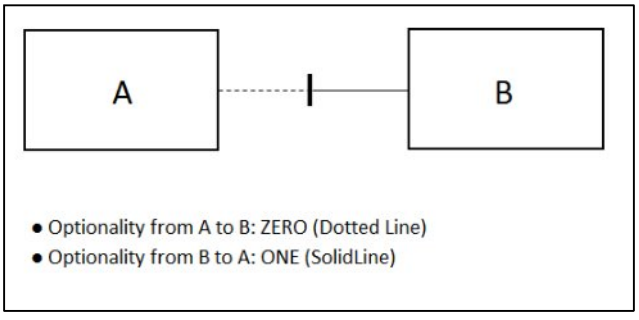
Participation in entity relationship is either optional or mandatory (…. Or _____). This means that minimum number of record which are present in child table per parent record. It can either be zero or one.

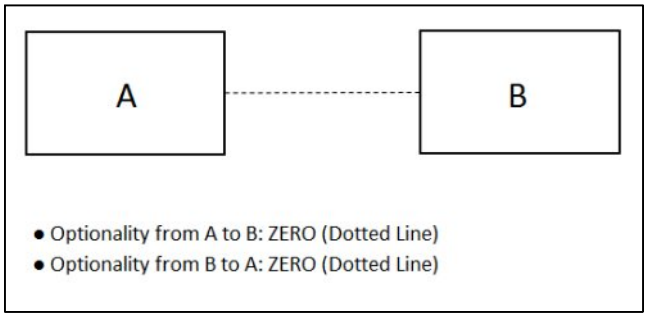
If we examine the *Parent & Child* relationship, it is quite possible for parent not to have any child record. Therefore, child is optional to parents.

On the other side, a child must have a parent and therefore, parent is mandatory to child. Hence minimum number of child record per parent in child table will show either 0 (…) or one (_____) . A dotted or solid line shows this kind of relationship. To translate …… or _____ following rules are to be followed:

Optional: -----: zero or one

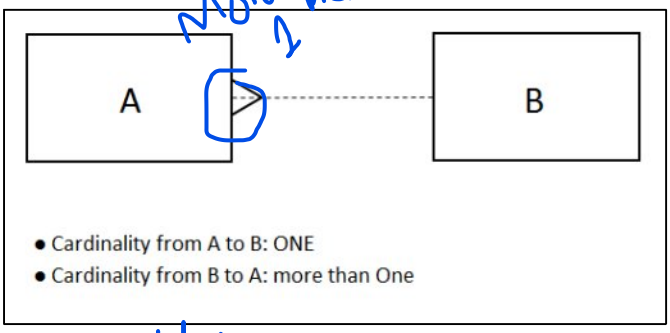
Mandatory: ____: Exactly one

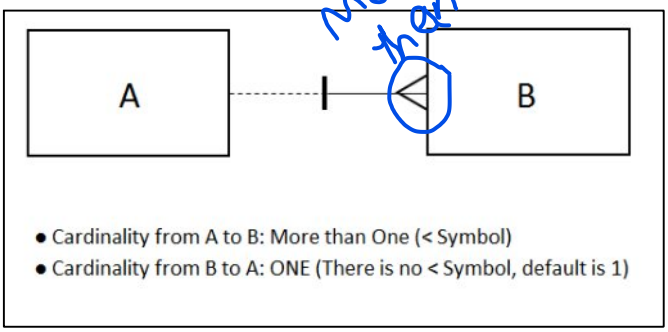




6. Cardinality

Cardinality expresses the maximum number of record which is present in child table per parent record in parent table. It can either be 1 or more than one represented by > or < symbol. Cardinality is read with opposite entity.



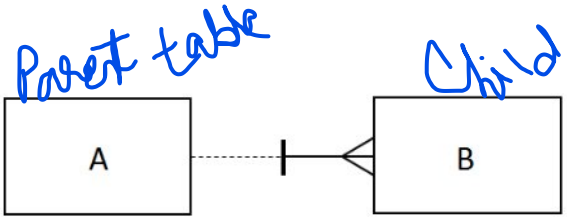


7. One-To-Many Relationship

A one-to-Many relationship is a type of cardinality that refers to the relationship between two entities A and B in which one record of entity A may be linked to zero, 1 or more records in entity B. Primary key of parent table will be written as Foreign Key in child table as a rule.

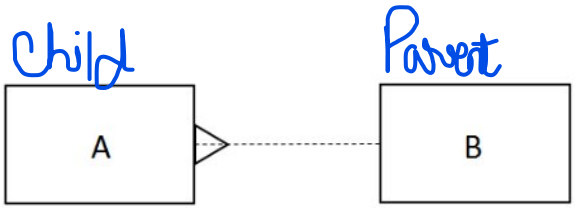
Consider the following illustrations:

Illustration 01:



- Relationship from A to B: A is having ZERO (Dotted Line) or MORE (<) occurrences in B
- Relationship from B to A: B is having exactly ONE (Solid Line) occurrence in A

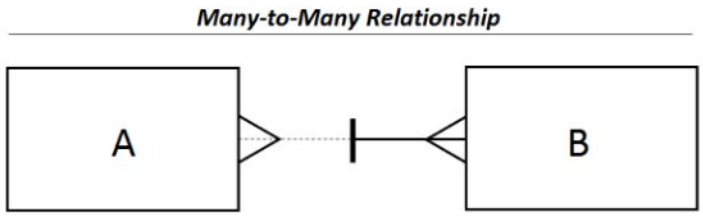
Illustration 02:

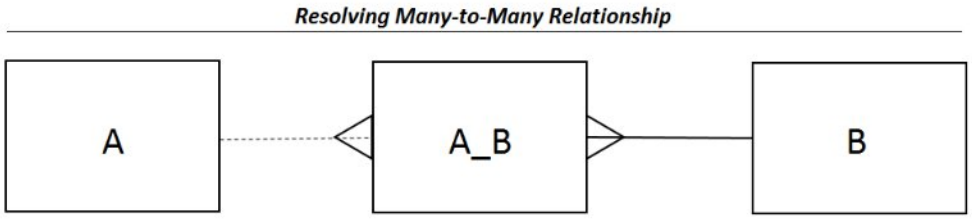


- Relationship from A to B: A is having ZERO (Dotted Line) or one (Cardinality) occurrence in B
- Relationship from B to A: B is having ZERO (Dotted Line) or MORE (>) occurrences in A

8. Many-To-Many Relationship

A many-to-many relationship is a type of cardinality that refers to the relationship between two entities A and B in which A may contain a parent record for which there are many children in B and vice versa. This means that a parent row in one table contains several child rows in the second table, and vice versa. Many-to-Many relations are tricky to represent and are not supported directly in the relational environment from implementation view point. To represent this kind of relationships, a third entity or intersection table is created where primary key (PK) of two tables act as foreign key (FK) and composite primary key (CPK) in third table. Many-to-many relationships are resolved into two one-to-many relationships, as shown below.





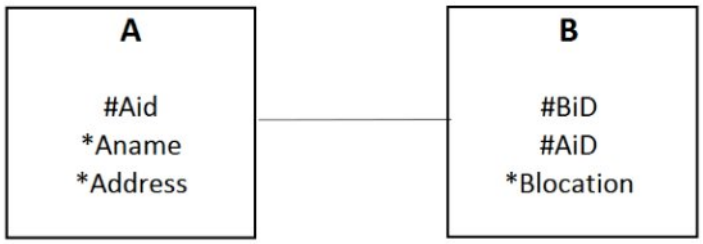
- Relationship from A to A_B: A is having ZERO (dotted line) or More (<)occurrence in A_B
- Relationship from B to A_B: B is having exactly ONE (solid line) or more (>)occurrence in A_B

9. One-To-One Relationship

A one-to-one relationship is a type of cardinality that refers to the relationship between two entities A and B in which one record of A may only be linked to one or zero record of B.

It is important to note that a one-to-one relationship is not a property of the data, but rather of the relationship itself as there is no parent-child relationship in on-to-one relation scenario. The following are rules to implement One-to-One:

i. There will be foreign key in any one of the participating table.
ii. Foreign key will be made as Unique key.



- Relationship from A to B: A is having exactly ONE occurrence in B (Solid Line)
- Relationship from B to A: B is having exactly ONE occurrence in A (Solid Line)

10. Implementing ERD using Scenario – 1

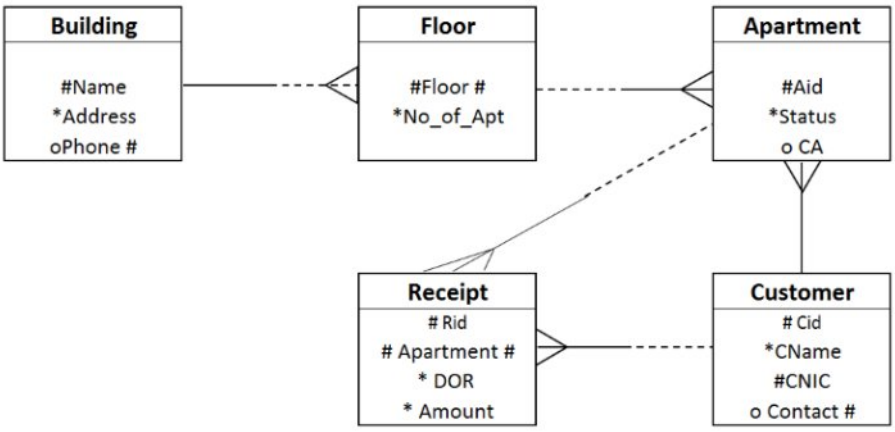
Scenario:

In a Building apartment renting scenario, there are apartments, buildings and customers. There are multiple floors in the building and on each floor there are multiple apartments, floor can have zero or no apartment. Each apartment can be rented at most one customer but customer can rent out multiple apartments from same building and apartment can be available on multiple floors or same. At the end of month a receipt is generated against which a rent is deposited.

The following entities can be derived from the above stated scenario:

- BUILDING
- APARTMENT
- CUSTOMER
- FLOOR
- RECEIPT

The relationship between the entities is illustrated in the below diagram:

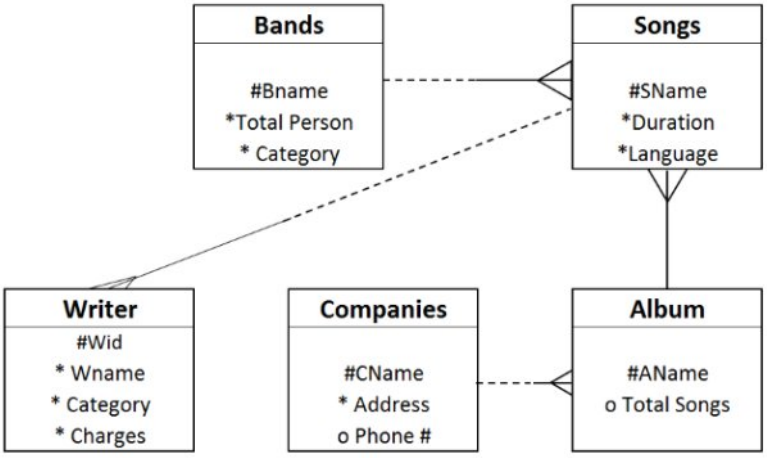


11. Implementing ERD using Scenario – 2

Consider another scenario to increase the understanding of the concept:

Scenario:

There are musical bands which record songs and request musical companies to launch their songs in the form of Album. Songs are written by song writers. Album can contain at least one song to be album and max of 12 songs. Writer can write multiple songs but songs are usually written irrespective of the number of person in the bands.



Module 02: Introduction to Oracle 11g on Cloud

1. Introduction

It's a cloud service hosted by Oracle with full access to the features and operations that are available with Oracle Database, but Oracle hosts the VM and cloud storage. You can perform all database management and development operations—without purchasing and maintaining hardware, without knowing backup and recovery commands, and without having to perform such complex tasks as database software upgrades and patching.

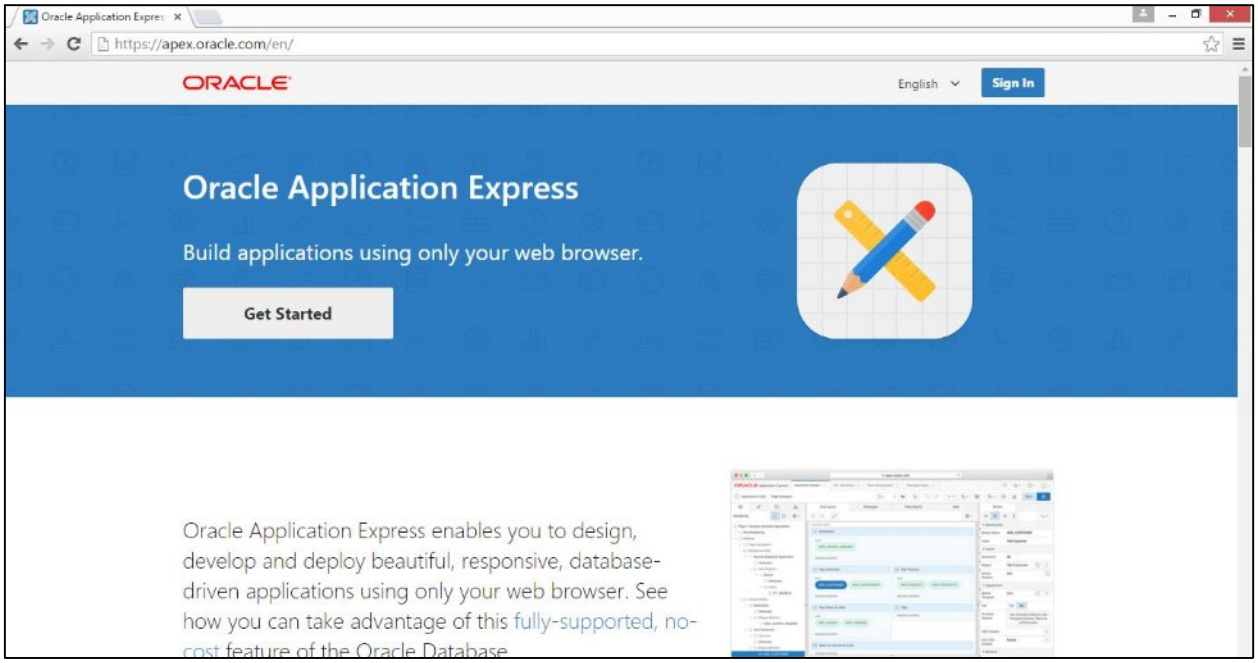
2. Login to Cloud

Login to the cloud will require the following steps:

Step 01:

Follow the following address to access Oracle 11g:

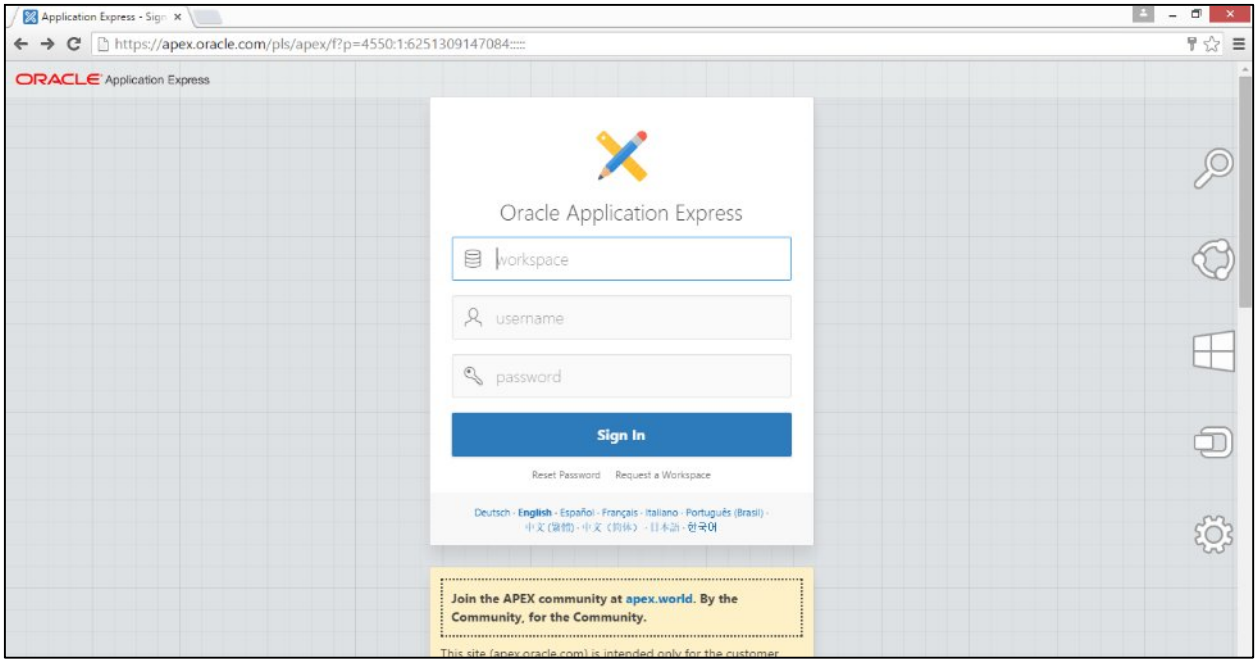
https://apex.oracle.com/en/



Step 02:

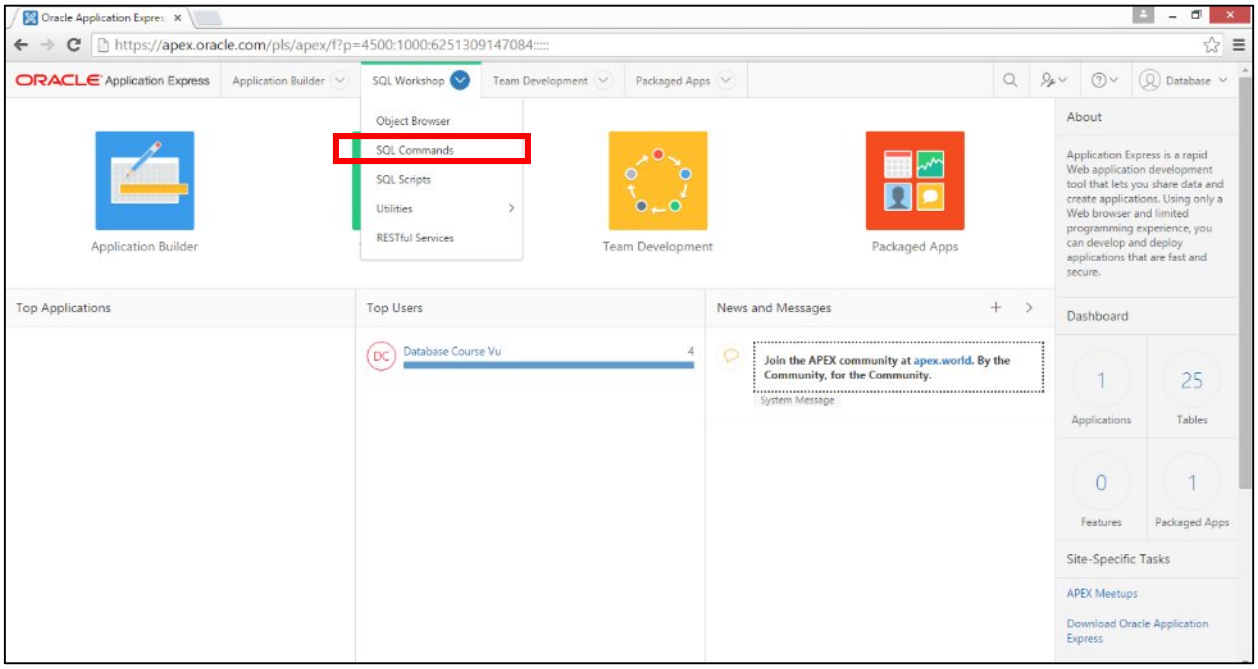
Click on the Sign In button on the upper right corner of the page and enter the following credentials:

Workspace Name	
Username	
Password	



Step 03:

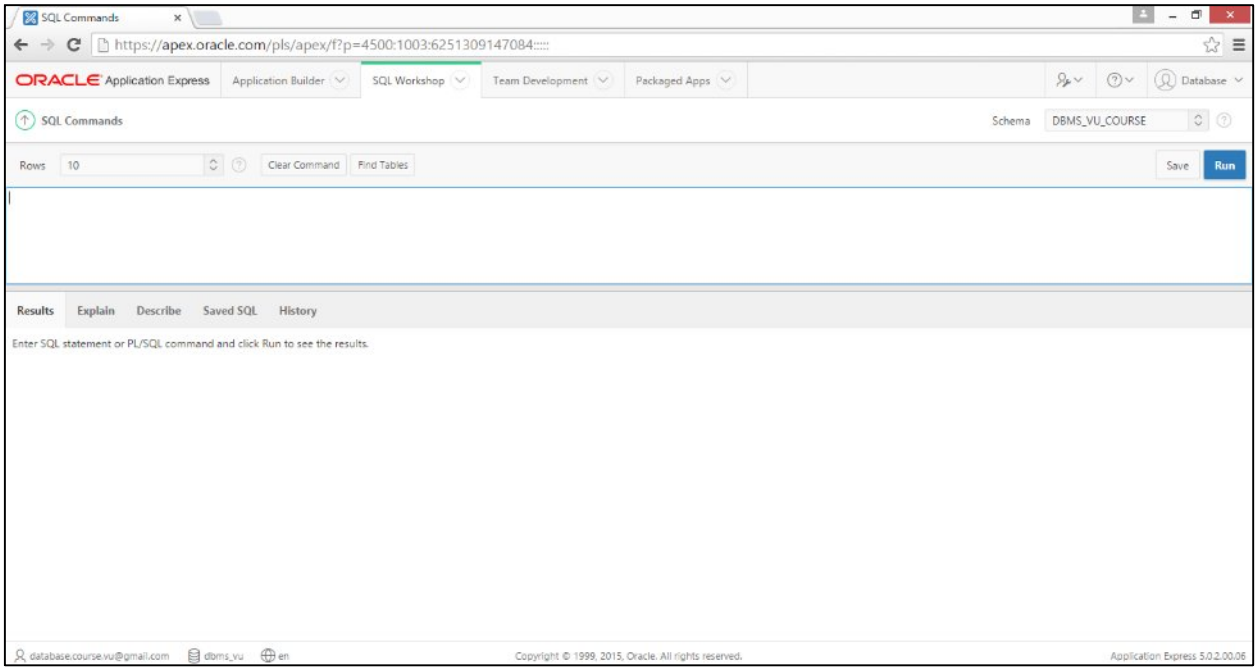
- Click on the SQL Workshop tab at the top-mid of the page



Step 04:

- Click on SQL Commands tab from the drop down list to access the code or enter SQL Statements / Commands and click Run to see the results

- Click on History Tab to access the saved code



Module 03: SQL Recap

1. Select Statement

Data Retrieval Language (DRL) is a command to retrieve data from a database object in the desired format. This is the most popular / flexible and the only way to retrieve data from a database. Command used for this purpose is called SELECT which allow us to specify the type of information which we want to retrieve. SELECT statements take the assumption that the tables are created and data has been populated in them. The basic syntax of SELECT statement is as follows:

```
SELECT distinct * | ColumnName FROM TableName;
```

Where:

- * mean all the column
- ColumnName is one or more column from table
- Distinct mean unique values from column

2. Implementing SELECT Statement

Few sample statements of SQL are as follows where the words in capital letters are SQL commands or statements and words in capital letters are table names.

```
SELECT depno FROM emp;

Details: It will display only deptno (column) and all rows from EMP Table.


SELECT DISTINCT deptno FROM emp;

Details: It will display only unique deptno (column) and all rows from EMP Table.


SELECT * FROM emp;

Details: It will display complete (all columns and all rows) from EMP Table.


SELECT ename, job FROM emp;

Details: It will display two columns (ename and  job) and all rows  from  EMP Table.
```

3. SQL and WHERE Clause

The WHERE clause includes a condition, which restricts the rows returned by the query. The WHERE clause eliminates all rows from the result set where the condition does not evaluate to True. From a large table, required rows can be fetch by using WHERE clause which applied to

Where restricts the number of rows
Select restricts the number of columns

each row of the table. If the given condition is satisfied then only it returns specific value from the table. You would use WHERE clause to filter the records and fetching only necessary records. The syntax would be like:

```
SELECT distinct * | ColumnName FROM TableName WHERE condition 1 and / or condition 2 and / or condition 3
```

Where conditions can include: >, <, =, <>, AND, OR, NOT

Thing to be remember is that If there are multiple conditions in where clause then at a time only one condition will be evaluated.

4. Implementing Where Clause

Consider the following examples assuming EMP as table name:

Example 01:

Write a query to find out list of all those employee name who are earning more than 2500 but less than 5000.

Solution:

```
SELECT ename FROM emp WHERE sal>2500 and sal < 5000;
```

Example 02:

Write a query to find out all those employees who are working in Dept # 20 with designation of Analyst but not earning more than 2000 and were hired at least 30 years ago.

Solution:

```
SELECT * FROM emp WHERE deptno=20 AND Job='Analyst' AND sal <2000 and

hiredate<sysdate -10000;

Note: Only days can be subtracted from Dates, here 10000 days will be subtracted from current (sysdate) date and then will be compared with hiredate.
```

5. Wildcard Characteristics in SQL

To broaden the selections of a SQL statement and to create regular expression where complete value to be searched is not known but part of value to be searched is known. There are two wildcard characters are used.

i. Percent sign (%) : Include Zero or more characters
ii. Underscore (_): Include only one

The percent sign is analogous to the asterisk (*) wildcard character used with MS-DOS. The percent sign allows for the substitution of one or more characters in a field. The underscore is

similar to the MS-DOS wildcard question mark character. The underscore allows for the substitution of a single character in an expression.

6. Implementing Wildcards – 1

wildcard are implemented using like like operator

Consider the following scenario:

Write a query to display list of name of all those employees who are having either E in the name or the name should end with G with at least two characters but should be working in Dept#30 and salary at least 1500.

The query would be like:

```
SELECT * FROM emp WHERE ename like '%E%' OR ename like '%-G' AND deptno=30 AND sal
>=1500;
```

Details: This query will return all the columns (*) and only those rows which are either have E due to % sign there can be zero or more character before and after E or the name should have two character ending with G, underscore (_)G is taking care of atleast two characters ending with G and department should belong to 30 and salary should be greater than 1500 (at least is translated into greater than or equals to (>=)

7. Implementing Wildcards – 2

Another scenario is: Write a query to display all information about all those employees who are having ER in the job with at least three characters in job and should be earning at least 2500 but at most 5000 and should be with company for at most 15 years

less then

Greater then

```
SELECT * FROM emp WHERE job like '%ER-%' AND sal > 2500 AND sal < 5000 AND hiredate
<=sysdate – 5600;
```

8. Single Row Functions

The Single Row Function operates on single rows only and returns one result per row. Single-row functions are used to manipulate data items. They accept one or more arguments and return one value for each row returned by the query. Single row functions can be character functions, numeric functions, date functions, and conversion functions. These functions require one or more input arguments and operate on each row, thereby returning one output value for each row. Single row functions can be used in SELECT and WHERE statement. Following lines give few examples:

VIP

Round and Trunc:

single row function types are below underlined

```
SELECT ROUND (194.683, 1), TRUNC (194.683, 1) FROM DUAL;

SELECT ename, LENGTH (ename), INSTR (ename, 'A'), CONCAT (ename,job)  FROM emp WHERE
instr(ename,'A')=3;

SELECT SUBSTR ('ABCDEFG', 3, 4) "Substring" FROM DUAL;
```

9. Group Functions

These functions manipulate groups of rows to give one result per group of rows. Group functions compute an aggregate value based on a group of rows. Group functions cannot be used with the WHERE clause. The example of the group or multiple row function is given below:

10. Implementing Group Functions - 1:

Scenario: Write a query to display sum, minimum, maximum and average salaries which company is paying to its employees

Solution: group function types

```
SELECT COUNT (*), sum (sal), min (sal), max (sal), Avg (sal) from emp;
```

Details: SQL Statement will return five columns and one rows

11. Implementing Group Functions - 2:

Scenario: Write a query to display sum, minimum, maximum and average salaries which company is paying to its employees but employees from Dept# 20 should not be shown and average salaries should be less than 1500

Solution:

```
SELECT COUNT (*), sum (sal), min (sal), max (sal), Avg(sal) from emp WHERE deptno !=20 and
avg (sal) <=1500;
```

Details: Group functions can't be used in where clause. Only single row functions are allowed in Where clause. The result of this query would be and Error because, as mentioned above, group functions cannot be used with the WHERE clause.

12. GROUP BY Clause

GROUP BY clause group together similar row together to form group and the multiple row function is used with GROUP BY Clause. The SQL GROUP BY clause is used in collaboration with the SELECT statement to arrange identical data into groups.

The basic syntax of GROUP BY clause is given below.

```
SELECT column1, column2
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2
```

Step-1		
Deptno	Sal	Group #
10	5000	Group -1
20	1200	Group -2
30	5002	Group-3
10	2400	Group -1
10	1900	Group-1
20	2100	Group-2

13. Implementing GROUP BY Clause 01

Situation: What is the total salary paid to each department?

Steps to Solution:

- Need to group together all rows of each department separately i-e creating groups
- Need sum of salary each department – group

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

The complete steps are shown below:

Step-2		
Group #	No of Rows	Group Function: Sum of Sal - Output is required by Query
1	3	5000+2400+1900 = 9300
2	2	1200+2100 = 3300
3	1	5002

The final output of the query is shown in the figure below:

Output of Query	
Deptno	Sum of Salary
10	9300
20	3300
30	5002

14. Implementing GROUP BY Clause 02

Consider the following:

Scenario: What is average and maximum salary paid to each Job who are reporting to MGR 7839?

Solution – 1

```
SELECT avg (sal), maximum (sal) FROM emp
WHERE mgr = 7839
GROUP BY job;
```

Details: Where and Group by clause can be used together as far as Group functions are not used where clause.

Solution – 2

```
SELECT avg (sal), max(sal), job, mgr, sal FROM emp
WHERE mgr=7839
GROUP BY job
```

Details: Only those columns can come after Select clause which are written after Group By clause. Result – Error job, mrg are not written after group by clause

15. HAVING Clause to restrict groups (only used groups in having lause)

The having clause, is just like the where clause, that filters the results in aggregated / grouped data. The Where clause cannot be used in the aggregated data, so SQL having clause is introduced to filter the results. The HAVING clause enables you to specify conditions that filter which group results appear in the final results.

VVP

The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

The basic syntax is as follows:

```
SELECT column1, column2
FROM table1, table2
GROUP BY column1, column2
HAVING [ conditions ]
```

16. Implementing HAVING Clause – 1

Scenario: Write a query to display average salary of each department if there are at least 2 employees working in the department and minimum salary is more than average salary by 100.

Solution:

```
SELECT avg (sal) FROM emp
GROUP BY deptno
HAVING count (*) > 3 and min (sal)>avg(sal)+100;
```

17. Implementing HAVING Clause – 2

Scenario: Write a query to display maximum and minimum salary by each department if average salary is more than 1500 of the department and less than 3000. The employee should not be included if there is any occurrence of 'A' in the ename or earning no commission and is hired at least six month before

Solution:

```
SELECT max (sal) , min(sal) FROM emp
WHERE ename not like '%A%' or comm is null and months_between(sysdate, hiredate)>6
GROUP BY deptno
HAVING max(sal) > 4500 and avg(sal)<1500;
```

18. ORDER BY Clause

The Order by clause is used with the SQL SELECT statement to sort the results in ascending or descending order. You have to specify one or more columns for what you want to sort the table result set. The ORDER BY keyword sorts the records in ascending order by default. To sort the records in a descending order, you can use the DESC keyword. The basic syntax is as follows:

```
SELECT column_name,
FROM table_name
ORDER BY column_name ASC|DESC,
```

Another example would be like:

```
SELECT deptno, sal FROM emp
ORDER BY sal;
```

19. What are Joins?

Joins are required when data from multiple tables is required. The standard join operation is knows as inner join. It horizontally combines two or more tables into a single working table. A primary key field in one table can be foreign key field in another table and a join operation is used to combine tables using a common key in both tables.

(comparison of PK and FK are implementation in joins)

20. Implementing Joins – 1

Basic Join Statement is as follows

```
SELECT empno, ename, d.deptno, dname
FROM emp e, dept d
WHERE d.deptno=e.deptno;
```

No of joins-No.tables-1(3-2=1 Now 1 is join)

21. Implementing Joins – 2

Scenario: Write a query to display list of employee name and name of department of all those employees who are hired at least 10 years before and are working as Analyst.

Solution:

```
SELECT empno,ename, d.deptno, dname,
ROUND (months_between(sysdate, hiredate),0), hiredate
FROM emp e, dept d
WHERE d.deptno=e.deptno and months_between(sysdate, hiredate) > 120 and job='Analyst';
```

22. What are Self Joins? (when PK and FK belong to same table)

A self-join is a query in which a table is joined (compared) to itself. Self-joins are used to compare values in a column with other values in the same column in the same table. In self-join a table is joined with itself, especially when the table has a FOREIGN KEY which references its own PRIMARY KEY. To join a table itself, means that each row of the table is combined with itself and with every other row of the table. Self joins are used in a recursive relationship. To explain that further, think of a COURSE table with columns including PREREQUISTE, COURS_NO and others. There is a recursive relationship between PREREQUSITE and COURSE_NO as PREREQUSITE is valid only if it is also a valid COURSE_NO.

23. Implementing Self Joins - 1

The basic syntax of self-join is:

```
SELECT a.column_name, b.column_name...
FROM table1 a, table1 b
WHERE a.common_field = b.common_field;
```

An example would be like:

```
SELECT e.ename,e.empno, b.ename, b.empno
FROM emp e , emp b, dept d
WHERE e.empno=b.mgr
```

Consider a Customer Table with the following attributes:

- Table: Customer (id, name, age, address, salary)

The code for the Self-Join would be:

```
SELECT a.ID, b.NAME, a.SALARY
FROM CUSTOMERS a, CUSTOMERS b
WHERE a.SALARY < b.SALARY;
```

24. Implementing Self Joins – 2

Scenario: Write a query to display the employee name, employee number along with name and employee no. of to whom it is reporting of all the employees who belong to accounting department.

Solution:

```
SELECT e.ename,e.empno, b.ename, b.empno
FROM emp e , emp b, dept d
WHERE e.empno=b.mgr and d.deptno = b.deptno
AND dname = 'ACCOUNTING';
```

25. SubQueries

it is alternate to joins.(a subquery which returns more than one rows then we use IN operator)

A subquery is a query within a query. Subqueries enable you to write queries that select data rows for criteria that are actually developed while the query is executing at run time. SQL subquery is usually added in the WHERE Clause of the SQL statement. Most of the time, a subquery is used when you know how to search for a value using a SELECT statement, but do not know the exact value in the database. Subqueries are an alternate way of returning data from multiple tables, or simply, alternative of joins. The general syntax is as follows:

```
SELECT *
FROM t1
WHERE column1 = (SELECT column1 FROM t2);
```

26. Implementing SubQuery – 1

Scenario: Write a query to display all those deptno where minimum salary is less than average salary of all the salary among all the employee and location of department have at least 5 characters in it end with K

Solution:

```
SELECT e.deptno, MIN (sal)
FROM emp e, dept d
WHERE d.deptno=e.deptno AND loc like ('---K')
GROUP BY e.deptno
HAVING MIN (sal) < (SELECT AVG (sal) FROM emp);
```

27. Implementing SubQuery – 2

Scenario: Write a query to display information of all those employees who are earning minimum salary but employees are neither working as Manager nor Clerk earning commission

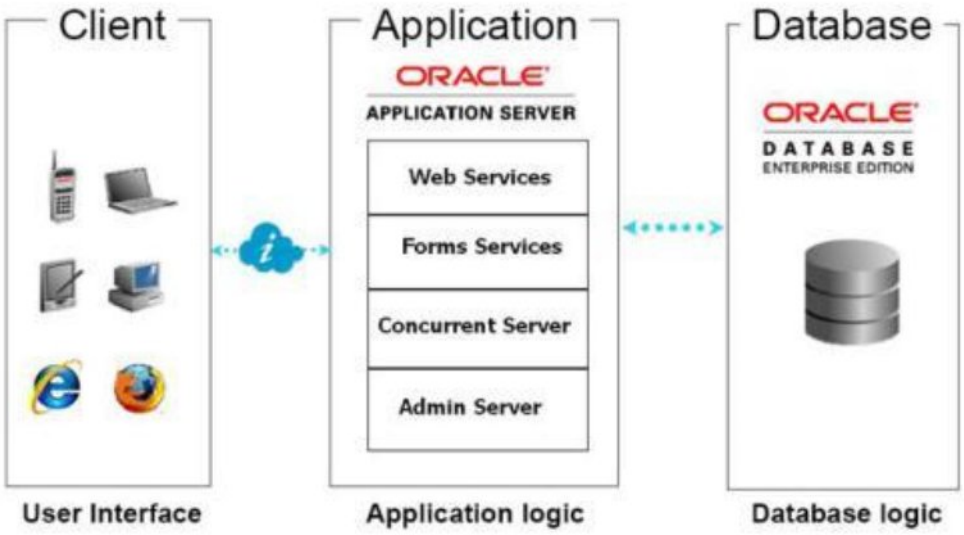
Solution:

```
SELECT ename, sal, deptno
FROM emp
WHERE sal = (SELECT MIN (sal) FROM emp)
AND job <> 'Manager' and job !='CLERK' and comm is not null;
```

Module 04: PL/SQL Concepts

1. Architecture of Oracle 11i

The architecture of Oracle 11i is shown in the image below followed by the explanation:



The architecture is distributed among the following three tiers:

1. Client / Desktop Tier:

The client interface is provided through HTML for the newer HTML-based applications, and via a Java applet in a Web browser for the traditional Forms-based interface. In Oracle Applications Release 11i, each user logs in to Oracle Applications Server using UI interface and then control is passed to Application Tier for verification as per Business Logic. Responsibility of Client layer is to make sure UI layer is presented to user successfully as per requirement.

2. Application Tier

The application tier has a dual role: hosting the various servers that process the business logic, and managing communication between the desktop tier and the database tier. This tier is sometimes referred to as the middle tier.

3. Database Tier

The database tier contains the Oracle database server, which stores all the data maintained by Oracle Applications. The database also stores the Oracle Applications online help information. More specifically, the database tier contains the Oracle data server files and Oracle Applications

database executable that physically store the tables, indexes, and other database objects for your system. In general, the database server does not communicate directly with the desktop clients, but rather with the servers on the application tier, which mediate the communications between the database server and the clients.

2. What is PL/SQL?

SQL is for iteract with database but pl is for programming language constructs using sql.
it is a declarative language. which use already defined commands.

PL/SQL stands for Procedural Language Extension to SQL. PL/SQL extends SQL by adding programming structures and subroutines available in any high-level language. PL/SQL is a procedural language designed specifically to embrace SQL statements within its syntax. PL/SQL program units are compiled by the Oracle Database server and are stored inside the database. And at run-time, both PL/SQL and SQL run within the same server process, bringing optimal efficiency. PL/SQL automatically inherits the robustness, security, and portability of the Oracle Database. PL/SQL is used for both server-side and client-side development.

3. Why PL/SQL?

Any application which need to access database need interface to access DB and an application that uses Oracle Database is worthless unless only correct and complete data is persisted. The longstanding way to ensure this is to expose the database only via an interface that hides the implementation details -- the tables and the SQL statements that operate on these. This approach is generally called the thick database paradigm, because PL/SQL subprograms inside the database issue the SQL statements from code that implements the surrounding business logic; and because the data can be changed and viewed only through a PL/SQL interface.

Since SQL is a declarative language it lacks language constructs like loops, procedures, functions; and these construct are required to write generic code, to provide programming language functionalities to SQL, PL/SQL is used with SQL as an embedded part in it.

4. How PL/SQL Works?

The basic unit of a PL/SQL source program is the block, which groups related declarations and statements. Block is minimum executable unit of PL/SQL which consists of Mandatory and Optional Parts. A PL/SQL block is defined by the keywords DECLARE, BEGIN, EXCEPTION, and END. These keywords partition the block into a declarative part, an executable part, and an exception-handling part. Blocks can be nested: Because a block is an executable statement, it can appear in another block wherever an executable statement is allowed. Following is the basic structure of a PL/SQL block.

```
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 raised in executable part]

END;
```

5. PL/SQL Executable

This section is enclosed between the keywords BEGIN and END and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a NULL command to indicate that nothing should be executed. Example is as follows:

```
BEGIN
DBMS_OUTPUT.PUT_LINE ('First Program in Oracle 11i');
END;



Detail: The Program will display First Program in Oracle 11i on the screen.
DBMS_OUTPUT.PUT_LINE is equivalent to cout or printf in C++/C
```

Module 05: General Programming Language Fundamentals of PL/SQL

1. Variable Declaration in PL/SQL

```
DECLARE
    I number (10):=0;
    Name varchar2(10);
    DOB date:=sysdate;
    Cost real:=390;
    PI CONSTANT NUMBER := 3.141592654;

BEGIN
     Dbms_output.put_line (I || name || dob || PI);

END;
```

In declaration section 5 variables are declared with respective data types. I, Name, DOB, Cost and PI are variables with associated Data types number (10), varchar2 (10), date, real and constant number respectively. In PL/SQL variables can only be declared in declaration section.

To display output on screen Dbms_output.put_line (I || name || dob || PI); is used and concetnation operator (||) is used to join output of multiple variables on the screen

Output of the Program: 006/10/20163903.141592654

2. Manipulating Variables in Blocks

```
DECLARE
    a integer: = 10;
    b integer: = 20;
    c integer;
    f real;

BEGIN
    c: = a + b;
    dbms_output.put_line ('Value of c: ' || c);
    f: = 70.0/3.0;
    dbms_output.put_line ('Value of f: ' || f);

END; /
```

In declaration section 4 variables are declared with respective data types. a,b,c,f are variables with associated Data types. In begin section variable C is assigned with new value which is sum of a and b. Value of c is displayed on screen using dbms_output.put_line('Value of c: ' || c); . In variable f new value is assigned by dividing 70.0 / 3.0 and value of F is displayed using

```
dbms_output.put_line('Value of f: ' || f);
```

Since F is real datatype and it can handle float value

Output of the Program:

```
Value of c: 30

Value of f: 23.333333333333333333333333333333333333
```

3. Scope of Variables in Nested Blocks

```
DECLARE
    -- Global variables
    num1 number: = 95;
    num2 number: = 85;

BEGIN
    dbms_output.put_line ('Outer Variable num1: ' || num1);
    dbms_output.put_line('Outer Variable num2: ' || num2);

DECLARE
    -- Local variables
    num2 number: = 195;
    num3 number: = 185;

BEGIN
    dbms_output.put_line('Inner Variable num1: ' || num1);
    dbms_output.put_line('Inner Variable num2: ' || num2);

  END;

END;
```

In PL/SQL there can be block within blocks, variables declared in outer block are accessible in inner block but variables declared inside inner block are not accessible in outer block. Like variable num1 and num2 are accessible in inner block but variable num3 is not accessible in outer block. If variable with same is declared in inner and outer block then value of variable declared in inner block will override the value in inner block. Output of the program is as follow, value of num2 which is declared in both outer and inner block, value of 195 (declared in inner block) is shown instead of 85 as initialized in outer block; however value of num1 is same in inner and outer block as num1 is declared only in outer block.

Output of the Program:

```
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 95
Inner Variable num2: 195
```

Module 06: SQL in PL/SQL

1. SELECT INTO Syntax

A variable that has been declared in the declaration section of the PL/SQL block can later be given a value with a select statement. The SELECT INTO statement retrieves data from one or more database tables, and assigns the selected values to variables or collections. The syntax is as follows:

```
SELECT item_name
    INTO variable_name
    FROM table_name;
```

By default, a SELECT INTO statement must return only one row. Otherwise, PL/SQL raises the predefined exception TOO_MANY_ROWS and the values of the variables in the INTO clause are undefined. The WHERE should contain condition to return at max match one row. Number of column before and after INTO clause needs to be same with respective data type in order to run the query.

2. Implementing SELECT INTO Syntax – I

```
DECLARE
    Id number (10):=0;
    Current_date date;

BEGIN
    SELECT 4982, sysdate INTO id, current_date
    FROM dual;
    dbms_output.put_line('Values are : ' || id || current_date);

END;

/
```

Explanation:

4982 and sysdate (06/10/2016) are loaded into local variables (id,current_date) and are

Displayed on the screen using dbms_output.put_line

Output of the Program

Values are: 498206/10/2016

3. Implementing SELECT INTO Syntax – II

```
DECLARE
    Id number(10):=0;
    Hire_date date;
    Name emp.ename%type;

BEGIN
    SELECT empno, ename, hiredate INTO id, name, hire_date FROM emp;
    dbms_output.put_line('Values are : ' || id || current_date);
```

```
END;
    /
```

Explanation:

The program will throw the following exception:

Error: ORA-01422: exact fetch returns more than requested number of rows

Because there no where clause in SQL statement due to which there are multiple rows return from the query and Select statement is not able to handle multiple rows.

4. Implementing SELECT INTO Syntax – III

```
DECLARE
    Id number(10):=0;
    date_hire date;
    Name emp.ename%type;

BEGIN
    SELECT empno, ename, hiredate INTO id, name, date_hire FROM emp WHERE empno=7369;
    dbms_output.put_line('Values are : ' || id || date_hire||name);

END;

/
```

Explanation:

Empno, ename , hiredate are loaded into local variables i-e id,name,hire_date ; since now there check in where clause on empno which is primary key so it will ensure that at max one row exists against empno=7369

Output of the Program

Values are: 736912/17/1980KAMRAN

5. Implementing SELECT INTO Syntax – IV

Consider the following practice question

Practice Question 01:

Write a PL/SQL block to retrieve maximum salary of the employee who is not working in department no 10 but have been associated with organization for past 5 years.

Solution:

```
DECLARE
    sal number(10):=0;

BEGIN
    SELECT max(sal) INTO sal FROM emp
    WHERE deptno!=10 and months_between (sysdate,hiredate)>=60;
    dbms_output.put_line('Maximum Salary is : ' || sal);

END;
    /
```

Explanation:

Maximum salary is retrieved into local PL/SQL variable i-e sal with employee should not be working in deptno 10 and due to months_between (sysdate, hiredate) it will return number of months between current date and hiredate and if number of months are greater than or equal to 60 i-e 5 years; only then that particular employee will be considered

Output of the Program:

```
Maximum Salary is : 54072
```

Note:

Value (54072) may vary from student to student as this value is shown in my login on www.apex.oracle.com

To do Questions by Students:

Write a PL/SQL block to retrieve maximum, minimum number of the employee who is not working as Analyst but belong to department having at least employees.

6. DML in PL/SQL

Getting data into and out of a database are two of the most important features of a database. It is possible to use DML statements INSERT, UPDATE and DELETE in PL/SQL while having no limitation on number of rows to be affected. Just to recap, brief explanation o DML statements is provided below:

- INSERT: The INSERT statement inserts rows into an existing table.
- UPDATE: The UPDATE statement updates (changes the values of one or more column) from a set of existing table rows.
- DELETE: The DELETE statement deletes rows from a table.

There is no change in syntax while using DML in PL/SQL, same syntax is true when DML is used outside PL/SQL

7. Implementing DML into PL/SQL – I

```
DECLARE
    No number (10):=2;

BEGIN
    INSERT INTO emp (empno) values (no);
    dbms_output.put_line('Row Successfully added');

END;
```

8. Implementing DML & SQL into PL/SQL – I

Imagine a situation where you have to write a query for the following scenario:

Scenario:

Write a PL/SQL block the set the salary of employee no 7369 increment by 25% of the maximum salary earned by any employee

Solution:

```
DECLARE
    No number(10):=2;
    salary emp.sal%type;

BEGIN
    SELECT max (sal) INTO salary FROM emp;
    UPDATE emp set sal = sal +salary*0.25 WHERE empno=7369;
    dbms_output.put_line('Row Updated');
    SELECT sal INTO salary FROM emp WHERE empno=7369;
    dbms_output.put_line('Updated Salary' || salary);

END;
```

Explanation:

In first SQL query, maximum salary is loaded into local variable salary

In update command salary of empno: 7369 is added with 25% of maximum salary (salary variable)

In last SQL query: Updated Salary is loaded into salary (local variable) and then displayed it on the screen

Output of the Program

```
Old sal = 67589.6, New sal = 84487Difference: 16897.4

Row Updated

Updated Salary84487
```

Practice Questions Website for SQL:
https://www.hackerrank.com/domains/sql/select/difficulty/all/page/1

9. Implementing DML & SQL into PL/SQL – II

Write a query for the following scenario

Scenario:

Write a PL/SQL block to insert a new row in employee table (empno, ename) only and code should not violate primary key constraint assuming empno is PK and for ename any 'ALLEN' can be used.

Solution:

```
DECLARE
    max_no emp.empno%type;

BEGIN
    SELECT MAX (empno) INTO max_no FROM emp;
    INSERT INTO emp (empno, ename) values (max_no+1, 'ALLEN');
    dbms_output.put_line ('Row added');

END;
```

Explanation:

In first SQL Query maximum empno is loaded into local variable max_no

In limited column Insert statement empno is max_no+1 (this will always ensure PK is unique) and ename is ALLEN.

Output of the Program:

Row Added

Note: There can be more than one solution to a same problem

10. Implementing DML & SQL into PL/SQL – III

Scenario:

Write a PL/SQL block to increase salary by 15% as retention bonus of all those employees who have been associated with company for more than 10 years

Solution:

```
BEGIN
    UPDATE emp set sal = sal*0.15 + sal
    WHERE months_between (sysdate, hiredate)>120;
    dbms_output.put_line('Salary updated');

END;


Output of the Program:

Salary updated
```

11. PL/SQL & Sequences - I

Sequence is a feature supported Oracle database systems to produce unique values on demand. Through sequences sequential unique numbers can be generated automatically to be used in tables. A sequence is a database object from which multiple users can generate unique integers. Using a sequence generator to provide the value for a primary key in a table is an easy way to guarantee that the key value is unique. Sequence numbers are generated independently of tables, so the same sequence can be used for one or for multiple tables however it is recommended not to used sequence in multiple table to avoid missing values .

12. PL/SQL & Sequences – II

Scenario:

Write a PL/SQL block which should insert primary key in the table using sequence in consistent way without unique key violation. Assuming there is existing data in primary key column.

Solution

```
CREATE sequence emp_no
START with 1
INCREMENT by 1
NoCache
NoCycle
ORDER;


CREATE table product (id number (10) primary key, pname varchar2(30));
  INSERT INTO product values(1, 'HD');
  select emp_no.nextval, emp_no.currval from dual;
```

```
DECLARE
   pid number (10):=0;
   new_id number(10):=0;

BEGIN
   SELECT max (id) into pid from product;
   dbms_output.put_line ('Maximum id : ' || pid);
   dbms_output.put_line ('Next Value of PK ' || ( pid + emp_no.nextval ));
   insert into product values (pid + emp_no.nextval, 'HD');
   dbms_output.put_line( ' Row Successfully added');

END;
```

Explanation:

First Sequence with emp_no is created to be used in PL/SQL Code

Product table is created which will be used to execute insert statement

In PL/SQL Code:

Maximum id from product table is loaded into local variable (pid)

In Insert statement value of pid is added to next value sequence using emp_no.nextval which will ensure uniqueness in the primary key column (pid) because maximum pid will have a numerical value added to it.

Row successfully Added will be displayed on the screen

Output of the Program:

Updated Maxid is: 1

Id: 1 successfully added

13. What is Commit?

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database. Every DML command (Insert, update, delete) is written to permanent storage after commit. The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command. The COMMIT statement makes permanent any changes made to the database during the current transaction. A commit also makes the changes visible to other users and execution of Block leads to auto-commit. The syntax for COMMIT command is as follows:

```
COMMIT;
```

14. Implementing PL/SQL and Commit

```
DECLARE
   pid number(10):=0;
   new_id number(10):=0;
   total_rows number(10):=0;

BEGIN
   SELECT MAX(id) INTO pid FROM product;
   INSERT INTO product values (pid + emp_no.nextval, 'HD');
   dbms_output.put_line( ' Row Successfully added');

COMMIT;
   SELECT COUNT (*) INTO total_rows FROM product;
   dbms_output.put_line( ' Total rows inserted : ' || total_rows);

END;
```

15. What is rollback?

The Rollback Statement is the transactional command used to undo transactions that have not already been saved to the database. The Rollback Statement can only be used to undo transactions since the last Commit Statement or Rollback Statement was issued. In very simple words, Used to undo the work performed by the current transaction. Rollback before commit will undo all the changes till last commit or start of block.

The syntax for ROLLBACK command is as follows:

```
ROLLBACK;
```

16. Implementing PL/SQL and Rollback

```
DECLARE
   pid number(10):=0;
   new_id number(10):=0;
   total_rows number(10):=0;

BEGIN
   SELECT MAX (id) INTO pid FROM product;
   INSERT INTO product values (pid + emp_no.nextval, 'HD');
   dbms_output.put_line( ' Row Successfully added');

ROLLBACK;
   SELECT COUNT (*) INTO total_rows FROM product;
   dbms_output.put_line( ' Total rows inserted' || total_rows);

END;
```

17. What is SAVEPOINT?

A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction. The SAVEPOINT statement names and marks the current point in the processing of a transaction. With the ROLLBACK TO statement, SAVEPOINT undo parts of a transaction instead of the whole transaction. Hence, Rollback is possible up to SAVEPOINT.

The syntax for rolling back to a SAVEPOINT is as follows:

```
ROLLBACK TO SAVEPOINT_NAME;
```

18. Implementing PL/SQL and SavePoint

```
DECLARE
   pid number(10):=0;
   new_id number(10):=0;
   total_rows number(10):=0;

BEGIN
   SELECT MAX (id) INTO pid FROM from product;
   INSERT INTO product values (pid + emp_no.nextval, 'HD');

SAVEPOINT A;
   INSERT INTO product values (pid + emp_no.nextval, 'HD');

SAVEPOINT B;

ROLLBACK to A;
   SELECT COUNT (*) into total_rows FROM product;
   dbms_output.put_line( ' Total rows inserted : ' || total_rows);

END;
```

Explanation:

Note: Should be inserting 2 but due to SAVEPOINT one row is rollback

Module 07: Conditional Control – I

1. IF – THEN Statements

This statement is used to execute the statement if the condition provided is true. The IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF. If the condition is TRUE, the statements get executed, and if the condition is FALSE or NULL, then the IF statement does nothing. END IF is to be used for each IF condition as shown in the following syntax:

```
BEGIN
  IF condition THEN
  Statement -1;
  Statement -2;

END IF;

END;
```

2. Implementing IF- Then Statement – I

Scenario:

Write a PL/SQL block to insert a row in emp table if total number of employees in department # 20 are more less than 50 because as policy maximum number of employees can be 50 in any department

Solution is not possible with SQL only because now conditional insertion of data in table is required. This create the need for control structure

Solution:

```
DECLARE
  total number (10):=0;
  max_id number(10):=0;

BEGIN
  SELECT count (*) into total FROM emp WHERE deptno=20;
  SELECT max (empno) into max_id from emp;

IF total <=50 THEN
  dbms_output.put_line('Table is not empty');
  INSERT into emp(empno, deptno, ename) values (max_id+1,20, 'Asim');
  dbms_output.put_line('Row is inserted in Department - 20');

END IF;

END;



Output of the Program:

Table is not empty
```

```
Row is inserted in Department - 20
```

3. Implementing IF- Then Statement-II

Scenario:

Write a PL/SQL block to update the salary of all the employees if there are at least 15 employees in the company and out of at least 5 have been attached with organization for more than 5 years then raise the salary of all the employee by 10%.

Solution

```
DECLARE
   total_emp number(10):=0;
   total number(10):=0;
   percentage real(10):=0.10;

BEGIN
   SELECT COUNT (months_between (sysdate, hiredate)) into total_emp from emp WHERE
   months_between (sysdate, hiredate)>=60;
   SELECT COUNT (*) into total FROM emp;

IF total_emp > 5 and total > 15 THEN
   update emp set sal = sal + sal*percentage;
   dbms_output.put_line('Increment given' );

END IF;

END;
```

Explanation:

First SQL is counting total employees who are attached with organization for more than 5 years and Second SQL is counting total employees in the organization.

Conditional Increment is given using IF clause if total employees are greater than 15 and those who are attached with organization is more than 5.

Output:

```
Increment given
```

4. IF – Then – Else Statement

A sequence of IF-THEN statements can be followed by an optional sequence of ELSE statements, which execute when the condition is FALSE. Only once closing END IF is required for this statement. The basic syntax is as follows:

```
BEGIN
  IF condition THEN
  Statement -1;
  Statement -2;

ELSE
  Statement-1;

END IF;

END;
```

5. Implementing IF- Then-else Statement-I

Consider the following scenario to understand the IF-THEN-ELSE Statement:

Write a PL/SQL block to check whether you are logged in as 'APEX_PUBLIC_USER' and if this is true then display success message and then check whether maximum salary has been achieved against designation of ANALYST or not. As company policy maximum salary which can be paid to Analyst is not more than 5000 otherwise display message not achieved.

Solution

```
DECLARE
  max_sal number(10):=0;
  designation varchar2(30):='ANALYST';

BEGIN
  SELECT MAX (sal) INTO max_sal FROM emp WHERE job=designation;

IF user = 'APEX_PUBLIC_USER'  and max_sal <=5000 THEN
  dbms_output.put_line ('Current Maximum Salary for ' || designation || ' is : ' || max_sal);
  dbms_output.put_line ('Maximum Salary not achieved');

ELSE
  dbms_output.put_line ('Maximum achieved');

END IF;

END;
```

Explanation:

User is the environment variable which is holding value which is recognizable by system as
  Schema user i-e APEX_PUBLIC_USER i-e This is default user value

Output of the Program:

Maximum Achieved

6. Implementing IF- Then-else Statement-II

Write a PL/SQL block to proceed as follow of emp = 7369:

I. Salary > 3000 and less than 6000 and designation = 'MANAGER'
   Message: Senior Management
II. Not part of Management

Solution

```
DECLARE
  designation emp.job%type;
  salary emp.sal%type;

BEGIN
  select job , sal into designation, salary from emp where empno=7369;

IF designation ='ANALYST' and salary > 3000 and salary < 6000 THEN
  dbms_output.put_line('Senior Management');

ELSE
  dbms_output.put_line('Not  Part of Management');

END IF;

END;
```

Output of the Program:

Not Part of Management

Note: The output of the program may vary depending on data available in emp table of student account

7. ELSIF Statement

The IF THEN ELSIF statement runs the first statements for which condition is true. Remaining conditions are not evaluated. If no condition is true, the else statements run, if they exist; otherwise, the IF THEN ELSIF statement does nothing. There will be one END IF per IF keyword. The basic syntax is as follows:

```
BEGIN
  IF condition THEN
  Statement -1;
  Statement -2;
```

```
ELSIF condition THEN
  Statement-1;

END IF; -- Only one End if per if

END;
```

8. Implementing IF- THEN-ELSIF Statement I

Scenario

Write a PL/SQL block to proceed as follow of emp = 7369:

1. Salary > 3000 and less than 6000 and designation = 'MANAGER'
   Message: Senior Management
2. Salary > 6000 and less than 9000 designation = 'PRESIDENT'
   Message: Executive Management
3. Not eligible

Solution

```
DECLARE
  designation emp.job%type;
  salary emp.sal%type;

BEGIN
  SELECT job , sal into designation, salary FROM emp WHERE empno=7369;
  IF designation ='PRESIDENT' and salary > 6000 and salary < 9000 THEN
  dbms_output.put_line('Executive Management');

ELSIF designation='MANAGER' and salary > 3000 and salary < 6000 THEN
  dbms_output.put_line('Senior Management');

ELSE
  dbms_output.put_line('Not part of Management');

ENF IF;

END;
```

Output of the Program:

```
Not part of Management
```

Note: The output of the program may vary depending on data available in emp table of student account

9. Implementing IF-THEN-ELSIF Statement-II

Scenario

Write a PL/SQL block to precede display for any given date:

1. If day is Saturday or Sunday then display ‘ Weekend’
2. If day is Monday or Tuesday then display ‘ Start of week’
3. If Day is Wednesay or Friday then display ‘ Toward end of week’

Solution

```
DECLARE
   v_date DATE := TO_DATE('18-Dec-2013', 'DD-MON-YYYY');
   v_day  VARCHAR2(15);

BEGIN
   v_day := RTRIM(TO_CHAR(v_date, 'DAY'));
   DBMS_OUTPUT.PUT_LINE ('Day of the Date: ' || ' 18-Dec-2012 is : '  || v_day);

IF v_day IN ('SATURDAY', 'SUNDAY') THEN
   DBMS_OUTPUT.PUT_LINE (v_date||' falls on weekend');

ELSIF v_day in ('MONDAY', ' TUESDAY') THEN
   DBMS_OUTPUT.PUT_LINE (v_date||' falls on start of week');

ELSIF v_day in ('WEDNESDAY', ' FRIDAY') THEN
   DBMS_OUTPUT.PUT_LINE (v_date||' falls toward mid week');

END IF;
   DBMS_OUTPUT.PUT_LINE('Done...');

END;
```

Explanation:

V_date is initialized with 18-Dec-2013, to_char (v_date, ‘DAY’) will return DAY on 18-Dec-2013.

There are multiple IF and ELSIF structure as per requirement, point to note is that there is only one END because there is only ONE if keyword

Output of the Program:

```
Day of the Date:  18-Dec-2012 is : WEDNESDAY

12/18/2013 falls toward mid week

Done...
```

10. NESTED IF Statement

NESTED IF statement revolves around IF conditions associated with Parent & Child. This statement works in such a way that if Parent IF is true, Child IF will also be true. In other words, in NESTED IF statement, IF statement associated with Child will only be executed if IF statement associate with the parent is executed. Following example would elaborate further.

11. Implementing Nested-IF – I

```
DECLARE
   a number(3) := 100;
   b number(3) := 200;

BEGIN
   IF( a  between 0 and 100 ) THEN
   IF( b  between 101 and 200 ) THEN
   dbms_output.put_line('Value of a is 100 and b is 200' );

END IF;

  END IF;
   dbms_output.put_line('Exact value of a is : ' || a );
   dbms_output.put_line('Exact value of b is : ' || b );

END;
```

Explanation:

First IF condition will be evaluated to TRUE if value of a is between 0 and 100 (0 and 100 are included) only then next IF will be checked for TRUE or FALSE, in next IF condition will be evaluated to TRUE is value of b is between 101 and 200, if both IF conditions will be true only then dbms_output.put_line('Value of a is 100 and b is 200'); will get executed. In this case both IF conditions are true (as shown in output below)

Output of the Program

```
Value of a is 100 and b is 200

Exact value of a is : 100

Exact value of b is : 200
```

```
DECLARE
   a number(3) := 100;
   b number(3) := 200;

BEGIN
   IF( a  between 0 and 100 ) THEN
   IF( b  between 101 and 200 ) THEN
   dbms_output.put_line('Value of a is 100 and b is 200' );

END IF;
```

```
  END IF;
   dbms_output.put_line('Exact value of a is : ' || a );
   dbms_output.put_line('Exact value of b is : ' || b );

END;
```

12. Questions to Practice:

i. Write at least 3 different versions of questions solution discussed.

ii. Write a PL/SQL block to check whether number of employess in Emp table are odd or even, if number of employees are odd then display message ‘ Odd number of employees else display ‘ Even number of employees’

Module 08: Conditional Control – II

1. Case Statement

Case statements are alternate to the IF-Then statements. The CASE statement chooses from a sequence of conditions, and executes a corresponding statement. This statement selects ONLY one sequence of statement to execute and when IF case is not matched then else part is executed. Exception is raised when ELSE part is not written with no matching case in IF

condition. But still this statement is considered to be more readable and efficient. The syntax is given below:

```
CASE selector

WHEN expression1 THEN sequence_of_statements1;

WHEN expression2 THEN sequence_of_statements2;

ELSE sequence_of_statements; --Optional

END CASE
```

2. Implementing CASE Statement – I

Scenario: Write a PL/SQL block to implement the following business rules:

Grade	Message
A	Excellent
B	Good
C	Fair
D	Fail
Any other	No such grade

Solution:

```
DECLARE
  grade char(1) := 'A';

BEGIN
  CASE grade -- Selector
  WHEN 'A' then dbms_output.put_line('Excellent');
  WHEN 'B' then dbms_output.put_line('Very good');
  WHEN 'C' then dbms_output.put_line('Well done');
  WHEN 'D' then dbms_output.put_line('You Fail');
  ELSE dbms_output.put_line('No such grade');

END CASE;

END;
```

Explanation:

Value of local variable grade is initialized with A, in the case statement first case is matched and respective message (Excellent) is displayed. Whenever case is matched related statements are executed and control is transferred after END CASE.

Output of the Program

```
Excellent
```

3. Implementing CASE Statement – II

Scenario: Write a PL/SQL block to implement the following business rules:

User	Message
APEX_PUBLIC_USER	Sample User
SCOTT	Owner of Schema
DBA	Admin
Any other	Invalid User

Solution:

```
BEGIN
   CASE user
   WHEN 'APEX_PUBLIC_USER' then dbms_output.put_line(user || ' is Test User');
   WHEN 'SCOTT' then dbms_output.put_line(user|| 'Owner of Schema');
   WHEN 'DBA' then dbms_output.put_line(user || 'Admin User');
   ELSE dbms_output.put_line('Not a Valid user');

END CASE;

END;

Output of the Program:

APEX_PUBLIC_USER is Test User
```

4. Searched Case Statements

It does not have a selector and Conditions are mentioned in WHEN clause. Conditions are evaluated to be true or false. The SQL searched CASE expression gives you more flexibility when writing your comparison conditions. The reason you get more flexibility is because of the fact that each comparison condition is written out as a complete condition, and the comparison operator (like “>”, “<", etc.) is also written out in the condition.

The basic syntax is as follows:

```
CASE
   WHEN Condition -1 THEN sequence_of_statements1;
   WHEN Condition -2 THEN sequence_of_statements2;
   ELSE sequence_of_statements; --Optional

END CASE
```

If no case condition is evaluated to be true and there is no else then exception is raised

5. Implementing Searched CASE Statement – I

Scenario: Write a PL/SQL block to implement the following business rules from emp table:

Salary	Message	Action (salary_stats table)
Between 10000 and 15000	Status='Excellent'	Insert row with current date, max sal and status
Between 7000 and 9999	Status: Very Good	Insert row with current date, max sal and status
Between 4000 and 6999	Status: Good	Insert row with current date, max sal and status
Any other	Status No such Salary	Insert row with current date, max sal and status

Solution:

```
DROP table salary_stats;

CREATE table salary_stats (date_recorded date, salary number(15), Status varchar2(15));

SELECT * FROM salary_stats;

DECLARE
  salary number (10):=0;
  status varchar2(15);

BEGIN
  SELECT max (sal) into salary FROM emp;
  CASE
  WHEN salary > 10000 and salary < 15000 THEN
  status :='Excellent';
  INSERT INTO salary_stats values (sysdate, salary, status);
  dbms_output.put_line(status || '  and row inserted');
  WHEN salary > 7000 and salary < 10000 THEN
  status :='Very Good';
  INSERT INTO salary_stats values (sysdate, salary, status);
  dbms_output.put_line(status || ' and row inserted');
  WHEN salary > 4000 and salary < 70000 THEN
  status :='Good';
  INSERT INTO Salary_stats values (sysdate, salary, status);
  dbms_output.put_line(status || ' and row inserted');

ELSE
  status :='Not in Range';
  INSERT INTO salary_stats values (sysdate, salary, status);
  dbms_output.put_line(status || ' and row inserted');
```

```
END CASE;

END;
```

Explanation:

As per requirement, maximum sal is loaded into local variable salary and then conditions are written after WHEN clause as per question. Point to note here is that variable name after CASE key word rather there is WHEN keyword with condition. In every WHEN clause insert statement is executed depending on the case. As per data in the tables, following output is generated and row is inserted in salary_stats table. ELSE part is executed here because no CASE was matched, if there would have been no ELSE part then program will throw exception

Output of the Program:

Not in Range and row inserted

Note: The output of the program may vary depending on data available in emp table of student account

6. Implementing Searched CASE Statement – II

Scenario: Write a PL/SQL block to implement the following business rules from emp table of empno=7369

Designation	Action (salary_stats table)
CLERK	Update Salary by 0.09 of current salary
MANAGER	Update Salary by 0.08 of current salary
ANALYST	Update Salary by 0.07 of current salary
Any other	No Raise

Solution

```
DECLARE
  jobid emp.job%TYPE;
  empid  emp.empno%TYPE := 7369;
  sal_raise  NUMBER(3,2);

BEGIN
  SELECT job INTO jobid from emp WHERE empno = empid;

CASE
  WHEN jobid = 'CLERK' THEN sal_raise := .09;
  update emp set sal = sal + sal*sal_raise where empno=empid;
  dbms_output.put_line ('Salary raised by ' || sal_raise);
```

```
   WHEN jobid = 'MANAGER' THEN sal_raise := .08;
   update emp set sal = sal + sal*sal_raise where empno=empid;
   dbms_output.put_line ('Salary raised by ' || sal_raise);
   WHEN jobid = 'ANALYST' THEN sal_raise := .07;
   update emp set sal = sal + sal*sal_raise where empno=empid;
   dbms_output.put_line ('Salary raised by ' || sal_raise);
   ELSE sal_raise := 0;

END CASE;

END;
```

Explanation:

As per data in emp table there is not matching JOBID i-e CLER OR MANAGER OR ANALYST, control is transferred to ELSE part of the program, in the else part there is no dbms_output.put_line statement so effectively there will no output on the screen. As shown below: Statement Processed is showing that there is not syntax error and code is executed successfully only.

Output of the Program:

Statement Processed

Note: The output of the program may vary depending on data available in emp table of student account

Practice Question:

Write at least 3 different variations of questions discussed.

- Given the tables below:
- Order(oid, cid, order_date, order_amount);

Order Amount	Action (Order Table)
>10000 and < 20000	Give discount to customer by 5%
>20000 and < 30000	Give discount to customer by 8%
>30000 and < 40000	Give discount to customer by 10%
Any other	No Discount

Module 09: Iterative Control – I

1. Simple Loop

The simple loop is different from other loops available in other programming languages; simple loop consists of a structure to repeat statements. The sequence of statements is executed in each iteration and it needs a termination in terms of exit statement to stop the loop. Loop without exit condition considers to be an infinite loop. Sequence of statement(s) may be a single statement or a block of statements. Loop will keep on execution the statement as long as exit condition is FALSE and will terminate as soon as exit condition is evaluated to true, this behavior is totally opposite other type of loops (FOR, WHILE, DO-WHILE). The basic condition is given in the followings:

```
LOOP
   Statement -1;
   Statement -2;

EXIT | EXIT WHEN condition is true;

End loop;

End;
```

2. Implementing Simple Loop – I

```
DECLARE
   rep number (10):=0;

BEGIN

LOOP
   dbms_output.put_line (rep);

END LOOP;

END;
```

Explanation:

There is no exit condition written in the program to terminate the loop, program will run indefinitely and will result in abnormal termination. There is no syntax error in the code.

Output of the Program:

ORA-10260: limit size () of the PGA heap set by event 10261 exceeded ORA-10260: limit size (1024000) of the PGA heap set by event 10261 exceeded

3. Implementing Simple Loop - II

```
DECLARE
   rep number(10):=0;

BEGIN

LOOP
   if rep >=5 then
   dbms_output.put_line ('Value of Rep:  ' || rep);
   EXIT;

END IF;
   rep:=rep+1;

END LOOP;

END;
```

Explanation:

Local variable rep is initialized with 0, the loop will keep on executing when rep is <5, as soon as value of rep is greater than 5 the loop and program will exit.

Output of the Program:

```
Value of Rep:  5
```

4. Implementing Simple Loop with SQL – I

Task to do:

Write a PL/SQL block to insert 10 row in emp table, add data in empno column only with starting value of 0 and ending value of 10.

```
DECLARE
   rep number (10):=0;

BEGIN

LOOP
   INSERT INTO emp (empno) values (rep);
   rep:=rep+1;
   EXIT WHEN rep > 10 ;

END LOOP;

END;
```

Explanation:

The loop will be in execution when value of rep is less than 11, the loop will run till the condition of exit (rep>10) is false and will terminate on true condition i-e when value of rep is greater than 11. In each iteration a row is added in Emp table.

Output of the Program:

Statement processed.

5. Implementing Simple Loop with SQL – II

Task to do:

Write a PL/SQL block to display the reverse of maximum salary from EMP table.

```
DECLARE
   N NUMBER(5):=0;
   REV NUMBER(5):=0;
   R NUMBER(5):=0;
   total_emp number(10):=0;

BEGIN
   SELECT max(sal) into n FROM emp;
   Dbms_output.put_line('maximum Salary is : ' || n);

LOOP
   R:=MOD(N,10); --return reminder
   REV:=REV*10+R;
   N:=TRUNC(N/10);
   EXIT WHEN n = 0;

END LOOP;
   Dbms_output.put_line('Reverse Value :  ' || rev);

END;
```

Explanation:

As per output below the maximum salary is 117564 and is loaded into local variable n. and is displayed.

First iteration:

R = Mod (117564,10) – It will return remainder which is 4

Rev:=Rev*10 + R = 0*0 + 4 =4

N:=N/10 – 117564 / 10 = 11756 – N is a number due to which float value will not be shown.

Exit Condition is false because N !=0

Second Iteration:

R = Mod (11756, 10) – It will return 6

Rev:=4*10+6 = 46

N:=N/10 = 11756/10=1175

Third Iteration:

R = Mod (1175, 10) – It will return 5

Rev:=46*10+5 = 465

N:=N/10 = 1175/10=117

Fourth Iteration:

R = Mod (117, 10) – It will return 7

Rev:=465*10+7 = 4657

N:=N/10 = 117/10=11

Firth Iteration:

R = Mod (11, 10) – It will return 1

Rev:=4657*10+1 = 46571

N:=N/10 = 11/10=1

Sixth Iteration:

R = Mod (1, 10) – It will return 1

Rev:=46571*10+1 = 465711

N:=N/10 = 1/10=0

Loop will terminate after 6th Iteration.

Output of the Program:

Maximum Salary is: 117564

Reverse Value : 465711

Note: The output of the program may vary depending on data available in emp table of student account

6. While Loop

Basic loop structure encloses sequence of statements in between the LOOP and END LOOP statements. Sequence in statements is executed in each iteration and the loop exits when the exit condition evaluates to be false. While loop syntax is given below:

```
WHILE condition LOOP
  sequence_of_statements

END LOOP;
```

7. Implementing While Loop – I

```
DECLARE
  done BOOLEAN: = TRUE;

BEGIN
  WHILE done LOOP
  DBMS_OUTPUT.PUT_LINE ('This line got printed once only');
  done := FALSE;

END LOOP;

END;

Output of the Program:

This line got printed once only
```

8. Implementing While Loop – II

Consider the following lines of code:

```
DECLARE
  counter number (4) := 1;

BEGIN
  WHILE counter <= 8 loop
  dbms_output.put_line (counter);
  IF counter=4 then
  Exit; -- Loop will be terminated

END IF;
  counter := counter +1;

End loop;

END;

Explanation:

 Loop will be terminated without reaching false condition
```

```
Output of the Program:

1

2

3

4
```

9. Implementing While Loop – III

Task to Do:

Write a PL/SQL block to insert 10 row in emp table using while loop, populate only empno column only with starting value of 0 and ending value of 10.

```
Declare
   rep number (10):=0;

Begin
   while (rep < 10) loop
   insert into emp (empno) values (rep);
   rep:=rep+1;
   dbms_output.put_line (' Rows are inserted in the loop with row no ' || rep);

END LOOP;
    dbms_output.put_line (' Transaction Completed ');

END;


Output of the Program

Rows are inserted in the loop with row no 1

 Rows are inserted in the loop with row no 2

 Rows are inserted in the loop with row no 3

 Rows are inserted in the loop with row no 4

 Rows are inserted in the loop with row no 5

 Rows are inserted in the loop with row no 6

 Rows are inserted in the loop with row no 7

 Rows are inserted in the loop with row no 8

 Rows are inserted in the loop with row no 9
```

```
Rows are inserted in the loop with row no 10

Transaction Completed
```

10. Implementing While Loop – IV

Task to do:

Write a PL/SQL block to calculate the sum of first 100 even integers starting from 1

```
DECLARE
   counter number(4):=2;
   total number(5):=0;

BEGIN
   while counter < 100 loop
   total:=total+counter;
   dbms_output.put_line (' Current Sum is : ' || total );
   counter:=counter+2;

END LOOP;
   dbms_output.put_line (' Sum of even integers between 1 and 100 is '  || total );

END;
```

11. Implementing While Loop using SQL – I

Task to do:

Write a PL/SQL block to calculate the total of first 100 even integers starting from 1, at any point in time if total exceeds 400 and remain less than 800, add a new row in temp table with columns of sum, current date and status.

```
DECLARE
   counter number (4):=2;
   total number(5):=0;
   status varchar2(80):='';

BEGIN
   while counter < 100 loop
   total:=total+counter;
   dbms_output.put_line (' Current Sum is : ' || total );
   dbms_output.put_line (' Current Sum is : ' || total );
   counter:=counter+2;
         if (total > 400 and total < 800) then
```

```
        status:='Total is greater than 400 and less than 800';
        insert into temp values (total, sysdate,status);
        dbms_output.put_line ('Row Inserted');
      END IF;

END LOOP;
  dbms_output.put_line ('Total  of even integers between 1 and 100 is '  || total );

END;
```

12. Do-While Loop

A DO-While loop is a statement that executes a statement at least once and then repeatedly executes the block, or not, depending on a given condition at the end of the block. The code within the block is executed, and then the condition is evaluated. If the condition is true the code within the block is executed again. This repeats until the condition becomes false. This can be implemented using simple loop and no direct implementation is provided in PL/SQL.

13. Implementing Do While Loop

```
DECLARE
  current_val number (10):=0;

BEGIN

LOOP
  dbms_output.put_line ('Value of count is : ' || current_val);
  current_val:=current_val+1;
  exit when current_val> 10 ;

END LOOP;

END;
```

Explanation:

Since there is no direct implementation of Do-While Loop, Simple loop is an implementation of Do-While Loop also because exit condition is evaluated in the end to be TRUE or FALSE and the code is executed once already.

Output of the Program

```
Value of count is : 0

Value of count is : 1

Value of count is : 2

Value of count is : 3

Value of count is : 4

Value of count is : 5
```

```
Value of count is : 6

Value of count is : 7

Value of count is : 8

Value of count is : 9

Value of count is : 10

Statement processed.
```

14. Numeric FOR Loop

A FOR LOOP is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. It's called Numeric because it requires an integer as its terminating point. Iteration count is by default one and it can't be changed in the body of the For Loop. Starting value (lower_limit) in the loop should be less then end value (upper_limit) for loop to continue. The basic syntax is given below:

```
FOR counter IN [Reverse]
   lower_limit..upper_limit

LOOP
   --Value of counter can't be changed in the body
   -- increment of one in counter is fixed statements;

END LOOP;
```

15. Implementing Numeric FOR Loop – I.

```
DECLARE

  i NUMBER:= 100;

BEGIN

FOR i IN 1..10 LOOP

dbms_output.put_line ( ' Value of I : ' ||i);

i:=i+1; --Error

 END LOOP;

end;
```

Explanation:

Code is trying to change the value of the For Loop counter (i) in the body of the code, in PL/SQL For Loop Counter is not allowed to be changed during the body of the code.

Output of the Program:

```
ORA-06550: line 6, column 1: PLS-00363: expression 'I' cannot be used as an assignment target
ORA-06550: line 6, column 1: PL/SQL: Statement ignored 4. FOR i IN 1..10 LOOP 5.
dbms_output.put_line ( ' Value of I : ' ||i); 6. i:=i+1; --Error 7. END LOOP; 8. end ;
```

16. Implementing Numeric FOR Loop – II

```
DECLARE
  start_val NUMBER: = 10;
  end_val NUMBER := 0;
  i number:=0;

BEGIN
  FOR i IN start_val..end_val LOOP
  dbms_output.put_line ('Value of i : '|| i);

END LOOP;

END;
```

Explanation:

The control will not enter in the loop because to enter in first iteration start_val (10) should be less than end_val (0) which is FALSE here.

Output of the Program:

Statement processed.

17. Implementing Numeric For Loop – III.

To do task:

Write a PL/SQL block to insert rows in temp table, if the row inserted is odd then store message 'Value is odd' and if the row added is even then display message 'Value is even'

```
DECLARE
  x NUMBER: = 100;
  i number:=0;

BEGIN
  FOR i IN 1..10 LOOP
  IF MOD (i,2) = 0 THEN     -- i is even
  dbms_output.put_line('Value of i is even');
  INSERT INTO temp VALUES (i, x, 'i is even');
```

```
ELSE
   INSERT INTO temp VALUES (i, x, 'i is odd');
   dbms_output.put_line('Value of i is odd');

END IF;
   x := x + 100;

END LOOP;

COMMIT;

END;
```

Output of the Program:

```
Value of i is odd

Value of i is even

Value of i is odd

Value of i is even

Value of i is odd

Value of i is even

Value of i is odd

Value of i is even

Value of i is odd

Value of i is even
```

18. FOR Loop with Reverse Option

By default, iteration proceeds from the initial value to the final value, generally upward from the lower bound to the higher bound. You can reverse this order by using the REVERSE keyword. In such case, iteration proceeds the other way. After each iteration, the loop counter is decrease by one. Value of upper limit should be greater than lower limit to continue and again iteration count can't be changed in the body of the For Loop.

19. Implementing Numeric Reverse FOR Loop – I

```
DECLARE
   loop_start Integer: = 1;

BEGIN
   FOR i IN REVERSE loop_start..5
```

```
LOOP
   DBMS_OUTPUT.PUT_LINE ('Loop counter is ' || i);

END LOOP;

END;
```

Output of the Program

```
Loop counter is 5

Loop counter is 4

Loop counter is 3

Loop counter is 2

Loop counter is 1
```

20. Implementing Numeric Reverse FOR Loop – II

```
DECLARE 2
   loop_start Integer := 5;

BEGIN
   FOR i IN REVERSE loop_start..1

LOOP
   DBMS_OUTPUT.PUT_LINE('Loop counter is ' || i);

END LOOP;

END;
```

Explanation:

```
 Value of loop_start is greater than 1, control will not enter in the loop because loop_start
   should be less than 1 to enter in the loop.
```

Output of the Program

```
Statement processed.
```

21. Implementing Reverse FOR Loop with SQL- I

To do task:

Write a PL/SQL block to insert rows in temp table, if the row inserted is odd then store message ‘Value is odd’ and if the row added is even then display message ‘Value is even’

```
DECLARE
   x NUMBER := 100;
   i number:=0;
```

```
BEGIN
   FOR i IN REVERSE 1..10 LOOP
   IF MOD(i,2) = 0 THEN     -- i is even
   dbms_output.put_line('Value of i is even‘ || i);
    INSERT INTO temp VALUES (i, x, 'i is even');

ELSE
   INSERT INTO temp VALUES (i, x, 'i is odd‘);
   dbms_output.put_line('Value of i is odd');

END IF;
   x := x + 100;

END LOOP;

COMMIT;

END;
```

Output of the Program:

Note: Make sure temp table empty otherwise there is possibility there Unique Constraint is violated and exception is thrown.

```
Value of i is even10

Value of i is odd9

Value of i is even8

Value of i is odd7

Value of i is even6

Value of i is odd5

Value of i is even4

Value of i is odd3

Value of i is even2

Value of i is odd1
```

Module 10: Iterative Control – II

1. Continue Statement

Continue statement is used to Conditionally Exit from current iteration of loop, the statements after continue statement are skipped and control is transferred to next iteration. In other words, it forces the next iteration of the loop to take place, skipping any code in between.

2. Implementing Continue with Basic Loop

Following piece of coder serves as a simple example to illustrate the implementation of Continue Statement with Basic Loop

```
DECLARE
   X NUMBER := 0;

BEGIN

LOOP
   DBMS_OUTPUT.PUT_LINE ('Inside loop: x = ' || TO_CHAR(x));
   x := x + 1;
   IF x < 3 THEN

CONTINUE;

END IF;
   DBMS_OUTPUT.PUT_LINE ('Inside loop, after CONTINUE: x = ' || TO_CHAR(x));

 EXIT WHEN x = 5;

END LOOP;
   DBMS_OUTPUT.PUT_LINE (' after loop: x = ' || TO_CHAR(x));

END;
```

Output of the Program

```
Inside loop: x = 0

Inside loop: x = 1

Inside loop: x = 2

Inside loop, after CONTINUE: x = 3

Inside loop: x = 3

Inside loop, after CONTINUE: x = 4

Inside loop: x = 4

Inside loop, after CONTINUE: x = 5
```

```
After loop: x = 5
```

3. Continue When Statement

Continue-When statement is also use to unconditionally exit from current iteration of the loop. As in the case of continuous statement, statements after continues-when statement is also skipped and the control is transferred to the next iteration. Following example would further elaborate the concept.

4. Implementing Continue WHEN with Loop

Following piece of coder serves as a simple example to illustrate the implementation of Continue-When Statement with Loop

```
DECLARE
   x NUMBER := 0;

BEGIN

LOOP
   DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
   x := x + 1;
   CONTINUE WHEN x < 3;
   DBMS_OUTPUT.PUT_LINE
   ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));

EXIT WHEN x = 5;

END LOOP;
   DBMS_OUTPUT.PUT_LINE (' After loop:  x = ' || TO_CHAR(x));

END;
```

5. Implementing Continue WHEN with For Loop

Following piece of coder serves as a simple example to illustrate the implementation of Continue-When Statement with FOR Loop

```
DECLARE
   val number(3):=3;

BEGIN
   FOR i IN 1 .. 10 LOOP
    dbms_output.put_line('i=' || TO_CHAR(i));
   CONTINUE WHEN (i+1) = val;
   dbms_output.put_line('Did not jump to the top of the loop');

END LOOP;

END;
```

6. Nested Loops

Loop can be nested with any other or same type of loops. A nested loop is a loop within a loop, an inner loop within the body of an outer one. How this works is that the first pass of the outer loop triggers the inner loop, which executes to completion. Then the second pass of the outer loop triggers the inner loop again. This repeats until the outer loop finishes. Of course, a break within either the inner or outer loop would interrupt this process.

7. Syntax of Nested Loops

```
LOOP – Main Loop
   Sequence of statements1
   LOOP – Nested or inner loop
    Sequence of statements2

END LOOP;

END LOOP;

END
```

8. Implementing Nested Loops – I

```
BEGIN
   FOR v_outerloopcounter IN 1..2
   LOOP
   FOR v_innerloopcounter IN 1..4
   LOOP
   DBMS_OUTPUT.PUT_LINE('Outer Loop counter is ' ||  v_outerloopcounter);

DBMS_OUTPUT.PUT_LINE(' Inner Loop counter is ' || v_innerloopcounter);



END LOOP;

END LOOP;

END;
```

Explanation:

For every iteration of outer loop all the iterations of inner loop will be executed.

Output of the Program

Outer Loop counter is 1

Inner Loop counter is 1

Outer Loop counter is 1

```
 Inner Loop counter is 2
Outer Loop counter is 1
 Inner Loop counter is 3
Outer Loop counter is 1
 Inner Loop counter is 4
Outer Loop counter is 2
 Inner Loop counter is 1
Outer Loop counter is 2
 Inner Loop counter is 2
Outer Loop counter is 2
 Inner Loop counter is 3
Outer Loop counter is 2
 Inner Loop counter is 4
Statement processed.
```

9. Implementing Nested Loops – II

```
DECLARE
   v_counter1  INTEGER:=0;
   v_counter2  INTEGER:=0;

BEGIN
   WHILE v_counter1 < 3
   LOOP
   DBMS_OUTPUT.PUT_LINE('v_counter1 : ' || v_counter1);
   LOOP
   DBMS_OUTPUT.PUT_LINE('v_counter2: ' || v_counter2);
   v_counter2 := v_counter2 + 1;
   v_counter2 := v_counter2 + 1;
   EXIT WHEN v_counter2 >= 2;

END LOOP;
   v_counter1 := v_counter1+1;

END LOOP;

END;

Output of the Program

v_counter1 : 0
```

```
v_counter2: 0
v_counter1 : 1
v_counter2: 2
v_counter1 : 2
v_counter2: 4
```

Module 11: Cursor

1. Introduction to Cursors

For Oracle to process a SQL statement it needs to create an area of memory known as the context area; this will have the information needed to process the statement. A cursor is a shared memory area in RAM to handle multiple rows which are returned by execution of SQL query. Through the cursor, a PL/SQL program can control the context area and what happens to it as the statement is processed. Cursor contains information to process statements and it is also use to handle multiple rows in SQL.

2. Cursor and SQL

Without cursor SQL SELSCT statement will run only one row and in order to run more than one rows, cursors comes in as a solution because a cursor is construct to handle multiple rows return from select statement. In SQL procedures, a cursor makes it possible to define a result set (a set of data rows) and perform complex logic on a row by row basis.

3. Types of Cursor

There are following two types of cursor along with their details:

A. Implicit:

This cursor is created automatically by oracle server whenever SQL statements are executed and the user is unaware of this and cannot control or process the information in an implicit cursor.

B. Explicit

Explicit cursors are programmer defined cursors for gaining more control over the context area. An explicit cursor should be defined in the declaration section of the PL/SQL Block. The advantage of declaring an explicit cursor over an indirect implicit cursor is that the explicit cursor gives the programmer more programmatic control.

4. Cursor Attributes

There are four attributes of cursors and these are explained in the following table:

No.	Attribute	Syntax	Description
1	%NOTFOUND	cursor_name%NOTFOUND	A Boolean attribute that returns TRUE if the previous FETCH did not return a row and FALSE if it did.
2	%FOUND	cursor_name%FOUND	A Boolean attribute that returns TRUE if the previous FETCH returned a row and FALSE if it did not.
3	%ROWCOUNT	cursor_name%ROWCOUNT	The number of records fetched from a cursor at that point in time.
4	%ISOPEN	cursor_name%ISOPEN	A Boolean attribute that returns TRUE if the cursor is open and FALSE if it is not.

5. Steps to process Cursor

Following steps are involved in processing the cursor:

- *Step 01:* Declaring the cursor for initializing in the memory [in DECLARE section]
- *Step 02:* Opening the cursor for allocating memory [in BEGIN section]
- *Step 03:* Fetching the cursor for retrieving data [in BEGIN section]
- *Step 04:* Closing the cursor to release allocated memory [in BEGIN section]

6. Processing Implicit Cursor – I

```
DECLARE
   total_rows number(2):=0;

BEGIN
   delete from temp;
   IF sql%notfound THEN
   total_rows := sql%rowcount;
   dbms_output.put_line(sql%rowcount  ||' Row is deleted');
   ELSIF sql%found THEN
   total_rows := sql%rowcount;
   dbms_output.put_line( total_rows || ' Rows are deleted ');

END IF;
   IF total_rows=0 then
   INSERT INTO temp values (total_rows, 1, 'Row is inserted');
   dbms_output.put_line(sql%rowcount || '  Row is inserted');

END IF;

END;
```

Explanation:

After execution of delete statement, %notfound will return the count of total number of rows delete, if there are one or more rows got delete then %notfound will be evaluated to false and control will be passed to ELSIF.

Output of the Program:

```
10 Rows are deleted
```

7. Processing Implicit Cursor – II

```
DECLARE
   total_rows number (2):=0;

BEGIN
   UPDATE emp set sal = sal + sal * 0.15 where sal <3000;
   dbms_output.put_line (sql%rowcount || ' Row is Updated');
```

```
   DELETE From temp;
   dbms_output.put_line (sql%rowcount || ' Row are deleted');

END;
   dbms_output.put_line( total_rows || ' Rows are deleted ');

END IF;
   IF total_rows=0 then
   INSERT INTO temp values (total_rows, 1, 'Row is inserted');
   dbms_output.put_line(sql%rowcount || '  Row is inserted');

END IF;

END;
```

Explanation:

Implicit cursor is attached with every DML statement which get executed and all the attributes of implicit cursor are attached to lastly executed DML statement. After execution update statement in the code %rowcount will show total number of rows effected by update and same is true for delete. The code will insert row in temp table always because value of total_rows is always 0 in the code.

Output of the Program:

```
1 Row is Updated

0 Row are deleted

0 Rows are deleted

1 Row is inserted
```

8. Processing Explicit Cursor – I

```
DECLARE
   cursor c1 is select ename, job from emp where deptno = 10;
   name emp.ename%type;
   designation emp.job%type;

BEGIN
   open c1;

LOOP
   fetch  c1 into name, designation;
   dbms_output.put_line('Name : ' || name || ' Designation : ' || designation);
   dbms_output.put_line(c1%rowcount || ' Row are processed ');
   EXIT when c1%notfound;
```

```
END LOOP;
  dbms_output.put_line(c1%rowcount || ' Outside loop Row are processed ');

END;
```

Explanation:

Cursor is defined in Declaration section, Cursor is a name attached with SELECT statement. When cursor will be open in Begin section it will get two column (ename, job) and all the rows where deptno=10 and there can be multiple rows. Without Cursor it is not possible that SELECT statement can return multiple rows but with cursor it is possible. When first row will be fetched from HardDisk to RAM C1 will be pointing to first row in RAM using fetch keyword. In the loop value of first row i-e two columns; are loaded into PL/SQL local variable name and designation. Then name and designation values are displayed on screen. C1%rowcount is counter which contain value of total number currently fetched. For first iteration c1%rowcount is 1 as shown in the output below and for next iteration there will be addition of 1 into it. In the exit condition c1%notfound will be evaluated to FALSE because there is next row in the cursor and control will be transferred back at the start of the loop and C1 will start pointing to next row in RAM and same process will be repeated. When c1 will be at last row in the RAM c1%notfound will be evaluated to TRUE because there will be no next row and loop will terminate.

Output of the Program:

```
Name: KING Designation: PRESIDENT

1 Row are processed

Name : CLARK Designation : MANAGER

2 Row are processed

Name : MILLER Designation : MT

3 Row are processed

Name : Special Designation :

4 Row are processed

Name : Special Designation :

4 Row are processed

4 Outside loop Row are processed

Statement processed.
```

Note: The output of the program may vary depending on data available in emp table of student account

9. Processing Explicit Cursor – II

```
DECLARE
   cursor c1 is select hiredate, deptno,job from emp where job like ('%MAN%');
   doh emp.hiredate%type;
   dno emp.deptno%type;
   jd emp.job%type;

BEGIN
   open c1;

LOOP
   fetch c1 into doh,dno,jd;
   dbms_output.put_line('Date of hiring: ' || doh );
   dbms_output.put_line('Department no: ' || dno );
   dbms_output.put_line('Job: ' || jd );
   EXIT WHEN c1%notfound;

END LOOP;

END;
```

Output of the Program

```
Date of hiring: 05/01/1981

Department no: 30

Job: MANAGER

Date of hiring: 06/09/1981

Department no: 10

Job: MANAGER

Date of hiring: 04/02/1981

Department no: 20

Job: MANAGER

Date of hiring: 02/20/1981

Department no: 30

Job: SALESMAN

Date of hiring: 02/22/1981

Department no: 30

Job: SALESMAN

Date of hiring: 09/28/1981
```

```
Department no: 30

Job: SALESMAN

Date of hiring: 09/08/1981

Department no: 30

Job: SALESMAN

Date of hiring: 09/08/1981

Department no: 30

Job: SALESMAN
```

Note: The output of the program may vary depending on data available in emp table of student account

10. Processing Explicit Cursor – III

To do task:

Write a PL/SQL to display 4 records from emp table and if these 4 records have maximum salary among them display maximum salary.

Solution:

```
DECLARE
   cursor c1 is select  sal from emp;
   max_sal number(10):=0;
   sal emp.sal%type;

BEGIN
   SELECT MAX (sal) into max_sal from emp;
   open c1;

LOOP
   fetch c1 into sal;
   IF sal = max_sal then
   dbms_output.put_line ('Maximum salary is reached');

ELSE
   dbms_output.put_line ('Maximum salary Not reached');

END IF;
   EXIT WHEN c1%rowcount >4;

END LOOP;
```

```
END;
```

Output of the Program

```
Maximum salary Not reached
Maximum salary Not reached
Maximum salary Not reached
Maximum salary Not reached
Maximum salary Not reached
Statement processed.
```

Note: The output of the program may vary depending on data available in emp table of student account

11. Processing Explicit Cursor – IV

To do task:

Write a PL/SQL statement to name of employee having bottom 4-salary respectively.

Solution:

```
DECLARE
  cursor c1 is select ename, sal from emp order by sal;
  name emp.ename%type;
  salary emp.sal%type;

BEGIN
  open c1;

LOOP
  fetch c1 into name, salary;
  dbms_output.put_line('Name : ' || name);
  dbms_output.put_line('Salary : ' || salary);
  EXIT WHEN c1%rowcount > 4;

END LOOP;

END;
```

Explanation:

In the cursor Name and Sal are retrieved from emp table but in Ascending order and then in the loop only first 4 rows are displayed using check on c1%rowcount>4.

Output of the Program

```
Name : Special

Salary : 2783

Name : ADAMS

Salary : 8513.197

Name :

Salary : 9680

Name :

Salary : 9680

Name : MILLER

Salary : 9993.4021
```

Note: The output of the program may vary depending on data available in emp table of student account

12. Processing Explicit Cursor – V

```
DECLARE
   cursor c1 is
   SELECT deptno, sal, ename
   FROM emp
   WHERE job = 'ANALYST';
   total_val number(10):=0;

BEGIN
   total_val := 0;
   FOR employee_rec in c1

LOOP
   total_val := total_val + employee_rec.sal;
   dbms_output.put_line ('Total Salary : ' || total_val);

END LOOP;

END;
```

Output of the Program

```
Total Salary : 15064

Total Salary : 29143


Statement processed.
```

13. Processing Explicit Cursor – VI

```
DECLARE
   CURSOR c1 IS
   SELECT ename, job FROM emp
   WHERE job  LIKE '%CLERK%' AND mgr > 120;

BEGIN
   FOR item IN c1

LOOP
   DBMS_OUTPUT.PUT_LINE
   ('Name = ' || item.ename || ', Job = ' || item.job);

END LOOP;

END;
```

Explanation:

Cursor is open, fetched and closed using FOR Loop. In the FOR loop to process Cursor only declaration in declare section is required rest of the steps are done automatically by FOR Loop.

Output of the Program

Statement processed.

14. Processing Explicit Cursor with SQL – I

To do task:

Write a PL/SQL block to update the salary of all those employees who are working in organization for at least 10 years, increase their salary by 20% and display empno, update salary and currentdate

Solution:

```
DECLARE
   cursor c1 is select empno, sal from emp where months_between(sysdate, hiredate)>120;
   eno emp.empno%type;
   salary emp.sal%type;
   update_sal number(10):=0;

BEGIN
   open c1;

LOOP
   fetch c1 into eno, salary;
   update_sal:=salary+salary*0.20;
```

```
  update emp set sal = update_sal where
  empno=eno;
  dbms_output.put_line('Employee no: ' || eno );
  dbms_output.put_line('Updated Salary: ' || update_sal);
  EXIT WHEN c1%notfound;

END LOOP;

END;
```

15. Processing Explicit Cursor with SQL- II

To do task:

Write a PL/SQL block to insert a row a temp table if sum of salary exceeds 20000 , insert current sum of salary, total employees having sum of 20000 salary and current date, otherwise display salary of current employee and name of employee.

Solution:

```
DECLARE
  cursor c1 is select ename, sal from emp;
  name emp.ename%type;
  salary emp.sal%type;
  total number(10):=0;
  rows number(10):=0;

BEGIN
  open c1;

LOOP
  fetch c1 into name, salary;
  total:=total+salary;
  rows:=c1%rowcount;
  IF (total > 20000) then
  INSERT INTO temp values (total, rows, sysdate);
  dbms_output.put_line('Total salary : ' || total );
  dbms_output.put_line('Total Employees  : ' || rows );
  dbms_output.put_line('Current Date : ' || sysdate );

END IF;
  EXIT WHEN c1%notfound;

END LOOP;
  close c1;

END;
```

Module 12: Error Handling & Built-in Exceptions

1. Error vs. Exception

Errors can be defined as compile time syntax issues while run-time error are defined as exceptions. In other words, an exception is a PL/SQL error that is raised during program execution. Exceptions can be internally defined (by the run-time system) or user defined. PL/SQL provide solution to handle such conditions using EXCEPTION block in the program and an appropriate action is taken against the error condition.

2. Need for Exception Handling

Exception cause the program to terminate the program abnormally and that is not practically possible to foresee all such problematic events. Even the best programmer can have bugs in code written by him. Whenever an exception is raised, flow of the program breaks and the control is lost from the developer view point. So it's important to take precautionary measures in advance to avoid such happenings.

3. Handling Exceptions

As mentioned above, PL/SQL provides a solution to handle exceptions by using EXCEPTIONS block in the program. Using this, run-time errors can be handled to avoid abnormal termination of the program. The syntax if EXCEPTION block is coming in the following lines. Before that its important to see how exception handling works.

4. How Exceptions Handlin Works

Whenever a run-time error is generated, an exception is raised. The flow or the normal execution of the program stops. Control is transferred to the EXCEPTION block of the PL/SQL where the necessary actions are defined to handle exception. Internal exceptions are raised implicitly (automatically) by the run-time system. User-defined exceptions must be raised explicitly by RAISE statements, which can also raise predefined exceptions.

5. Types of Exceptions

There are two types of Exceptions. Details are as follows:

- *Internal Exceptions:* These are pre-defined and they are executed in case of violations of any database rule by the program. Internal exceptions are raised automatically.
- *User-Defined:* PL/SQL also allows user to define their own exceptions according to the need of the program. This type of exceptions is not raised automatically and user should raise it.

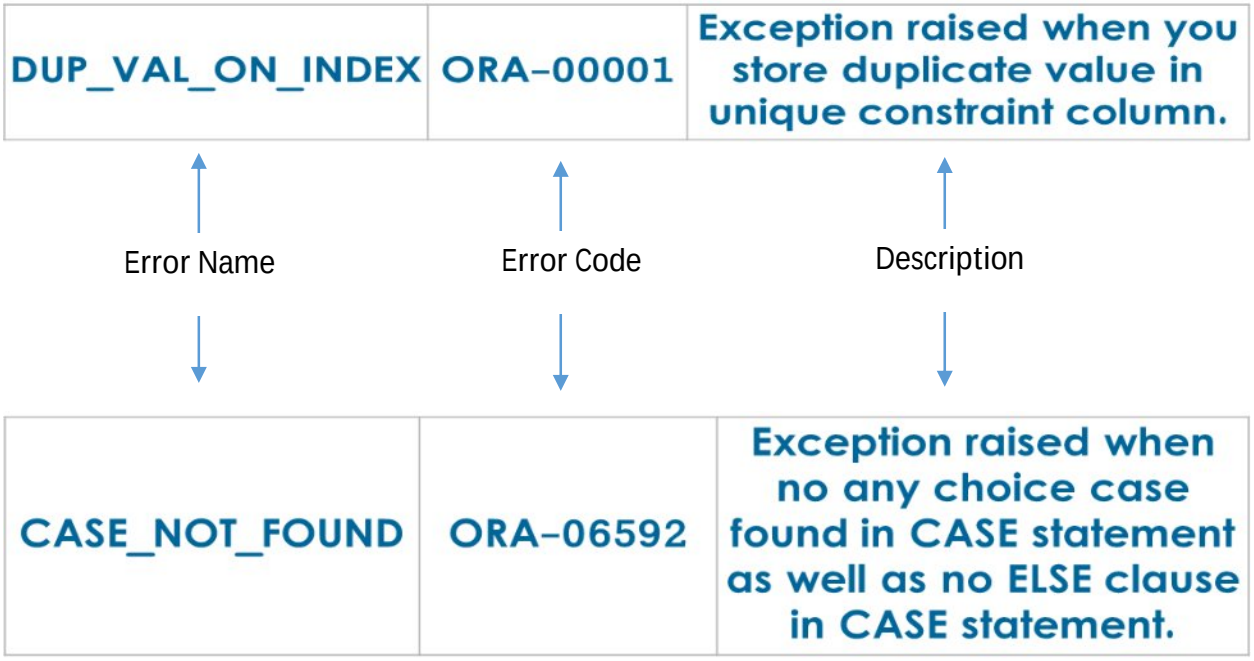
6. Formation of Built-In Exceptions

There are three (03) components of a Built-In Exception:

- Exception Name
- Exception Unique Number: These are system generated

- Description

Consider the following examples:



7. Syntax of Exception

Below is the basic syntax of an exception:

```
Declare Section
   All the declarations

Begin
   Executable statements

Exception Section – One per block
   Exception handling statements

End;
```

Things to be kept in mind:

- One Exception section per block
- Nested blocks can have separate exception section

8. Scope of Exception

```
Declare Section
   All the declarations
```

```
Begin
   Executable statements

Exception handling statements
   declare – Inner block
         begin
         exception – local to innerblock

End;

Exception Section – Global exception

End;
```

9. Implementing Build-In Exception – I

```
Declare
   name varchar2 (30):='';

Begin
   SELECT ename into name
   FROM emp
   WHERE empno=9882;
   dbms_output.put_line(Name);

Exception
   WHEN no_data_found then
   dbms_output.put_line('Record not matched');
   WHEN others then
   dbms_output.put_line(SQLCODE);
   dbms_output.put_line(SQLERRM);

End;
```

Explanation:

When Query is executed in begin block and there is no matching record in emp table then no_data_found exception is raised automatically by the server, in the exception section user friendly message is displayed by handling the particular exception. If there is any other type of exception is raised then WHEN others THEN will take care of it. SQLCODE will return code of the exception raised and SQLERRM will raised the associated message with it.

Output of the Program

```
Record not matched
```

10. Implementing Built-In Exception and SQL-II

```
Declare
   name varchar2 (30):='';

Begin
   INSERT INTO emp(empno) values (7369);
   dbms_output.put_line(Name);

Exception
   WHEN no_data_found then
   dbms_output.put_line('Record not matched');
   WHEN DUP_VAL_ON_INDEX then
   dbms_output.put_line('Unique Key Violated');
   WHEN others then
   dbms_output.put_line(SQLCODE);
   dbms_output.put_line(SQLERRM);

End;
```

Explanation:

Exception can be raised with Query and any DML statement like Insert in this code.

Output of the Program

Unique Key Violated

11. Implementing Built-In Exception and SQL- III

```
Declare
   name varchar2(30):='';

Begin
   SELECT ename into name from emp;

Exception
   WHEN no_data_found then
   dbms_output.put_line('Record not matched');
   WHEN DUP_VAL_ON_INDEX then
   dbms_output.put_line('Unique Key Violated');
   WHEN others then
   dbms_output.put_line(SQLCODE);
   dbms_output.put_line(SQLERRM);

End;
```

Explanation:

Select is returning more than one row without using cursor, multiple rows return exception will be raised and control will be transferred to WHEN others THEN.

Output of the Program

-1422

ORA-01422: exact fetch returns more than requested number of rows

12. Implementing Nested Exceptions – I

```
Declare
  name varchar2(30);

Begin
  SELECT ename into name from emp where empno=7369;
  dbms_output.put_line(name);

        Begin
         INSERT INTO emp (empno) values(7369);
         dbms_output.put_line(name);

Exception
  WHEN dup_val_on_index then
  dbms_output.put_line('Duplicated values in inner block');
  when others  then
  dbms_output.put_line('In inner block');
  dbms_output.put_line(SQLCODE);
           end;

Exception
  WHEN NO_data_found then
  dbms_output.put_line('No Data Found in outer Block');
  WHEN others  then
  dbms_output.put_line('No Data found in outer block');

End;
```

Explanation:

Local exception (inner block) always overrides exception defined in the outer block. If there is no matching exception handler in the inner block then program look of exception handler in outer block.

Output of the Program

KAMRAN

Duplicated values in inner block

13. Implementing Nested Exceptions – II

```
Declare
   name varchar2(30);

Begin
   SELECT ename into name from emp where empno=7369;
   dbms_output.put_line(name);

         Begin
          select ename into name from emp where empno=7338;
          dbms_output.put_line(name);

Exception
   WHEN INVALID_NUMBER then
   dbms_output.put_line(SQLCODE);

         End;

Exception
   when NO_DATA_FOUND then
   dbms_output.put_line('No Data found in outer block');

End;
```

Output of the Program

KAMRAN

No Data found in outer block

Module 13: User Defined Exceptions

1. What is User Defined Exception?

Any exception defined by the user in order to avoid any specific problematic situation or event are known as User Defined Exceptions. User must have to define them explicitly as they are not available by default in the system. User defined exceptions are also raised by the user (hence they are not automatically raised) as per control of the program. Exceptions can be declared only in the declarative part of a PL/SQL block. Below is the basic syntax of user defined exceptions.

2. Syntax of user Defined Exceptions

The syntax of user defined exception is as follows:

```
DECLARE
   user_define_exception_name EXCEPTION;

BEGIN
   statement(s);
   IF condition THEN
   RAISE user_define_exception_name;

END IF;

EXCEPTION
   WHEN user_define_exception_name THEN
   User defined statement (action) will be taken;

END;
```

3. Implementing User Defined Exception – I

```
Declare
   Invalid_salary exception;
   name emp.ename%type;
   salary emp.sal%type;

Begin
   SELECT name, sal into name, salary from emp where empno=7369;
   IF salary <5000 then
   raise invalid_salary;
   else
   dbms_output.put_line('Valid Salary');
   End if;

Exception
   WHEN invalid_salary then
   dbms_output.put_line('InValid Salary');
```

```
End;

Output of the Program:

Valid Salary
```

4. Implementing User Defined Exception – II

Scenario:

Write a PL/SQL block to insert row in emp table, while inserting data in emp table if salary is less than minimum salary then raise an exception other.

Solution:

```
DECLARE
   low_sal EXCEPTION;
   min_sal NUMBER:= 10000;
   new_sal NUMBER:= 8000;

BEGIN
   INSERT INTO EMP(EMPNO, DEPTNO, SAL)
   VALUES (4000,20,new_sal);
   IF new_sal < min_sal THEN
   RAISE low_sal;

END IF;

EXCEPTION
   WHEN low_sal THEN
   DBMS_OUTPUT.PUT_LINE ('Salary is less than '||min_sal);
   WHEN OTHERS THEN
   DBMS_OUTPUT.PUT_LINE (SQLERRM);

END;

Output of the Program

Salary is less than 10000
```

5. Implementing User Defined Exception – III

```
DECLARE

past_due  EXCEPTION;

acct_num  NUMBER;

BEGIN
```

```
DECLARE
 acct_num NUMBER;


due_date DATE := SYSDATE - 1;
todays_date DATE := SYSDATE;
BEGIN
IF due_date < todays_date THEN
RAISE past_due;
END IF;
END;  -- sub-block ends
EXCEPTION
-- Does not handle raised exception
WHEN past_due THEN
DBMS_OUTPUT.PUT_LINE ('Handling PAST_DUE exception in Global declaration.');
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE ('Could not recognize PAST_DUE_EXCEPTION in this scope.' ||
   SQLCODE);
END;
```

Explanation:

Past_due exception is declared in the outer block but raised in the inner block. Inner block will first search for past_due exception handler in the inner block first and if there is no exception handler then exception handler is searched in the outer block

Output of the Program

```
Handling PAST_DUE exception in Global declaration.
```

Module 14: Advance Exceptions

1. Raise_Application_Error

It's a built-in procedure lets you issue user-defined ORA- error messages from stored subprograms. That way, you can report errors to your application and avoid returning unhandled exceptions. It allows system to terminate any PL/SQL block and behave like pre-define or user define exception.

2. Raise vs Raise_Application_Error

Raise is used to call pre-defined or user-defined exception while on the other hand Raise_application_erorr let the developer show the customized message with number. PL/SQL Block terminates the processing when there is some error. If we want to raise an exception and change the path of processing developer can place RAISE statements. By Raise statement, developer can raise user defined exceptions.

3. Syntax of Raise_Application_Error

The syntax to Pre-define procedure to return user-friendly message back to user is as follows:

```
raise_application_error(
      error_number, message[, {TRUE | FALSE}]);
```

Where,

- Error_number is a negative integer in the range -20000 to -20999
- Message is a character string up to 2048 bytes long

4. Implementing Raise_application_error – I

```
Declare
  name emp.ename%type;
  salary emp.sal%type;

Begin
  select ename, sal into name, salary from emp where empno=7369;
  if salary <5000 then
  Raise_application_error(-20030, 'Invalid salary');
  else
  dbms_output.put_line('Valid Salary');

End if;

Exception
  when others then
  dbms_output.put_line(SQLERRM);

End;
```

```
Output of the Program:

Valid Salary
```

5. Implementing User Defined Exception – II

Consider the following scenario:

Write a PL/SQL block to retrieve the ename, job, mrg and hiredate for a particular empno and make sure data is retrieved without error i-e if any field have null value then raise the following exceptions:

Exception No	Message
-20010	No Name
-20020	No Job
-20030	No Manager
-20040	No Hire Date

```
Declare
   v_ename   emp.ename%TYPE;
   v_job        emp.job%TYPE;
   v_mgr       emp.mgr%TYPE;
   v_hiredate     emp.hiredate%TYPE;
   p_empno emp.empno%type:=7654;

BEGIN
   SELECT ename, job, mgr, hiredate
   INTO v_ename, v_job, v_mgr, v_hiredate FROM emp
   WHERE empno = p_empno;
   IF v_ename IS NULL THEN
   RAISE_APPLICATION_ERROR(-20010, 'No name for ' || p_empno);

END IF;
   IF v_job IS NULL THEN
   RAISE_APPLICATION_ERROR(-20020, 'No job for' || p_empno);


END IF;
   IF v_mgr IS NULL THEN
   RAISE_APPLICATION_ERROR(-20030, 'No manager for ' || p_empno);

END IF;
   IF v_hiredate IS NULL THEN
   RAISE_APPLICATION_ERROR(-20040, 'No hire date for ' || p_empno);

END IF;
   DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' validated without errors');
```

```
EXCEPTION
   WHEN OTHERS THEN
   DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
   DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);

END;

Output of the Program

Employee 7654 validated without errors
```

6. What is exception_init Pragma

The EXCEPTION_INIT pragma associates a user-defined exception name with an error code. Ora – Error no can be intercepted and specific handler can be written. In other words, it allows you to handle the Oracle predefined message by your own message which means you can instruct compiler to associate the specific message to oracle predefined message at compile time. Pragma signifies that it's a compiler directive and it is processed at compile time not run-time.

7. Syntax of Exception_init Pragma

Here is the syntax to associate customize message to pre-defined exceptions:

```
DECLARE
   user_define_exception_name EXCEPTION;

PRAGMA EXCEPTION_INIT(user_define_exception_name,-error_number);

BEGIN
   statement(s);
    IF condition THEN
   RAISE user_define_exception_name;

END IF;

EXCEPTION
   WHEN user_define_exception_name THEN User defined statement (action) will be taken;

END;
```

8. Implementing Exception_init Pragma – I

```
DECLARE
   salary number;
   FOUND_NOTHING exception;
   Pragma exception_init(FOUND_NOTHING ,100);
```

```
Begin
   SELECT sal in to salary from emp where ename =‘Akbar';
   dbms_output.put_line(salary);

Exception
   WHEN FOUND_NOTHING THEN
   dbms_output.put_line(SQLERRM);

End;
```

Explanation:

User defined Found nothing exception is associated with built-in exception # 100.

Output of the Program

ORA-01403: no data found

9. Implementing Exception_init Pragma – II

```
DECLARE
   deadlock_detected EXCEPTION;
   PRAGMA EXCEPTION_INIT(deadlock_detected, -60);

BEGIN
   dbms_output.put_line('In the begin section');

EXCEPTION
   WHEN deadlock_detected THEN
   dbms_output.put_line('Deadlock is detected');

END;
```

Module 16: Collection

1. What is Collection?

A collection is an ordered group of elements, all of the same type. It is a general concept that encompasses lists, arrays, and other familiar data types. Each element has a unique subscript that determines its position in the collection and data is accessed through index which can be random. Although collection can hold multiple data but still size of collection is dynamic.

2. PL/SQL Tables or Associated Array

Associative arrays are sets of key-value pairs, where each key is unique and is used to locate a corresponding value in the array. The key can be an integer or a string. Assigning a value using a key for the first time adds that key to the associative array. Subsequent assignments using the same key update the same entry. It is important to choose a key that is unique, either by using the primary key from a SQL table, or by concatenating strings together to form a unique value.

Key thing to remember is that size of the PL/SQL Table is dynamic and index is not needed to be numeric only i-e Index can be string also, index can be random also meaning user can assign index number of its own choice.

The collection is indexed using BINARY_INTEGER values, which do not need to be consecutive. Index can be sparse or dense.

3. Syntax of PL/SQL Tables

The syntax is given below:

```
Step-1:

TYPE table_type_name IS TABLE OF datatype [NOT NULL] INDEX BY BINARY_INTEGER;
  Binary_integer: It store less than number

Range: -2147483647 to 2147483647

Step-2:

Variable of type Table_type_name
```

4. Implementing PL/SQL Tables -I

```
DECLARE
  TYPE JobTabTyp IS TABLE OF  emp.ename%type
  INDEX BY BINARY_INTEGER;
  job_tab JobTabTyp;  -- declare local PL/SQL table
  job_title emp.job%TYPE;
  designation varchar2(16):='Prog';
  counter number(10):=0;
```

```
BEGIN
   loop
   job_tab(counter) :=designation;   -- Assigning designation variable to first index of  job_tab

   dbms_output.put_line (job_tab(counter));
   counter:=counter+1;
   exit when counter >10;

End loop;

END;
```

Explanation:

Value of local variable Designation is assigned to every index of PL/SQL Table i-e job_tab

Output of the Program

```
Prog

Prog

Prog

Prog

Prog

Prog

Prog

Prog

Prog

Prog

Prog
```

PL/SQL Tables and Attributes

Attributes are available with PL/SQL tables which make them easy to use. There are total 7 attributes namely EXISTS, COUNT, FIRST, LAST, PRIOR, NEXT, and DELETE. They make PL/SQL tables easier to use and the applications easier to maintain. Among these attributes some needs parameters and some act like a procedure, for example DELETE.

5. Syntax of using PL/SQL Table with attributes

The syntax of using PL/SQl table with attributes is as follows: but this should be kept in mind that any or no attribute can be used with PL/SQL tables

```
plsql_table_name{
   FIRST |
```

```
NEXT |
DELETE[(index[, index])]
EXISTS(index) |
COUNT |
NEXT(index) |
PRIOR(index)}
```

6. PL/SQL Table and First and Next Attribute

First & Last

These are PL/SQL table attributes, which can be appended to the name of a PL/SQL table. FIRST and LAST return the first and last (smallest and largest) index numbers in a PL/SQL table. If the PL/SQL table is empty, FIRST and LAST return nulls. If the PL/SQL table contains only one element, FIRST and LAST return the same index number.

Next

NEXT(n) returns the index number that succeeds index n in a PL/SQL table. Parameter is required in executing next attribute. If n has no successor, NEXT(n) returns a null.

7. PL/SQL Table and Count Attribute

Count is a numeric attribute which return total number of index created. In other words, COUNT returns the number of elements that a PL/SQL table contains and it is useful because index are not sequential by default.

8. PL/SQL Table and Exist Attribute

EXISTS (n) returns TRUE if the nth element in a PL/SQL table exists. Otherwise, EXISTS (n) returns FALSE. You can use EXISTS to avoid the exception NO_DATA_FOUND, which is raised when you reference a nonexistent element.

9. PL/SQL Table and Delete Attribute

It can be used with name of PL/SQL table to delete particular entry from the table. This attribute has three forms. DELETE removes all elements from a PL/SQL table. DELETE (n) removes the nth element. If n is null, DELETE (n) does nothing. DELETE (m, n) removes all elements in the range m..n. If m is larger than n or if m or n is null, DELETE (m, n) does nothing.

10. Implementing PL/SQL Table -II

```
DECLARE
  cursor c1 is select ename from emp where sal <3000;
  type c2 is table of emp.ename%type index by binary_integer;
```

```
 c3 c2;
   counter number(10):=0;

Begin
   for i in c1 loop
   counter: =counter+2;
   c3(counter):=i.ename;
   dbms_output.put_line (c3(counter));

End loop;

End;
```

Explanation:

A cursor is declared to fetch all those ename where salary < 3000. Then PL/SQL Table c2 is declared to hold ename type data and in last step c3 is declared of c2 type. In the begin section for loop is used to process cursor, now the counter of PL/SQL is having increment of 2 instead of 1 i-e index in c3 are created with gap of 2 not 1; this is possible in PL/SQL . Value from cursor is coped into PL/SQL Table and is displayed in next step.

Output of the Program

```
Special

Statement processed.
```

11. Implementing PL/SQL Table -III

```
DECLARE
   cursor c1 is select ename from emp;
   type c2 is table of emp.ename%type index by binary_integer;

  c3 c2;
   counter number(10):=0;
   b number(10):=0;

Begin
   for i in c1 loop
   counter :=counter+2;
   c3(counter):=i.ename;
   --dbms_output.put_line (c3(counter));

End loop;
   dbms_output.put_line ('Value of counter : ' || counter);
   dbms_output.put_line (c3.count());
   while (b<counter) loop
   if c3.exists(b) then
   dbms_output.put_line ('Value exists at Index : '|| b);
```

```
End if;
  b:=b+1;

End loop;

End;
```

Explanation:

The code is extension to previous example, after copying data in the PL/SQL table c3, in the while loop values from the PL/SQL are displayed but since index are not consecutives and c3.exists (b) will return true if index exists at bth location otherwise it will return false and IF will be evaluated to false. For every true value of c3.exists (b) respective value will be displayed on screen.

12. Implementing PL/SQL Table -IV

Scenario

Write a PL/SQL block to load all the Employee names into PL/SQL Table and copy the values in another PL/SQL Table before deleting those values from first PL/SQL Table

```
DECLARE
  cursor c1 is select ename from emp;
  type c2 is table of emp.ename%type index by binary_integer;

  c3 c2;

  c4 c2;
  counter number(10):=0;
  b number(10):=0;

Begin
  for i in c1 loop
  counter :=counter+2;
  c3(counter):=i.ename;
  --dbms_output.put_line (c3(counter));

End loop;
  dbms_output.put_line ('Value of counter : ' || counter);
  dbms_output.put_line (c3.count());
  while (b<counter) loop
  if c3.exists(b) then
  dbms_output.put_line ('Value exists at Index : '|| b);
  c4(b):=c3(b);
  c3.delete(b);
  dbms_output.put_line ('Value is deleted at Index '|| b);
  dbms_output.put_line ('Value copied in new PL/SQL Table '|| c4(b));
```

```
End if;
   b:=b+1;

End loop;

End;
```

13. Implementing PL/SQL Table and SQL- I

Scenario

Write a PL/SQL block to load all the Employee names into PL/SQL Table and insert the values into another table along with employee number and currentdate

```
DECLARE
   cursor c1 is select ename from emp;
   type c2 is table of emp.ename%type index by binary_integer;

c3 c2;

c4 c2;
   counter number(10):=0;
   b number(10):=0;

Begin
   for i in c1 loop
   counter :=counter+2;
   c3(counter):=i.ename;
   --dbms_output.put_line (c3(counter));

End loop;
   dbms_output.put_line ('Value of counter : ' || counter);
   dbms_output.put_line (c3.count());
   while (b<counter) loop
   if c3.exists (b) then
   insert into history values (b,c3(b),sysdate());
   dbms_output.put_line ('Row is inserted using PL/SQL Table');

End if;
   b:=b+1;

End loop;

End;
```

Module 17: Records

1. What is Record?

A record is a group of related data items stored in fields, each with its own name and data-type. Suppose a database have various data about an employee such as name, salary, and hire date. These items are logically related but dissimilar in type. A record containing a field for each item lets you treat the data as a logical unit. Thus, records make it easier to organize and represent information. Records consist of different fields, similar to a row of a database table. A record is a composite data type, which means that it can hold more than one piece of information, as compared to a scalar data type, such as a number or string.

2. Types of Record

There are following three types of Records. Their details are coming below.

- Table Base
- Cursor Base
- User Defined

3. Table Base

A table-based record, or table record, is a record whose structure (set of columns) is drawn from the structure (list of columns) of a table. Each field in the record corresponds to and has the same name as a column in the table. The fact that a table record always reflects the current structure of a table makes it useful when managing information stored in that table.

4. Implementing Table-based Records – I

```
Declare
  emp_rec emp%rowtype;

Begin
  select * into emp_rec from emp where empno=7369;
  dbms_output.put_line (emp_rec.ename || ' ' || emp_rec.job);

Exception
  when no_data_found then
  dbms_output.put_line('No matching record found');

End;
```

5. Implementing Table-based Records - II

```
DECLARE
  emp_rec   emp%ROWTYPE;

BEGIN
  emp_rec.empno := 500;
  emp_rec.ename := 'Special';
```

```
  emp_rec.ename := 'Consultant';
  emp_rec.mgr := 7369;
  emp_rec.hiredate := sysdate;
  emp_rec.sal := 2000;
  emp_rec.comm := NULL;
  emp_rec.deptno := 10;

INSERT INTO emp
  VALUES emp_rec;
  dbms_output.put_line('Record base insertion is done');

END;
```

6. Cursor-based Records

A cursor-based record, or cursor record, is a record whose structure is drawn from the SELECT list of a cursor. RowType in used in Oracle to create Cursor based records. You could declare a cursor record with the same syntax as a table record, but you don't have to match a table's structure. A SELECT statement creates a "virtual table" with columns and expressions as the list of columns. A record based on that SELECT statement allows you to represent a row from this virtual table in exactly the same fashion as a true table record.

7. Implementing Cursor -based Records – I

Scenario

Write a PL/SQL block which should look for first occurrence of employee either with designation of CLERK or salary greater than 3000. Program should exit after this displaying success message.

Solution

```
Declare
  cursor c1 is select * from emp;
  c2 c1%rowtype;

Begin
  open c1;
  loop
  fetch c1 into c2;
  if c2.sal > 2000 or c2.job='CLERK' then
  dbms_output.put_line ('Value is matched');
  exit

End if;
  exit when c1%notfound;

End loop;

End;
```

```
Output of the Program:

Value is matched
```

8. Implementing Cursor -based Records – II

```
Declare
   cursor c1 is select empno,ename,dname from emp e, dept d where e.deptno=d.deptno;
   c2 c1%rowtype;

Begin
   open c1;
   loop
   fetch c1 into c2;
   if c2.dname='SALES' then
   update emp set sal = sal*0.20 + sal where empno=c2.empno;
   dbms_output.put_line ('Salary is updated by 20% of Employee no: '  || c2.empno);

End if;
   Exit when c1%notfound;

End loop;

End;
```

9. User Defined Record

UDR is composite data type and with UDR programmer have control over definition. User can define structure of UDR and variable of UDR type. With the programmer-defined record, you have complete control over the number, names, and datatypes of fields in the record. To declare a programmer-defined record, you must perform two distinct steps:

- Declare or define a record TYPE containing the structure you want in your record.
- Use this record TYPE as the basis for declarations of your own actual records having that structure.

10. Syntax of User Defined Record

The syntax for creating a User Defined Records is given below:

```
Step# 1:
   TYPE <type_name> IS RECORD
   (<field_name1> <datatype1>,
   <field_name2> <datatype2>,
   ... <field_nameN> <datatypeN> );

Step # 2:
   Declare variable of Type_name;
```

- With UDR programmer is able to define its own structure

- Multiple variables of same UDR can be created

11. Implementing User Defined Record – I

Scenario

Write a PL/SQL block to display the information of first 5 employees who are working as MAN in their ename

Solution

```
Declare
   type emp_rec is RECORD (empno emp.empno%type, ename emp.ename%type, sal
   emp.ename%type);
   emp_rec_val emp_rec;
   cursor c1 is select empno, ename, sal from emp where job like ('%MAN%');

Begin
   open c1;
   loop
   fetch c1 into emp_rec_val;
   dbms_output.put_line (emp_rec_val.ename || emp_rec_val.sal );

Exit when c1%rowcount > 5;

End loop;

End;
```

12. Implementing User Defined Record – II

```
Declare
   type emp_rec is RECORD (empno emp.empno%type, ename emp.ename%type, sal
   emp.ename%type);
   emp_rec_val emp_rec;
   cursor c1 is select empno, ename, sal from emp where job like ('%MAN%');

Begin
   open c1;
   loop
   fetch c1 into emp_rec_val.empno, emp_rec_val.ename,emp_rec_val.sal;
   if emp_rec_val.ename ='BLAKE' then
   insert into history values (emp_rec_val.empno, emp_rec_val.ename,sysdate());
   dbms_output.put_line ('Data Added in history table' );

End if;
   exit when c1%rowcount > 5;

End loop;
```

```
End;
```

Module 18: Procedures

1. What is Sub-Program?

A PL/SQL subprogram is a named PL/SQL block that can be invoked with a set of parameters. A subprogram created to perform a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. Subprograms can either be created at schema level, inside a package, or inside a PL/SQL block (which can be another subprogram).

2. What is Procedure?

Procedure is a database object and a type of subprogram which is created to perform a certain task. Such task can be called multiple times to avoid repetition and have efficiency. Procedure is a subprogram that can take parameters and be called. Generally, you use a procedure to perform an action. A procedure has two parts: the specification and the body.

3. Details of Procedure

Programmers can specify the name of the procedure, its parameters, its local variables, and the BEGIN-END block that contains its code and handles any exceptions. You can specify whether the procedure executes using the schema and permissions of the user who defined it, or the user who calls it. It can receive zero or more parameters as an input and it can return value to the calling environment but the returning value is optional. Summarizing:

- Procedure can performs one or more tasks
- Procedure may or may not return value
- Procedures are normally used for executing business logic.

4. Syntax of Creating PL/SQL Procedure

The Syntax of Creating PL/SQL Procedure is as follows:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
   [(parameter_name [IN | OUT | IN OUT] type [, ...])] {IS | AS

}

BEGIN
   < procedure_body >

END procedure_name;
```

As mentioned above it can be called multiple times and different ways are available to compile, debug and run the procedure.

5. Creating First Procedure

Step -1:

```
CREATE OR REPLACE PROCEDURE first_proc
  AS

BEGIN
    dbms_output.put_line('Hello World!');

END;

/

Output of the Program:

Procedure Created

Step-2:

Calling Procedure from another Block:

Begin
  first_proc;

End;


Output of the Program:

Hello World.
```

6. Debugging the Procedure

Procedure is compiled and executed separately. Data Dictionary can be used to identify errors

```
CREATE OR REPLACE PROCEDURE first_proc

AS

BEGIN
    dbms_output.put_line('Hello World!);

END;

/

Output of the Program:

ORA-24344: success with compilation error

To view Errors user_errors data dictionary is to be browsed as below to find out error:

Select * from user_errors where name='FIRST_PROC';
```

7. Implementing Procedure – I

```
CREATE OR REPLACE PROCEDURE name_proc

AS
   cursor c1 is select * from emp where ename like ('%AK%');
   c2 c1%rowtype;

BEGIN
   loop
   fetch c1 into c2;
   dbms_output.put_line (c2.ename);
   exit when c1%notfound;
   end loop;

END name_proc;

Output of the Program:

Procedure Created

/

Calling the Procedure:

Begin
   Name_proc;

End;
```

8. Implementing Procedure – II

```
CREATE OR REPLACE PROCEDURE name_proc

AS
   cursor c1 is select ename from emp where ename like ('%AK%');
   c2 c1%rowtype;

BEGIN
   loop
   fetch c1 into c2;
   dbms_output.put_line (c2.ename);
   exit when c1%notfound;
   end loop;

END name_proc;

/

Calling the Procedure:

Begin
   Name_proc;
```

```
End;
```

9. Procedure and Parameters

The two parameters along with their description are as follows:

Parameter	Description
IN	• An IN parameter lets you pass a value to the Procedure. • It is a read-only parameter. • Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. • If there is no parameter type mention in header then IN is default • If there is no parameter type mention in header then IN is default
OUT	• It returns value to the calling program. • Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it.

10. Dropping a Procedure

A procedure is deleted with the DROP PROCEDURE statement. Procedure name will be deleted from data dictionary automatically. The syntax to a drop a procedure is:

```
DROP PROCEDURE procedure_name;
```

Where,

procedure_name is the name of the procedure that you wish to drop.

11. Implementing IN / OUT parameter – I

Scenario

Write a PL/SQL procedure to find minimum among two values passed to the procedure. Program should display the value of minimum value returned.

```
DECLARE
   a number;
   b number;
   c number;
   PROCEDURE findMin(x IN number, y IN number, z OUT number) IS

BEGIN
   IF x < y THEN
   z:= x; --

ELSE
   z:= y;
   END IF;

END;
```

```
BEGIN
   a:= 23;
   b:= 45;
   findMin(a, b, c); -- Parameter c will received value assigned to x inside procedure .
   dbms_output.put_line(' Minimum of (23, 45) : ' || c);

END;

/
```

Explanation:

Since X is OUT type parameter, value to be return from procedure is copied into x

Output of the Program

Minimum of (23, 45): 23

12. Implementing IN / OUT parameter – II

Scenario:

Write a PL/SQL procedure to adjust the salary of the employee by percentage of the salary provided at run-time. The procedure should return name of the employee which get updated salary. Procedure will receive employeeid and % of salary to be updated.

```
CREATE OR REPLACE PROCEDURE adjust_salary(
   in_employee_id IN emp.ename%TYPE,
   in_percent IN NUMBER, employeename out varchar2

) IS

BEGIN
   -- update employee's salary
   UPDATE emp
   SET sal = sal + sal * in_percent / 100
   WHERE empno = in_employee_id;
   select ename into employeename from
   emp where empno=in_employee_id;
   END;
```

Output of the Program:

Procedure created.

--Calling the Procedure

```
Declare
   id emp.empno%type:=7369;
```

```
   percent number(10):=10;
   name emp.ename%type;

Begin
   adjust_salary (id,percent,name);
   dbms_output.put_line ('Name of Employee with updated salary ' || name );

End;

Output of the Program:

Old sal = 141076, New sal = 155183.6Difference: 14107.6

Name of Employee with updated salary KAMRAN
```

13. Implementing Procedure with parameter – III

Scenario:

Write a PL/SQL procedure to display list of the employee names working in any particular department. Procedure will receive deptno as input only.

```
create or replace PROCEDURE Get_emp_names (
   Dept_num IN NUMBER) IS
   Emp_name      VARCHAR2(10);
   CURSOR        c1  IS
   SELECT Ename FROM Emp
   WHERE deptno = dept_num;

BEGIN
   OPEN c1;
   LOOP
   FETCH c1 INTO Emp_name;
   DBMS_OUTPUT.PUT_LINE(Emp_name);
   EXIT WHEN C1%NOTFOUND;
   END LOOP;
   CLOSE c1;

END;

Output of the Program

Procedure created.

--Calling the Procedure

Begin
```

```
Get_emp_names (10);
End;
Output of the Procedure:
KING
CLARK
MILLER
Special
Special
```

14. Implementing IN / OUT with Multiple OUT values

Scenario

Write a PL/SQL procedure to adjust the salary of the employee by percentage of the salary provided at run-time. The procedure should return name and designation of the employee which get updated salary. Procedure will receive employeeid and % of salary to be updated.

```
CREATE OR REPLACE PROCEDURE adjust_salary(
in_employee_id IN emp.ename%TYPE,
in_percent IN NUMBER, employeename out varchar2, employeedesignation out varchar2
)
IS
BEGIN
UPDATE emp
SET sal = sal + sal * in_percent / 100
WHERE empno = in_employee_id;
select ename, job into employeename , employeedesignation from emp
where empno=in_employee_id;
END;
```

Explanation:

Multiple values to be send as output are copied from ename, job into employeename, employeedesignation i-e both are OUT parameter type. In the calling environment OUT values are received into name and desig variables which are locally defined PL/SQL variable.

Output of the Program

Procedure created

```
Declare
   id emp.empno%type:=7369;
   percent number(10):=10;
   name emp.ename%type;
   Desig emp.job%type;

Begin
   adjust_salary (id,percent,name, desig);
   dbms_output.put_line ('Name of Employee with updated salary ' || name);
   dbms_output.put_line ('Employee Designation : ' || desig);

End ;
```

Module 19: Functions

1. What is Function?

Function is a database object and a type of subprogram which is created to perform a certain task. A function is a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value. The tasks performed by the function can be called multiple times to avoid repetition and have efficiency.

2. Details of Function

The further details about function are as follows:

- It can receive zero or more parameters as an input.
- It can return value to the calling environment.
- Returning value is Mandatory. In other words, Procedure may or may not return value whereas function should return one value
- Function are normally used for computation.

3. Function vs. Procedure

In Function vs. Procedure comparison, there are following points:

- Both are Database Objects.
- Both can receive one or more parameters
- Procedure can performs one or more tasks whereas function performs a specific task.
- Procedure may or may not return value whereas function should return one value.
- Functions are normally used for computation whereas procedures are normally used for executing business logic.

4. Syntax of Creating PL/SQL Function

The syntax to create PL/SQL function is given below:

```
CREATE [OR REPLACE] FUNCTION function_name [ (parameter [,parameter]) ]
   RETURN return_datatype

BEGIN
   < function_body >
   Return value

END function_name;
```

Again,

- It can be called multiple times
- It must return a value

5. Creating First Function

```
CREATE OR REPLACE FUNCTION totalemployees
   RETURN number IS
   total number(2) := 0;

BEGIN
   SELECT count(*) into total
   FROM emp;
   RETURN total;

END;

/
```

Output of the Program:

```
Function created.
```

Calling a Function:

```
DECLARE
   c number(2);

BEGIN
   c := totalemployees(); dbms_output.put_line('Total no. of Employees: ' || c);

END;

/
```

Output of the Program:

```
Total no. of Employees: 46
```

6. Debugging the Function

Function is compiled and executed separately. Data Dictionary can be used to identify errors.

```
CREATE OR REPLACE FUNCTION totalemployees
   RETURN number IS
   total number(2) := 0;

BEGIN
   SELECT count(*) into total
   FROM em;
   RETURN total;

END;
```

```
/
```

Output of the Program:

```
ORA-24344: success with compilation error
```

Debugging the Function:

```
select * from user_errors where name='TOTALEMPLOYEES';
```

7. Dropping a Function

A function is deleted with the DROP FUNCTION statement. Function name will be deleted from data dictionary automatically. The syntax to a drop a function is:

```
DROP FUNCTION function_name;
```

Where,

Function_name is the name of the function that you wish to drop.

8. Function and IN & OUT parameters

There are three type of parameter type in Functions and Procedures. IN and OUT types are already discussed in the Procedure section. Third parameter type is INOUT. Parameter type INOUT can behave as both IN and OUT simultaneously meaning it can used to passed value into the function and same parameter can be used to write out (returning value) value by function

9. Implementing Function using IN Parameter – I

```
CREATE or replace FUNCTION Ask_salary(emp_no IN NUMBER)
  RETURN NUMBER
  IS emp_sal NUMBER(11,2);

BEGIN
  SELECT sal
  INTO emp_sal
  FROM emp
  WHERE empno=emp_no;
  RETURN(emp_sal);

END;
```

Explanation:

This function will receive emo_no as input and will return salary of the employee to the calling environment. The value of emp_no can't be changed in the function body because parameter type of emp_no is IN.

Output of the Program:

```
Function created.

Calling a Function:

Declare
   salary emp.sal%type;

Begin
   salary:=ask_salary(7369);
   dbms_output.put_line(salary);

End;

Output of the Program:

170701.96
```

10. Implementing Function using IN and OUT Parameter–II

```
CREATE or replace FUNCTION Ask_salary(emp_no IN NUMBER, emp_name out varchar2)
   RETURN NUMBER
   IS emp_sal NUMBER(11,2);

BEGIN
   SELECT sal, ename
   INTO emp_sal, emp_name
   FROM emp
   WHERE empno=emp_no;
   RETURN(emp_sal);

END;

Explanation:

   This function will receive emo_no as input and will return ename and return salary of the
   employee to the calling environment. This function is returning multiple values using Function.
   The value of emp_no can't be changed in the function body because parameter type of
   emp_no is IN.




Calling a Function:

Declare
   salary emp.sal%type;
   emp_name varchar2(30);

Begin
   salary:=ask_salary(7369, emp_name);
```

```
  dbms_output.put_line(salary ||emp_name);
  dbms_output.put_line(emp_name);

End;
```

11. Implementing Function using IN OUT – III

```
CREATE OR REPLACE FUNCTION inout_fn (outparm IN OUT VARCHAR2)
  RETURN VARCHAR2 IS

BEGIN
  outparm := 'Coming out';
  RETURN 'return param';
  END inout_fn;

/

DECLARE
  retval VARCHAR2(20);
  ioval  VARCHAR2(20) := 'Going in';

BEGIN
  DBMS_OUTPUT.put_line('In: ' || ioval);
  retval := inout_fn(ioval);
  DBMS_OUTPUT.put_line('Out: ' || ioval);
  DBMS_OUTPUT.put_line('Return: ' || retval);

END;

/
```

Explanation:

In the function outparm is having INOUT parameter type i-e value can be updated during the function body and the same parameter can be used to return value to the calling environment. Ioval is passed from the calling environment, in the body of the function ioval is updated to 'Coming out' and is same value is written back to outparam i-e this is only possible when parameter type is INOUT both.

Output of the Program

```
In: Going in

Out: Coming out

Return: return param
```

12. Using Function and Procedure Together

Scenario

Write a PL/SQL Procedure to add new employee in the Database, Name of Employee and Salary will be passed as parameter to procedure and date of hiring will be inserted in addition to it. A function should return 1 if there is at least one hiring in last 1 month otherwise it should return 0

Solution:

```
Create or replace PROCEDURE hire_employee (name VARCHAR2, salary number) AS
   id number(10):=0;

BEGIN
   select max(empno) into id from emp;
   INSERT INTO emp(empno, ename, sal, hiredate) VALUES (id+1, name, salary, sysdate);

END hire_employee;

Create or replace function check_employee_status
   return number as
   total number(10):=0;

Begin
   select count(*) into total from emp where hiredate = sysdate - 30;
   if total>1 then
   return 1;
   else return 0;
   End if;

End check_employee_status;

Declare
   name varchar2(15):='KAMRAN';
   salary_emp number(10):=15000;
   Result number(2);

Begin
   hire_employee (name, salary_emp);
   result:= check_employee_status();
   if (result=1) then
   dbms_output.put_line ('There is hiring in last month');

Else
   dbms_output.put_line ('There is No hiring in last month');

End if;

End;
```

Module 20: Triggers

1. What is Trigger?

A trigger is a PL/SQL unit stored in a database which if fired when a DML is executed on a table. It comes with enable and disable feature, which means it be can enabled and disabled. Triggers can be invoked or fired repeatedly but one cannot invoke it explicitly as trigger is fired automatically when a attached DML is executed. While a trigger is disabled, it does not fire. The way to create triggers is coming in the following lines.

2. Triggers and Views

Trigger as mentioned above is a database object and it can be defined on the table, view, schema, or database with which the event is associated. Trigger on base-table is fired if DML is issued against View. If the trigger is created on a table or view, then the triggering event is composed of DML statements, and the trigger is called a DML trigger.

3. Rationale for Trigger

Trigger supplement standard Oracle capabilities and trigger can allow time-base insertion in the specific table. Triggers let you customize your database management system. For example, you can use triggers to:

- Automatically generate virtual column values
- Log events
- Gather statistics on table access from audit and security viewpoint.

4. Triggers Type: Row and Statement Level

When you define a trigger, you can specify the number of times the trigger action is to be run. Based on this, there are following two types of Triggers.

Row Level

A row trigger is fired for each row that 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.

Statement Level

A statement trigger is fired once on behalf of the triggering statement, regardless of the number of rows affected by the associated DML, 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.

5. Triggers Type: Before & After

When defining a trigger, you can specify the trigger timing—whether the trigger action is to be run before or after the triggering statement. BEFORE and AFTER apply to both statement and row triggers. BEFORE and AFTER triggers fired by DML statements can be defined only on tables, not on views. However, triggers on the base tables of a view are fired if an INSERT, UPDATE, or DELETE statement is issued against the view. BEFORE and AFTER triggers fired by DDL statements can be defined only on the database or a schema, not on particular tables.

Before: BEFORE triggers run the trigger action before the triggering statement is run. It is execute before the execution of DML statement.

After: AFTER triggers run the trigger action after the triggering statement is run. Trigger is executed after execution of associated DML statement.

6. Syntax of Creating Trigger in PL/SQL

The syntax is as follows: When Clause is used when conditional execution of Trigger is required

```
CREATE [OR REPLACE ] TRIGGER trigger_name
  {BEFORE | AFTER | INSTEAD OF } {INSERT [OR] | UPDATE [OR] | DELETE}
  [OF col_name] ON table_name
  [REFERENCING OLD AS o NEW AS n]
  [FOR EACH ROW]
  WHEN (condition)

Declare

BEGIN
  sql statements

END;
```

7. Creating First Trigger in PL/SQL

Please consider the following scenario in order to understand the concept of triggers:

Write a Trigger to store the previous values of the product if there change in any price of the product. History table should be maintained to keep record of the previous values just like a shadow table.

```
drop table product;
  CREATE TABLE product
  (product_id number(5),
  product_name varchar2(32),
  supplier_name varchar2(32),
  unit_price number(7,2) );


drop table product_price_history;
```

```
CREATE TABLE product_price_history
  (product_id number(5),
  product_name varchar2(32),
  supplier_name varchar2(32),
  unit_price number(7,2) );

CREATE or REPLACE TRIGGER price_history_trigger
  BEFORE UPDATE OF unit_price
  ON product
  FOR EACH ROW

BEGIN
  INSERT INTO product_price_history
  VALUES (:old.product_id,  :old.product_name, :old.supplier_name, :old.unit_price);

END; /

When UPDATE is issued as follow:

UPDATE PRODUCT SET unit_price = 800 WHERE product_id = 1;

Product_Price_history Table is automatically populated

select * from product_price_history;

Select * from User_triggers where trigger_name = 'PRICE_HISTORY_TRIGGER';
```

8. Implementing Trigger in PL/SQL - II

```
CREATE OR REPLACE TRIGGER print_salary_changes
  BEFORE DELETE OR INSERT OR UPDATE ON emp
  FOR EACH ROW
  WHEN (NEW.job <> 'AD_PRES') -- do not print information about President

DECLARE
  sal_diff  NUMBER;

BEGIN
  sal_diff  := :NEW.sal  - :OLD.sal;
  DBMS_OUTPUT.PUT(:NEW.ename || ': ');
  DBMS_OUTPUT.PUT('Old sal = ' || :OLD.sal || ', ');
  DBMS_OUTPUT.PUT('New sal = ' || :NEW.sal );
  DBMS_OUTPUT.PUT('New sal = ' || :NEW.sal || ', ');
  DBMS_OUTPUT.PUT_LINE('Difference: ' || sal_diff);

END;
```

9. Implementing Trigger in PL/SQL - III

Scenario

Write a Trigger which should enforce referential integrity constraint while inserting data in Emp table. Insert or Updating in Emp table should ensure the corresponding deptno in dept table should existence prior to DML statement, if there is no matching parent record in Dept table then DML should not be executed.

```
CREATE OR REPLACE TRIGGER Emp_dept_check
   BEFORE INSERT OR UPDATE OF Deptno ON Emp
   FOR EACH ROW WHEN (new.Deptno IS NOT NULL)

DECLARE
   Dummy INTEGER;  -- to be used for cursor fetch
   Invalid_department EXCEPTION;
   Valid_department   EXCEPTION;
   CURSOR Dummy_cursor  IS
   SELECT Deptno FROM Dept
   WHERE Deptno = :new.Deptno
   FOR UPDATE OF Deptno;

BEGIN
   OPEN Dummy_cursor;
   FETCH Dummy_cursor INTO Dummy;
   IF Dummy_cursor%NOTFOUND THEN
   RAISE Invalid_department;

ELSE
   RAISE valid_department;
   END IF;
   CLOSE Dummy_cursor;

EXCEPTION
   WHEN Invalid_department THEN
   CLOSE Dummy_cursor;
   Raise_application_error(-20000, 'Invalid Department‘ Number' || :new.deptno);
   WHEN Valid_department THEN
   CLOSE Dummy_cursor;

END;
```

Output of the Program

```
Trigger created.
```

```
When Row is inserted into Emp table with invalid deptno (90) as below:


insert into emp (empno, deptno) values(99,90);


Trigger get executed and data is not inserted in Emp table.

ORA-20000: Invalid Department‘ Number90 ORA-06512: at
  "SAMPLEDATABASE.EMP_DEPT_CHECK", line 21 ORA-04088: error during execution of trigger
  'SAMPLEDATABASE.EMP_DEPT_CHECK' 1. insert into emp (empno, deptno) values(99,90);
```

10. Implementing Table Level Trigger in PL/SQL - IV

Scenario

Write a Trigger which should be used to track every insert statement i-e not the data which is inserted. Log table will provide the count along with data / time of every insertion on Emp Table.

```
CREATE TABLE Emp_log ( Log_date DATE, Action CHAR(30) );
  CREATE OR REPLACE TRIGGER Log_emp_update
  AFTER UPDATE ON emp –- Table level

BEGIN
  INSERT INTO Emp_log (Log_date, Action) VALUES (SYSDATE, 'Employee Table Updated');

END;
```

11. Enabling and Disabling the Triggers

As mentioned above, triggers can be disabled and enabled, depending upon the requirements.

- Trigger can be made temporarily ineffective by using disable statement
- Trigger can be made effective by using enable statement

The basic syntax for both is as follows:

```
alter trigger emp_dept_check disable;

alter trigger emp_dept_check enable;
```

12. Dropping the Triggers

Trigger can be made dropped using Drop Trigger Command and in that case Entry from data dictionary will be removed.

The syntax is as follows:

```
drop trigger emp_dept_check;
```

Through this all the entries from the Data Dictionaries will be removed.

Module 21: Package

1. What is package?

It is Database Object which group together different Database Objects. groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions. A package is compiled and stored in the database. Packages offer several advantages: modularity, easier application design, information hiding, added functionality, and better performance. Packages let you encapsulate logically related types, items, and subprograms in a named PL/SQL module. Each package is easy to understand, and the interfaces between packages are simple, clear, and well defined. This aids application development.

2. Parts of package

There are following two parts of a Package:

Package Specification: The specification is the interface to your applications; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The spec holds public declarations, which are visible to the application

Package Body: The body fully defines cursors and subprograms, and so implements the spec. The body holds implementation details and private declarations, which are hidden from your application.

3. Syntax of Package

Syntax for Creating Package is as follows:

```
Step-1:

CREATE PACKAGE Name of Package AS
   Prototype of Database Object (Function, Procedure, Trigger)

End Package Name

Step-2:

CREATE PACKAGE Body Name AS
   Implemtation of Prototype of Database Object (Function, Procedure, Trigger) defined in
   Package Specification
   End Database Object Name

End Package Name
```

Again,

- Package Specification contain definitions only like Global Variables
- Package Body contain implementation