CHAPTER

Five

MANAGEMENT OF SCHEMA OBJECTS AND CONCEPT OF PL/SQL

CHAPTER OUTLINE

5.1 SQL SUB QUERIES WITH EXAMPLES

Subqueries : A subquery (or) innerquery (or) nested query is a query with in another SQL query and embeded within the where clause. It is used to return data that will be used in the main query as a condition to further restricts the data to be retrieved. Subquery can be used with select, insert, update, delete statements along with operators like =, <, >, <=, > =, in between

- Subquery is an approach which provides the capability of embedding the first query into another query.
- Subquery must be enclosed with in parantheses().
- A subquery can have only one column in the select statement unless multiple columns are in main query for the subquery to compare it's selected column.
- An order by clause cannot be used in a subquery although the main query can have order by clause and also group by clause can be used to perform the same function as order by clause in a subquery.
- A subquery that returns more than one row can only be used with multiple value operator such as IN operator.
- The Between operation cannot be used with in subquery.

Sub Queries : A Subquery is a query within another SQL query and embedded within the WHERE clause.

Important Rule :

- A subquery can be placed in a number of SQL clauses like WHERE clause, FROM clause, HAVING clause.
- You can use Subquery with SELECT, UPDATE, INSERT, DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.
- A subquery is a query within another query. The outer query is known as the main query, and the inner query is known as a subquery.
- Subqueries are on the right side of the comparison operator.
- A subquery is enclosed in parentheses.
- In the Subquery, ORDER BY command cannot be used. But GROUP BY command can be used to perform the same function as ORDER BY command.

1. **Subqueries with the Select Statement :** SQL subqueries are most frequently used with the SELECT statement.

Syntax :

```
SELECT column_name
FROM table_name
WHERE column_name expression operator
(SELECT column_name from table_name WHERE ... );
```

EXAMPLE - 1

Consider the EMPLOYEE table have the following records :

CREATE TABLE EMPLOYEE(ID integer,NAME varchar(20), AGE integer,ADDRESS varchar(20), SALARY float);

CREATE TABLE EMPLOYEE_BKP(ID integer,NAME varchar(20), AGE integer,ADDRESS varchar(20), SALARY float);

INSERT INTO EMPLOYEE VALUES(1,'John',20,'US',2000.00);

INSERT INTO EMPLOYEE VALUES(2,'Stephen',26,'Dubai',1500.00);

INSERT INTO EMPLOYEE VALUES(3,'David',27,'Bangkok',2000.00);

INSERT INTO EMPLOYEE VALUES(4,'Alina',29,'UK',6500.00);

INSERT INTO EMPLOYEE VALUES(5,'Kathrin',34,'Bangalore',8500.00);

INSERT INTO EMPLOYEE VALUES(6,'Harry',42,'China',4500.00);

INSERT INTO EMPLOYEE VALUES(7,'Jackson',25,'Mizoram',10000.00);

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
4	Alina	29	UK	6500.00
5	Kathrin	34	Bangalore	8500.00
6	Harry	42	China	4500.00
7	Jackson	25	Mizoram	10000.00

The subquery with a SELECT statement will be :

```
SELECT *
FROM EMPLOYEE
WHERE ID IN (SELECT ID
FROM EMPLOYEE
WHERE SALARY > 4500);
```

This would produce the following result :

ID	NAME	AGE	ADDRESS	SALARY
4	Alina	29	UK	6500.00
5	Kathrin	34	Bangalore	8500.00
7	Jackson	25	Mizoram	10000.00

Subqueries with the INSERT Statement :

- SQL subquery can also be used with the Insert statement. In the insert statement, data returned from the subquery is used to insert into another table.
- In the subquery, the selected data can be modified with any of the character, date functions.

Syntax :

```
INSERT INTO table_name (column1, column2, column3....)
SELECT *
FROM table_name
WHERE VALUE OPERATOR
```

EXAMPLE - 2

Consider a table EMPLOYEE_BKP with similar as EMPLOYEE.

Now use the following syntax to copy the complete EMPLOYEE table into the EMPLOYEE_BKP table.

```
INSERT INTO EMPLOYEE_BKP
SELECT * FROM EMPLOYEE
WHERE ID IN (SELECT ID
FROM EMPLOYEE);
```


Subqueries with the UPDATE Statement : The subquery of SQL can be used in conjunction with the Update statement. When a subquery is used with the Update statement, then either single or multiple columns in a table can be updated.

Syntax :

```
UPDATE table
SET column_name = new_value
WHERE VALUE OPERATOR
(SELECT COLUMN_NAME
FROM TABLE_NAME
 WHERE condition);
```

EXAMPLE - 3

Let's assume we have an EMPLOYEE_BKP table available which is backup of EMPLOYEE table. The given example updates the SALARY by .25 times in the EMPLOYEE table for all employee whose AGE is greater than or equal to 29.

```
UPDATE EMPLOYEE
SET SALARY = SALARY * 0.25
WHERE AGE IN (SELECT AGE FROM EMPLOYEE_BKP
WHERE AGE >= 29);
```

This would impact three rows, and finally, the EMPLOYEE table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
4	Alina	29	UK	1625.00
5	Kathrin	34	Bangalore	2125.00
6	Harry	42	China	1125.00
7	Jackson	25	Mizoram	10000.00

Subqueries with the DELETE Statement : The subquery of SQL can be used in conjunction with the Delete statement just like any other statements mentioned above.

Syntax :

```
DELETE FROM TABLE_NAME
WHERE VALUE OPERATOR
(SELECT COLUMN_NAME
FROM TABLE_NAME
WHERE condition);
```

EXAMPLE - 4

Let's assume we have an EMPLOYEE_BKP table available which is backup of EMPLOYEE table. The given example deletes the records from the EMPLOYEE table for all EMPLOYEE whose AGE is greater than or equal to 29.

```
DELETE FROM EMPLOYEE
WHERE AGE IN (SELECT AGE FROM EMPLOYEE_BKP
WHERE AGE >= 29 );
```

This would impact three rows, and finally, the EMPLOYEE table would have the following records.

ID	NAME	AGE	ADDRESS	SALARY
1	John	20	US	2000.00
2	Stephan	26	Dubai	1500.00
3	David	27	Bangkok	2000.00
7	Jackson	25	Mizoram	10000.00

5.2 SQL JOIN STATEMENTS WITH EXAMPLES

SQL joins : A SQL Join statement is used to combine data or rows from two or more tables based on a common field between them. Different types of Joins are :

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN

Consider the two tables below :

CREATE TABLE EMPLOYEE(EMP_ID integer, EMP_NAME varchar(20), CITY varchar(20), SALARY bigint, AGE integer);

CREATE TABLE PROJECT(PROJECT_NO integer,EMP_ID integer, DEPARTMENT varchar(20));

insert into EMPLOYEE values(1,'Angelina','Chicago',200000,30);

insert into EMPLOYEE values(2,'Robert','Austin',300000,26);

insert into EMPLOYEE values(3,'Christian','Denver',100000,42);

insert into EMPLOYEE values(4,'Kristen','Washington',500000,29);

insert into EMPLOYEE values(5,'Russell','Los angels',200000,36);

insert into EMPLOYEE values(6,'Marry','Canada',600000,48);

insert into PROJECT values(101,1,'Testing');

insert into PROJECT values(102,2,'Development');

insert into PROJECT values(103,3,'Designing');

insert into PROJECT values(104,4,'Development');

EMPLOYEE :

EMP_ID	EMP_NAME	CITY	SALARY	AGE
1	Angelina	Chicago	200000	30
2	Robert	Austin	300000	26
3	Christian	Denver	100000	42
4	Kristen	Washington	500000	29
5	Russell	Los angels	200000	36
6	Marry	Canada	600000	48

PROJECT :

PROJECT_NO	EMP_ID	DEPARTMENT
101	1	Testing
102	2	Development
103	3	Designing
104	4	Development

1. **Inner Join :** In SQL, INNER JOIN selects records that have matching values in both tables as long as the condition is satisfied. It returns the combination of all rows from both the tables where the condition satisfies.

Syntax :

```
SELECT table1.column1, table1.column2, table2.column1,....
FROM table1
INNER JOIN table2
ON table1.matching_column = table2.matching_column;
```

Query :

```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENT
FROM EMPLOYEE
INNER JOIN PROJECT
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

Output :

EMP_NAME	DEPARTMENT
Angelina	Testing
Robert	Development
Christian	Designing
Kristen	Development

2. **Left Join :** The SQL left join returns all the values from left table and the matching values from the right table. If there is no matching join value, it will return NULL.

Syntax :

```
SELECT table1.column1, table1.column2, table2.column1,....
FROM table1
LEFT JOIN table2
ON table1.matching_column = table2.matching_column;
```

Query :

```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENT
FROM EMPLOYEE
LEFT JOIN PROJECT
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

Output :

EMP_NAME	DEPARTMENT
Angelina	Testing
Robert	Development
Christian	Designing
Kristen	Development
Russell	NULL
Marry	NULL

3. **Right Join :** In SQL, RIGHT JOIN returns all the values from the values from the rows of right table and the matched values from the left table. If there is no matching in both tables, it will return NULL.

Syntax :

```
SELECT table1.column1, table1.column2, table2.column1,....
FROM table1
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;
```

Query :

```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENT
FROM EMPLOYEE
RIGHT JOIN PROJECT
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

Output :

EMP_NAME	DEPARTMENT
Angelina	Testing
Robert	Development
Christian	Designing
Kristen	Development

4. **Full Join :** In SQL, FULL JOIN is the result of a combination of both left and right outer join. Join tables have all the records from both tables. It puts NULL on the place of matches not found.

Syntax :

```
SELECT table1.column1, table1.column2, table2.column1,....
FROM table1
FULL JOIN table2
ON table1.matching_column = table2.matching_column;
```

Query :

```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENT
FROM EMPLOYEE
FULL JOIN PROJECT
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

Output :

EMP_NAME	DEPARTMENT
Angelina	Testing
Robert	Development
Christian	Designing
Kristen	Development
Russell	NULL
Marry	NULL

5.3 MANAGEMENT OF SCHEMA OBJECTS

The logical database is determined by schema objects which directly refers to database. A schema is a collection of logical structures of data, or schema objects. Schema objects can be created and manipulated with SQL. The schema objects are :

1. Tables
2. Index
3. Synonyms
4. Sequence
5. View
6. Clusters
7. Triggers
8. Stored functions, procedures and packages.

5.4 MANAGEMENT OF INDEXES

SQL has statements to create and drop indexes on attributes of base relation. These commands are generally considered to be part of the SQL data definition language (DDL).

An INDEX is a physical access structure that is specified on one or more attributes of the relation. The attributes on which an index is created are termed indexing attributes. An index makes accusing tuples based on conditions that involve its indexing attributes more efficient. This means that in general executing a query will take less time if some attributes involved in the query conditions were indexed than if they were not. In general, if attributes used in selection conditions and in join conditions of a query are indexed, the execution time of the query is greatly improved.

In SQL indexed can be created and dropped dynamically. The CREATE INDEX command is used to specify an index. Each index is given a name, which is used to drop the index when we do not need it any more.

Index is SQL is created on existing tables to retrieve the rows quickly. When there are thousands of records in a table retrieving information will take long time.

- Indexes are created on columns which are accessed frequently. So that the information can be retrieved quickly.
- Indexes can be created on single column (or) group of columns.
- When an index is created it first sorts the data and then it assigns a row id for each row.
- Even the SQL indexes are created to access the rows in table quickly, they slowdown when DML operations like insert, update and delete are performed on a table because the indexes and table both are updated, where DML operation is performed.
- We use indexes only on columns which are used to search the table frequently.
- We can create index for up to 16 columns. The user cannot see index. They are
- Just used to speed up search for query.

Types of Index : There are two types of index namely,

1. **Implicit Index :** These are created when a column is explicitly defined as primary key and unique key constraints.
2. **Explicit Index :** They are created by using create index syntax.

Syntax :

```
> CREATE  INDEX     indexname
  ON      tablename (column1, column2 ……);
```

Ex-1:
```
CREATE  INDEX     studentindex
ON      Student   (pin, name);
```

Ex-2:
```
CREATE  INDEX     Emp_Index
ON      Employee  (Emp_name);
```

In general, the index is arranged in ascending order of the indexing attribute values. If we want the values in descending order we can add the keyword DESC after the attribute name. The default is ASC for ascending. We can also create an index on a combination of attributes.

Example :

```
Create  Index     Emp_Index1
ON      Employee  (Emp_name ASC,
                   Comp_name DESC);
```

There are two additional options on indexes in SQL. The first is to specify the key constraint on the indexing attribute or combination of attributes.

The keyword unique following the CREATE command is used to specify a key.

Syntax :

```
> CREATE  UNIQUE    INDEX indexname
  ON      tablename (column1, column2 ……);
```

Ex-1:
```
CREATE    UNIQUE    INDEX Studentindex1
ON        Student   (pin, name);
```

The second option on index creation is to specify whether on index is clustering index. The keyword cluster is used in this case of the end of the create Index command. A base relation can have at most one clustering index but any number of non_clustering indexes.

To drop an index, we issue the DROP INDEX command. The reason for dropping indexes is that they are expensive to maintain whenever the base relation is updated and they require additional storage. However, the indexes that specify a key constraint should not be dropped as long as we want the system to continue enforcing that constraint.

If you want to delete the created index.

```
> DROP  INDEX   indexname;
```

Ex :
```
DROP    INDEX   studentindex;
DROP    INDEX   Emp_Index;
> ALTER TABLE   tablename
DROP    INDEX   indexname;
```

Cluster Index : A cluster index determines the physical order of data in a table. It is analogous to a telephone directory.

- The cluster index defines the physical storage order of data in the table. The table can contain only one cluster index, however index can comprise of multiple columns.
- Cluster index sort and store the data of rows in table based on their keyvalue.
- When a primary key is created a cluster index is automatically created.
- If the table is under heavy data modification, primary key is used to search a cluster index.

Creating a Cluster : The following statement creates a cluster named personal with the clusterkey column department and a cluster size of 512 bytes along with the storage parameters.

Syntax :

```
> CREATE  CLUSTER  clustername (clusterkeycolumn datatype(size)
SIZE value
STORAGE (initial value next value);
```

Ex :
```
CREATE    Cluster     Personal
(department varchar(30))
SIZE 512
STORAGE (INITIAL 100k NEXT 50k);
```

Creating a Cluster Index: The following statement creates the cluster index on cluster personal.

Syntax :

```
> CREATE  INDEX     indexname
  ON      CLUSTER   clustername;
```

Ex : > Create INDEX personalcluster
ON CLUSTER personal;

After creating the cluster index we can add tables to the index and perform DML operations.

Adding table to a Cluster:

Syntax :

> CREATE TABLE tablename
CLUSTER clustername (columnname)
AS
SELECT columnlist FROM
WHERE condition;

Ex : > CREATE TABLE dept10
CLUSTER personal (department)
AS
SELECT * FROM employee
WHERE id = 10;

HASH Cluster : The following statement creates a Hash cluster named language with the clusterkey column clang, a maximum of ten hash key values each of which is allocated 512 bytes along with storage parameters. This is used for faster retrieval of information.

Syntax :

> CREATE CLUSTER language(
Clang varchar(20))
SIZE 512
HASHKEY 10
STORAGE (INITIAL 100k NEXT 50k);

Deleting a Cluster :

> DROP CLUSTER clustername;
> DROP CLUSTER personal;

5.5 MANAGEMENT OF SEQUENCES

The quickest way to retrieve data from a table is to have a column in the table whose data uniquely identifies a row. By using this column and a specific value in the WHERE condition of a SELECT sentence the oracle engine will be able to identify and retrieve the row the fastest.

To achieve this, a constraint is attached to a specific column in the table that ensures that the column is never left empty and that the data values in the column are unique. Since human beings do data entry, it is quite likely that a duplicate value could be entered, which violates this constraint and the entire row is rejected.

If the value entered into this column is computer generated it will always fulfill the unique constraint and the row will always be accepted for storage.

Oracle provides an object called a SEQUENCE that can generate numeric values. The value generated can have a maximum of 38 digits.

A sequence can be defined to :

- Generate numbers in ascending or descending order.
- Provide intervals between numbers
- Caching of sequence numbers in memory to speed up their availability

A sequence is an independent object and can be used with any table that requires its output. A sequence is a database object that generates numbers in sequential order.

Applications : Most often use these numbers when they require a unique value in a table.

Creating Sequences :

Always give sequence a name so that it can be referenced later when required.

The minimum information required for generating numbers using a sequence is:

- The starting number
- The maximum number that can be generated by a sequence
- The increment value for generating the next number

This information is provided to oracle at the time of sequence creation

Syntax :

```
CREATE SEQUENCE <SequenceName>
[INCREMENT BY <IntegerValue>
[START WITH <IntegerValue>
MAXVALUE <IntegerValue> / NOMAXVALUE
MINVALUE <IntegerValue> /NOMINVALUE
CYCLE/NOCYLCLE
CACHE <IntegerValue>/NOCACHE
ORDER/NOORDER ]
```

KEYWORDS AND PARAMETERS

INCREMENT BY : Specifies the interval between sequence numbers. It can be any positive or negative value but not zero. If this clause is omitted, the default value is 1.

MINVALUE : Specifies the sequence minimum value.

NOMINVALUE : Specifies a minimum value of 1 for an ascending sequence and (10)^26 for a descending sequence.

MAXVALUE : Specifies the maximum value that a sequence can generate.

NOMAXVALUE : Specifies a maximum of 10^27 for an ascending sequence or -1 for a descending sequence. This is the default clause.

START WITH : Specifies the first sequence number to be generated. The default for an ascending sequence is the sequence minimum value (1) and for a descending sequence, it is the maximum value (-1)

CYCLE : Specifies that the sequence continues to generate repeat values after reaching either its maximum value.

NOCYCLE : Specifies that a sequence cannot generate more values after reaching the maximum value

CACHE **:** Specifies how many values of a sequence oracle pre-allocate and keeps in memory for faster access. The minimum value for this parameter is two.

NOCHACHE : Specifies that values of a sequence are not pre-allocated.

ORDER : This guarantees that sequence numbers are generated in the order of request. This is only necessary if using parallel server in parallel mode option. In exclusive mode option, a sequence always generates numbers in order.

NOORDER : This does not guarantee sequence numbers are generated in order of request. This is only necessary if you are using parallel server in parallel mode option. If the ORDER/NOORDER clause is omitted, a sequence takes the NOORDER clause by default.

Example :

Create a sequence by the name Sn, which will generate numbers from 1 up to 9999 in ascending order with an interval of 1.The sequence, must restart from the number 1 after generating number 999.

```
> CREATE SEQUENCE Sn
  INCREMENT BY 1
  START WITH 1
  MINVALUE 1
  MAXVALUE 999
  CACHE 20
  CYCLE;
```

Altering the Sequence:

Syntax :

```
ALTER SEQUENCE Sequencename
RESTART WITH value
MINVALUE value | NO MINVALUE
MAXVALUE value | NO MAXVALUE
INCREMENT BY value
CACHE value | NO CACHE
CYCLE value | NO CYCLE;
```

Example :

```
ALTER SEQUENCE Sn
RESTART WITH 50
MINVALUE 50
MAXVALUE 200
```

```
INCREMENT BY 2
NO CACHE
NO CYCLE;
```

Referencing a sequence

Once a sequence is created SQL can be used to view the values held in its cache. To simply view sequence value use a SELECT sentence as described below.

SELECT <SequenceName>.Nextval from DUAL;

This will display the next value held in the cache on the VDU screen. Every time nextval references a sequence its output is automatically incremented from the old value to the new value ready for use.

To reference the current value of a sequence:

SELECT <SequenceName>.CurrVal FROM DUAL;

Dropping a Sequence

The DROP SEQUENCE command is used to remove the sequence from the database.

Syntax :

```
DROP SEQUENCE <SequenceName>;
```

Ex : DROP SEQUENCE Sn;

AUTO-increment: (sequence) (mySQL)

```
> CREATE TABLE producer (
  Id INT NOT NULL AUTO_INCREMENT,
  lname varchar(30) NOT NULL,
  fname varchar(30),
  age INT,
  PRIMARYKEY(id));
```

How to insert value for that particular table?

```
> INSERT INTO producer (lname, fname, age)
  VALUES('Amarendra', 'bahubali', 30);
> INSERT INTO producer (lname, fname, age)
  VALUES('Mahendra', 'bahubali', 60);
> INSERT INTO producer (lname, fname, age)
  VALUES('Balala', 'Deva', 65);
> SELECT * FROM producer;
```

id	lname	fname	age
1	Amarendra	Bahubali	30
2	Mahendra	Bahubali	60
3	Balala	Deva	65

```
> DELETE FROM producer WHERE age = 65;
```

id	lname	fname	age
1	Amarendra	Bahubali	30
2	Mahendra	Bahubali	60

```
> INSERT INTO producer (lname, fname, age)
  VALUES ('deva', 'sena', 28);
> SELECT * FROM producer;
```

id	lname	fname	age
1	Amarendra	Bahubali	30
2	Mahendra	Bahubali	60
4	Deva	Sena	28

5.6 MANAGEMENT OF SYNONYMS

A synonym is simply an Alias (alternative name) for a table, view, sequence, stored procedure function and package. When there is a need to remove an application the synonym can be used.

The synonym plays an important role in distributed databases where applications can reference both local and remote database objects, to accomplish work the usage of synonym hides the location of data structure from application and enables location transperancy.

Creating Synonym :

> CREATE SYNONYM synonymname FOR object name tablename;

Ex : > CREATE SYNONYM vidhyarthi FOR student;

> CREATE SYNONYM worker FOR employee;

Deleting a Synonym :

> DROP SYNONYM synonymname;

Ex : DROP SYNONYM vidhyarthi;

5.7 MANAGEMENT OF VIEWS

A view in SQL terminology is a single table that is derived from other tables. These other tables could be base tables or previously defined views. A view does not necessarily exist in physical form; it is considered a virtual table in contrast to base tables whose tuples are actually stored in the database. This limits the possible update operations that can be applied to views but does not provide any limitations on querying a view. We can think of view as a way specifying a table that we need not exist physically.

A view is a virtual table or logical table. A view is dynamic because it is not related to physical schema. A view is similar to a database table which consists of rows and columns, you can query data against it.

- Views in SQL are kind of virtual tables.
- A view also has rows and columns as they are in a real table in the database.
- We can create a view by selecting fields from one or more tables present in the database.
- A View can either have all the rows of a table or specific rows based on certain condition.
- One of the main reasons for making use of views in an application schema is to isolate application logic from direct dependence on table structure.
- Views are like virtual window through which we can view or change information of a table.

Advantages of view :

- We use the view to hide the complexity of underline table to the end user and external application. i.e., view allows you to simplify complex queries.
- We can use a view to expose only non sensitive data to a specific group of users i.e., view helps to limit data access to specific users.
- View allows you to create the read only data to specific users. Hence providing extra security layer.
- A view enables backward compability

Disadvantages :

- **Performance:** Querying data from the view can be slow. If a view is created based on other view.
- You cannot create an index on a view.
- Table dependency.
- You cannot use sub-queries.

Specification of Views in SQL :

The command to specify a view is CREATE VIEW. 'We give the view a table name, a list of attribute names, and a query to specify the contents of view. If none of the view attributes result from applying functions or arithmetic operations, we do not have to specify attribute names for the view as they will be the same as the names of the attributes of the defining tables.

Example :

Consider the following sample tables

Student Details

S_ID	NAME	ADDRESS
1.	Harsh	Kolkata
2.	Ashish	Dungapur
3.	Pratik	Delhi
4.	Dhanraj	Bihar
5.	Ram	Rajasthan

Student Marks :

ID	NAME	MARKS	AGE
1.	Harsh	90	19
2.	Suresh	50	20
3.	Pratik	80	19
4.	Dhanraj	95	21
5.	Ram	85	18

Creating Views : We can create View using CREATE VIEW statement. A View can be created from a single table or multiple tables.

Syntax :

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE condition;
```

view_name: Name for the View

table_name: Name of the table

condition: Condition to select rows

EXAMPLE-1

Creating View from a single table:

In this example we will create a View named DetailsView from the table StudentDetails.

Query :

```
CREATE VIEW DetailsView AS
SELECT NAME, ADDRESS
FROM StudentDetails
WHERE S_ID < 5;
```

To see the data in the View, we can query the view in the same manner as we query a table.

```
SELECT * FROM DetailsView;
```


Output :

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

EXAMPLE-2

In this example, we will create a view named StudentNames from the table StudentDetails.

```
CREATE VIEW StudentNames AS
SELECT S_ID, NAME
FROM StudentDetails
ORDER BY NAME;
```

If we now query the view as,

```
SELECT * FROM StudentNames;
```

Output :

S_ID	NAMES
2	Ashish
4	Dhanraj
1	Harsh
3	Pratik
5	Ram

Deleting Views

We have learned about creating a View, but what if a created View is not needed anymore? Obviously we want to delete it. SQL allows us to delete an existing View. We can delete or drop a View using the DROP statement.

Syntax :

```
DROP VIEW view_name;
```

view_name: Name of the View which we want to delete.

For example, if we want to delete the View MarksView, we can do this as:

```
DROP VIEW MarksView;
```

Updating Views

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is not met, then we will not be allowed to update the view.

1. The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
2. The SELECT statement should not have the DISTINCT keyword.
3. The View should have all NOT NULL values.
4. The view should not be created using nested queries or complex queries.
5. The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

We can use the CREATE OR REPLACE VIEW statement to add or remove fields from a view.

Syntax :

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1, coulmn2,..
FROM table_name
WHERE condition;
```

For example, if we want to update the view MarksView and add the field AGE to this View from StudentMarks Table, we can do this as:

```
CREATE OR REPLACE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS,
StudentMarks.AGE
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```


If we fetch all the data from MarksView now as:

```
SELECT * FROM MarksView;
```

Output :

NAME	ADDRESS	MARKS	AGE
Harsh	Kolkata	90	19
Pratik	Delhi	80	19
Dhanraj	Bihar	95	21
Ram	Rajastl.an	85	18

Inserting a row in a view :

We can insert a row in a View in a same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a View.

Syntax :

```
INSERT INTO view_name (column1, column2, column3...) VALUES (value1, value2,
value3..);
```

EXAMPLE :

In the below example we will insert a new row in the View DetailsView which we have created above in the example of "creating views from a single table".

```
INSERT INTO DetailsView (NAME, ADDRESS) VALUES ("Suresh","Gurgaon");
```

If we fetch all the data from DetailsView now as,

```
SELECT * FROM DetailsView;
```

Output :

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar
Suresh	Gurgaon

Deleting a row from a View :

Deleting rows from a view is also as simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view. Also deleting a row from a view first delete the row from the actual table and the change is then reflected in the view.

Syntax :

```
DELETE FROM view_name
WHERE condition;
```

Example :

In this example we will delete the last row from the view DetailsView which we just added in the above example of inserting rows.

```
DELETE FROM DetailsView
WHERE NAME= "Suresh";
```

If we fetch all the data from DetailsView now as,

```
SELECT * FROM DetailsView;
```

Output :

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

Creating View from multiple tables :

In this example we will create a View named MarksView from two tables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the SELECT statement.

Query :

```
CREATE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

To display data of View MarksView:

```
SELECT * FROM MarksView;
```

Output:

NAME	ADDRESS	MARKS
Harsh	Kolkata	90
Pratik	Delhi	80
Dhanraj	Bihar	95
Ram	Rajasthan	85

Uses of a View :

A good database should contain views due to the given reasons :

Restricting data access Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table.

Hiding data complexity A view can hide the complexity that exists in a multiple table join.

Simplify commands for the user Views allows the user to select information from multiple tables without requiring the users to actually know how to perform a join.

Store complex queries Views can be used to store complex queries.

Rename Columns Views can also be used to rename the columns without affecting the base tables provided the number of columns in view must match the number of columns specified in select statement. Thus, renaming helps to hide the names of the columns of the base tables.

Multiple view facility Different views can be created on the same table for different users.

5.8 FAMILIARIZE WITH PL/SQL

PL/SQL stands for **Procedural Language/SQL**. PL/SQL extends SQL by adding control structures along with the procedural features of other programming languages. PL/SQL combines the flexibility of SQL with powerful feature of 3rd generation Language.

WARNING XEROX/PHOTOCOPYING OF THIS BOOK IS ILLEGAL

It was developed by Oracle Corporation in early 1980's to enhance the capabilities of SQL. The procedural construct and database access are present in PL/SQL.PL/SQL can be used in both in database in Oracle Server and in Client side application development tools.

PL/SQL extends SQL by adding constructs found in procedural languages, resulting in a structural language that is more powerful than SQL. PL/SQL is not case sensitive. C style comments (/*..... */) may be used in PL/SQL programs whenever required.

PL/SQL engine: Oracle uses a PL/SQL engine to process the PL/SQL Statements. A PL/SQL code can be stored in client system or at server side.

PL/SQL Features:

(i) Procedural language capability.

(ii) Better performance.

(iii) Exception handling.

Advantages of PL/SQL

Support for SQL, support for object-oriented programming, better performance, portability, higher productivity, Integration with Oracle

- Supports the declaration and manipulation of object types and collections.
- Allows the calling of external functions and procedures.
- Contains new libraries of built in packages.
- With PL/SQL, multiple sql statements can be processed in a single command line statement.

PL/SQL Block :

Each PL/SQL program consists of SQL and PL/SQL statement which forms a PL/SQL block. All PL/SQL programs are made up of blocks; each block performs a logical action in the program. A PL/SQL block consists of three parts

1. Declaration section
2. Executable section
3. Exception handling section

Only the executable section is required. The other sections are optional.

A PL/SQL block has the following structure:

```
DECLARE
/* Variable section/Declaration section */
BEGIN
/* Executable section */
EXCEPTION
/* Exception handling section */
END;
```

1. **Declaration section :**This is first section which is start with a reserved keyword DECLARE. It is optional and is used to declare any place holders like variables, constant record, cursor etc. which are used to manipulate data in execution statements and stores the data temporarily. All the identifiers (constants and variables) are declared in this section before they are used in SELECT command.
2. **Executable section :** This section starts with a reserved keyword BEGIN and ends with END. This section contain procedural and SQL statements. This is the only section of the block which is required. This is a mandatory section where the program logic is written to perform any task. The programmatic constructs like loops, conditional statements and SQL statements form the part of execution section.
   - The only SQL statements allowed in a PL/SQL program are SELECT, INSERT, UPDATE, DELETE and several other data manipulation statements.
   - Data definition statements like CREATE, DROP or ALTER are not allowed.
   - The executable section also contains constructs such as assignment statements, branches, loops, procedure calls and triggers.
3. **Exception handling section :** This section is optional, used to handle errors that occur during execution of PL/SQL program. This section starts with a reserved keyword EXCEPTION. If the exceptions are not handled the block terminates abruptly with errors.

The END indicate end of PL/SQL block.

PL/SQL programs, can be invoked either by typing it in sqlplus or by putting the code in a file and invoking the file. To execute it use / on SQL prompt or use . and run.

Note : Every statement in the above 3 sections must be ended with semicolon (;)

5.8.1 ARCHITECTURE OF PL/SQL

The PL/SQL compilation and run-time system is a technology, not an independent product. Think of this technology as an engine that compiles and executes PL/SQL blocks and subprograms. The engine can be installed in an Oracle server or in an application development tool such as Oracle Forms or Oracle Reports. So, PL/SQL can reside in two environments:

1. The Oracle server
2. Oracle tools.

These two environments are independent. PL/SQL is bundled with the Oracle server but might be unavailable in some tools. In either environment, the PL/SQL engine accepts as input any valid PL/SQL block or subprogram.

Below Fig. 5.1 shows the PL/SQL engine processing an anonymous block. The engine executes procedural statements but sends SQL statements to the SQL Statement Executor in the Oracle server.

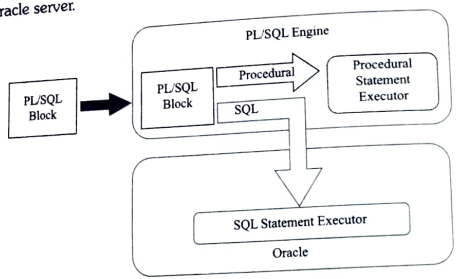

FIG 5.1 : PL/SQL Engine

5.9 VARIOUS DATA TYPES IN PL/SQL

Predefined PL/SQL datatypes are grouped into composite, LOB, reference, and scalar type categories.

- A composite type has internal components that can be manipulated individually, such as the elements of an array, record, or table.

- A LOB type holds values, called lob locators, that specify the location of LARGE OBJECTS, such as text blocks or graphic images, that are stored separately from other database data. LOB types include BFILE, BLOB, CLOB, and NCLOB.
- A reference type holds values, called pointers that designate other program items. These types include REF CURSORS and REFs to object types.
- A scalar type has no internal components. It holds a single value, such as a number or character string. The scalar types fall into four families, which store number, character, Boolean, and date/time data.

PL/SQL Datatypes : It is of four types,

(i) Scalar (ii) Large Object (LOB)
(iii) Composite (iv) Reference.

Scalar Datatypes : It is divided into four types,

(i) Numeric (ii) Character
(iii) Boolean (iv) Date time.

Numeric Datatypes : PL/SQL Predefined numeric datatype are,

(i) int (ii) integer (iii) small int
(iv) float (v) real (vi) double
(vii) number (viii) decimal.

PL/SQL character datatype :

(i) char (ii) varchar2 (iii) raw
(iv) nchar (v) nvarchar2 (vi) long
(vii) long raw (viii) rowid.

PL/SQL Boolen datatype :

(i) true (ii) false

PL/SQL date time :

(i) year (ii) month (iii) day
(iv) hour (v) minute (vi) second

PL/SQL Large object datatype : Large object datatype refers to data items such as graphical images, video clips and sound waves.

- Predefined PL/SQL LOB datatypes are,
  - (i) BFILE — B - Binary
  - (ii) BLOB — C - Character
  - (iii) CLOB
  - (iv) NCLOB

Composite datatype : It is a combination of various scalar datatypes,

Ex : Record

PL/SQL Variable :

Syntax :

- Variablename datatypes; [declaring a variable]

  Ex : Pin varchar2 (30);
- Variablename = value; [Initialization of variable]

(or)

Variablename datatype NOT NULL = value;

Ex : name varchar2 (30) NOT NULL = 'raju';

(or)

Ex : name varchar2 (30);

name = 'raju';

PL/SQL constant :

Syntax : Constant_name constant datatype = value;

Ex :

PI constant number (3, 2) = 3.14;

Write a PL/SQL program to add 2 numbers?

```
>SET SERVEROUTPUT ON;
>DECLARE
a int;          /* a number(3); or a integer:=10; */
b int;          /* b number(3); or b integer:=20; */
c int;          /* c number(3); or c integer; */
BEGIN
a:=10;
b:=20;
c:=a+b;
dbms_output.put_line(c);
END;
```

Output : cvalue = 30

Scalar Types :

BINARY_DOUBLE BINARY_FLOAT BINARY_INTEGER DEC DECIMAL DOUBLE PRECISION FLOAT INT INTEGER NATURAL NATURALIN NUMBER NUMERIC PLS_INTEGER POSITIVE POSITIVEN REAL SIGNTYPE SMALLINT	CHAR CHARACTER LONG LONGRAW NCHAR NVARCHAR2 RAW ROWID STRING UROWID VARCHAR VARCHAR2 BOOLEAN DATE	**COMPOSIT TYPES** RECORD TABLE VARRAY **COMPOSIT TYPES** REF CURSOR REF object_type **LOB Types** BFILE BLOB CLOB NCLOB

5.10 CONTROL STATEMENTS IN PL/SQL WITH EXAMPLES

Procedural computer programs use the basic control statements.

Control statements in PL/SQL are divided into two types.

(i) Conditional/Selection statements

(ii) Iteration/Looping statements.

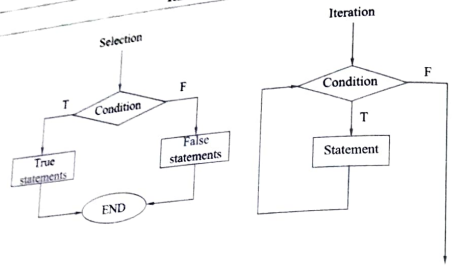

FIG 5.2 : PL/SQL Selection and Iteration Statements

The selection structure tests a condition, and then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition is any variable or expression that returns a BOOLEAN value (TRUE or FALSE).

The iteration structure executes a sequence of statements repeatedly as long as a condition holds true.

IF and CASE Statements: The IF statement executes a sequence of statements depending on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF.

The CASE statement is a compact way to evaluate a single condition and choose between many alternative actions. It makes sense to use CASE when there are three or more alternatives to choose from.

Using the IF-THEN Statement : The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF)

The sequence of statements is executed only if the condition is TRUE. If the condition is FALSE or NULL, the IF statement does nothing. In either case, control passes to the next statement.

Using CASE Statements: Like the IF statement, the CASE statement selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.

LOOP and EXIT Statements: LOOP statements execute a sequence of statements multiple times. There are three forms of LOOP statements: LOOP, WHILE-LOOP, and FOR-LOOP.

Using the LOOP Statement: The simplest form of LOOP statement is the basic loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP
sequence-of-statements
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. You use an **EXIT** statement to stop looping and prevent an infinite loop. You can place one or more EXIT statements anywhere inside a loop, but not outside a loop.

There are two forms of EXIT statements:

1. EXIT and
2. EXIT-WHEN.

Using the EXIT Statement: The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement.

Using the EXIT-WHEN Statement: The EXIT-WHEN statement lets a loop complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop.

Using the WHILE-LOOP Statement: The WHILE-LOOP statement executes the statements in the loop body as long as a condition is true:

```
WHILE condition LOOP
sequence-of-statements
END LOOP;
```

Using the FOR-LOOP Statement: Simple FOR loops iterate over a specified range of integers. The number of iterations is known before the loop is entered. A double dot (..) serves as the range operator. The range is evaluated when the FOR loop is first entered and is never re-evaluated. If the lower bound equals the higher bound, the loop body is executed once.

Conditional Statement in PL/SQL: PL/SQL support programming language features like conditional statments/iterative statment.

Conditional Statement : Oracle Database

(i) IF THEN
(ii) IF THEN ELSE
(iii) IF THEN ELSIF
(iv) EXIT
(v) CONTINUE
(vi) GOTO
(vii) CASE
(viii) EXIT WHEN

Syntax of IF statement:

```
IF <condition> THEN
<statement_list>
END IF;
```

If condition is true the statements present inside IF will get executed.

Ex :

```
IF (a<=20) THEN
C := C+1;
END IF;
```

Syntax of IF THEN ELSE :

```
IF <condition> THEN
<statement_list>
ELSE
<statement_list>
END IF;
```

Syntax of IF THEN ELSIF / Multiway Branch

```
IF <condition_1> THEN
<statement_list>
ELSIF <condition_2> THEN
<statement_list>
ELSIF <condition_n> THEN
<statement_list>
ELSE
```



```
<statement_list>
END IF;
```

Syntax for CASE:

```
CASE selector
WHEN value1 THEN
Statement-1;
WHEN value2 THEN
Statement-2;
WHEN value3 THEN
Statement-3;
—-
ELSE statment-n; — default case
END CASE;
```

Syntax for exit:

```
EXIT;
```

Syntax for goto:

```
GOTO label;
——-
——-
<<label>>
Statement;
```

EXAMPLE-1

Accept two numbers and print the largest number

```
DECLARE
x number;
y number;
BEGIN
x :=&x;
y :=&y;
if (x>y) then
dbms_output.put_line(x is largest than y);
```

```
else
dbms_output.put_line(y is largest than x);
end if;
END;
/
```

OUTPUT:

```
SQL> /
Enter value for x : 7
Enter value for y : 8
y is largest than x
```

EXAMPLE-2

Check whether the salary of SURESH is greater than 5000 or not.

```
DECLARE
B_salary emp.sal%type;
BEGIN
SELECT sal INTO B_salary
FROM emp
WHERE ename=SURESH;
IF (B_salary > 5000) THEN
dbms_output.put_line(SURESH salary is largest than 5000);
ELSE
dbms_output.put_line(SURESH salary is less than 5000);
END IF;
END;
/
```

Output

```
SQL> /
SURESH salary is less than 5000
```

EXAMPLE-3

Write a PL/SQL program using goto statement

```
>SET SERVEROUTPUT ON;
```



```
>DECLARE
a number(2):=30;
BEGIN
<<loopstart>>
WHILE a<50 LOOP
dbms_output.put_line('value of a is:'||a);
a := a+1;
IF a=35 THEN
a := a+14;
GOTO loopstart;
END IF;
END LOOP;
END;
```

Output :

```
value of a : 30
value of a : 31
value of a : 32
value of a : 33
value of a : 34
value of a : 49
```

Iterative Statements :

There are three types of loop/iterative statements in PL/SQL:

1. Simple loop
2. While loop.
3. For...loop

1. Simple Loop

Syntax 1:

```
LOOP
<commands> /* A list of statements. */
if <codition> then
EXIT;
```

```
End if;
END LOOP;
```

The loop breaks if <condition> is true.

For example :

```
DECLARE
i NUMBER := 0;
BEGIN
LOOP
i := i+1;
dbms_output.put_line(i);
If(i>=10) then
EXIT;
End if;
END LOOP;
END;
/
```

Output

```
SQL> /
1
2
3
4
5
6
7
8
9
10
```


Syntax 2:

```
LOOP
<commands> /* A list of statements. */
EXIT WHEN <condition>;
END LOOP;
```

For example :

```
DECLARE
i NUMBER := 0;
BEGIN
LOOP
i := i+1;
dbms_output.put_line(i);
EXIT when i>=10;
END LOOP;
END;
/
```

Output

```
SQL> /
1
2
3
4
5
6
7
8
9
10
```

2. While loop

Syntax:

```
WHILE <condition> LOOP
<commands> /* A list of statements. */
END LOOP;
```

For example:

```
DECLARE
i NUMBER := 0;
BEGIN
While i<=10 LOOP
i := i+1;
dbms_output.put_line(i);
END LOOP;
END;
/
```

Output

```
SQL> /
1
2
3
4
5
6
7
8
9
10
```


3. FOR...LOOP

Syntax :

```
FOR <var> IN[reverse] <start_value>. .<finish_value> LOOP
<commands> /* A list of statements. */
END LOOP;
```

Here, <var> can be any variable; it is local to the for-loop and need not be declared. Also, <start_value> and <finish_value> are constants. The value of a variable <var> is automatically incremented by 1.

The commands inside the loops are automatically executed until the final value of variable is reached. Reverse is optional part, when you want to go from maximum value to minimum value in that case reverse is used.

EXAMPLE-1

```
> DECLARE
a number(2);
BEGIN
FOR a IN 10..20 LOOP (or) FOR a IN reverse 10..20 LOOP
dbms_output.put_line(a);
END LOOP;
END;
```

EXAMPLE-2

```
BEGIN
FOR i IN 1. .10 LOOP
dbms_output.put_line(i);
END LOOP;
END;
```

Output

```
SQL> /
1
2
3
4
5
```

```
6
7
8
9
10
```

MYSQL : Syntax of conditional statements, according to MYSQL

If then else :

```
If condition1 THEN
Statement1;
[ElSEIF condition2 THEN
Statement;]
[ELSE]
Statement;
END IF;
```

While:

```
WHILE condition
DO
Statements;
END WHILE;
```

CASE Selector :

```
CASE case_expression
WHEN value1 THEN
Statement1;
WHEN value2 THEN
Statement2;
ELSE statementn;
END CASE;
```

Syntax for iterative statement :

```
iterate labelname;
```


Syntax for goto:

```
labelname;
Statements;
Iterate labelname;
Statements;
leave labelname;
```

PL/SQL EXAMPLE PROGRAMS

EXAMPLE-1

Print the largest number among three numbers.

```
>DECLARE
a integer:= 10;
b integer:= 20;
c integer:= 30;
BEGIN
IF ((a>b) and (a>c)) THEN
dbms_output.put_line('a is greater '||a);
ELSIF ((b>a) and (b>c)) THEN
dbms_output.put_line('b is greater '||b);
ELSIF ((c>a) and (c>b)) then
dbms_output.put_line('c is greater '||c);
END IF;
END IF;
END IF;
END;
```

Output :

```
30 is grater
```

EXAMPLE-2

Accept a number and check whether it is odd or even. If it is even no. p
square of it otherwise cube of it.

```
DECLARE
n1 number;
BEGIN
n1:=&n1;
IF (mod(n1, 2)=0) THEN
dbms_output.put_line(n1 || ' is even no');
dbms_output.put_line('Square of ' || n1 ||' is '|| n1*n1);
ELSE
dbms_output.put_line(n1 at ||' is odd no');
dbms_output.put_line('Cube of ' || n1 || ' is ' || n1 * n1 * n1);
END IF;
END;
```

Output:

```
Enter value for n1:4
4 is even no
square of 4 is 16
```

EXAMPLE-3

Write a PL/SQL program to check vowel or consonant using Oracle.

```
DECLARE
choice char(1):= 'i';
BEGIN
CASE choice
WHEN 'a' THEN
dbms_output.put_line('vowel');
WHEN 'e' THEN
dbms_output.put_line('vowel');
WHEN 'i' THEN
dbms_output.put_line('vowel');
WHEN 'o' THEN
dbms_output.put_line('vowel');
WHEN 'U' THEN
dbms_output.put_line('vowel');
ELSE dbms_output.put_line (choice || 'is consonant');
END CASE;
END;
```

Output: *Vowel*

EXAMPLE-4

Write a PL/SQL program to find reverse of a number using while loop:

```
DECLARE
num integer :=123;
i integer;
rev integer := 0;
BEGIN
WHILE (num > 0)
LOOP
i := mod(num,10);
rev := (rev * 10) + i;
num := num/10;
END LOOP;
dbms_output.put_line ('reverse of no =' || rev);
END;
```

Output:

```
reverse of no = 321
```

EXAMPLE-5

Write a PL/SQL program using label

```
> SET SERVEROUTPUT ON
> DECLARE
i number(1);
j number(1);
BEGIN
<< outer_loop>>
FOR i IN 1..3 LOOP
<< inner_loop>>
For j in 1..3 LOOP
dbms_output.put_line ('i is ='|| i);
dbms_output.put_line ('j is ='|| j);
END LOOP inner_loop;
END LOOP outer_loop;
```

```
END;
/
```

(Nesting of loop) output :

```
i is = 1
j is = 1
i is = 1
j is = 2
i is = 1
j is = 3
i is = 2
j is = 1
i is = 2
j is = 2
i is = 2
j is = 3
i is = 3
j is = 1
i is = 3
j is = 2
j is = 3
j is = 3
```

EXAMPLE-6

Write a PL/SQL program on exit when statement

```
> SET SERVEROUTPUT ON
> DECLARE
a number(2) : = 10;
BEGIN
WHILE a < 20 LOOP
dbms_output.put_line (a);
a : = a + 1;
EXIT WHEN a > 15;
```



```
END LOOP;
END;
```

Output :

```
10
11
12
13
14
15
```

EXAMPLE-7

Write a PL/SQL program using continue statement

```
> SET SERVEROUTPUT ON
> DECLARE
a number(2) : = 10;
BEGIN
WHILE a < 20 LOOP
dbms_output.put_line(a);
a: = a + 1;
IF a = 15 THEN
a: = a + 1;
CONTINUE;
END IF;
END LOOP;
END;
/
```

Output :

```
10
11
12
```

13
14
16
17
18
19

PL/SQL EXAMPLE PROGRAMS USING TABLES

EXAMPLE-1

Accept the deptno and print the no. of employees working in that department.

```
DECLARE
v_deptno emp.deptno%type;
v_count number;
BEGIN
v_deptno:=&v_deptno;
SELECT count(*) INTO v_count
FROM emp
WHERE deptno=v_deptno;
dbms_output.put_line('No. of emp working in '|| v_deptno || 'are' || v_count);
END;
/
```

Output :

```
SQL> /
Enter value for v_deptno: 20
No. of emp working in 20 are 2
```

EXAMPLE-2

Accept the deptno and print the department name and location.


```
DECLARE
v_deptno dept.deptno%type;
v_dname dept.dname%type;
v_loc dept.loc%type;
BEGIN
v_deptno:=&v_deptno;
SELECT dname, loc INTO v_dname, v_loc
FROM dept
WHERE deptno=v_deptno;
dbms_output.put_line('Department name is '|| v_dname ||' and location is '||
v_loc);
END;
/
```

Output :

```
SQL> /
Enter value for v_deptno: 10
Department name is ACCOUNTING and location is NEW YORK
```

EXAMPLE-3

Write a PL/SQL program on IF-THEN along with records.

```
SELECT salary INTO Csal FROM employee WHERE id = cid;
IF (Csal < = 20000) THEN
UPDATE employee SET
Salary = Salary + 20000 WHERE id = Cid;
dbms_output.put_line('salary is updated');
END IF;
END;
```

Output:

```
Salary is updated
```

5.11 SEQUENTIAL CONTROL GOTO AND NULL STATEMENTS

GOTO and NULL Statements: The GOTO statement is rarely used. Occasionally, it can simplify logic enough to warrant its use. The NULL statement can improve readability by making the meaning and action of conditional statements clear.

A **GOTO** statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labeled statement in the same subprogram.

NOTE " the use of GOTO statement is not recommended in any programming language because it makes it difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a GOTO can be rewritten so that it doesn't need the GOTO.

Overuse of GOTO statements can result in code that is hard to understand and maintain. Use GOTO statements rarely. For example, to branch from a deeply nested structure to an error-handling routine, raise an exception rather than use a GOTO statement.

Using the GOTO Statement: The GOTO statement branches to a label unconditionally. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. When executed, the GOTO statement transfers control to the labeled statement or block. The labeled statement or block can be down or up in the sequence of statements.

Syntax :

The syntax for a GOTO statement in PL/SQL is as follows "

```
GOTO label;
..
..
<< label >>
statement;
```

Here the label declaration which contains the label_name encapsulated within the << >> symbol and must be followed by at least one statement to execute.

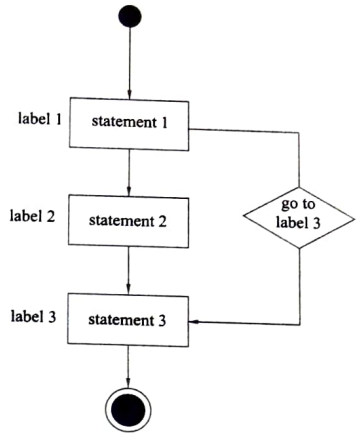

FIG 5.3 : Flow Diagram of GOTO Statement

EXAMPLE-1

```
> SET SERVEROUTPUT ON
DECLARE
    a number(2) := 10;
BEGIN
    <<loopstart>>
    WHILE a < 20 LOOP
    dbms_output.put_line ('value of a: ' || a);
        a := a + 1;
        IF a = 15 THEN
        a := a + 1;
        GOTO loopstart;
    END IF;
END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Restriction on GOTO Statement : Following is a list of some restrictions imposed on GOTO statement.

- Cannot transfer control into an IF statement, CASE statement, LOOP statement or sub-block.
- Cannot transfer control from one IF statement clause to another or from one CASE statement WHEN clause to another.
- Cannot transfer control from an outer block into a sub-block.
- Cannot transfer control out of a subprogram.
- Cannot transfer control into an exception handler.

Using the NULL Statement: Usually when you write a statement in a program, you want it to do something. There are cases, however, when you want to tell PL/SQL to do absolutely nothing, and that is where the NULL statement comes in handy. The NULL statement does nothing, and passes control to the next statement. Some languages refer to such an instruction as a no-op (nooperation).

The NULL statement has the following format:

NULL : The NULL statement is simply the reserved word NULL followed by a semicolon (;) to indicate that this is a statement and not a NULL value. The NULL statement does nothing except pass control to the next executable statement.

Improving Program Readability: Sometimes, its helpful to avoid any ambiguity inherent in an IF statement that doesnt cover all possible cases. For example, when you write an IF statement, you do not have to include an ELSE clause. To produce a report based on a selection, you can code:

```
IF: report_mgr.selection = 'DETAIL'
THEN
  exec_detail_report;
END IF;
```

What should the program be doing if the report selection is not DETAIL'? One might assume that the program is supposed to do nothing. You include an explicit ELSE clause that does nothing, you state very clearly, "Dont worry, I thought about this possibility and I really want nothing to happen:"

```
IF : report_mgr.selection = 'DETAIL'
THEN
  exec_detail_report;
ELSE
  NULL;
END IF;
```

Our example here was of an IF statement, but the same principle applies when writing CASE statements and CASE expressions. Similarly, if you want to temporarily remove all the code from a function or procedure, and yet still invoke that function or procedure, you can use NULL as a placeholder. Otherwise, you cannot compile a function or procedure without having any lines of code within it.

EXAMPLE-2

Using the NULL Statement to Show No Action.

```
DECLARE
vjobID VARCHAR2(10);
vempID NUMBER(6):= 110;
BEGIN
SELECT job-id INTO vjobID FROM employees WHERE employee-id = vempID;
```

```
IF vjobID = 'SALES-REP' THEN
UPDATE employees SET commission = commission*1.2;
ELSE
NULL;
END IF;
END;
/
```


REVIEW QUESTIONS

One Mark Questions :

1. Define sub query.
2. What is join statement?
3. List any two schema objects.
4. Define index.
5. Define sequence.
6. Define synonym.
7. Define view.
8. What is PL/SQL?
9. List any three data types of PL/SQL.
10. List the conditional statements in PL/SQL.
11. List the iterative statements in PL/SQL.
12. Define NULL statement.

Three Mark Questions :

1. List different types of JOIN statements.
2. List different types of schema objects.
3. Write about indexes.
4. Write the syntax to create a sequence.
5. Write the syntax to create synonym.
6. Write the syntax to create a view.
7. Write any three advantages of views.
8. Write the features of PL/SQL.
9. Write the benefits of PL/SQL.
10. Write the syntax of any two conditional statements in PL/SQL.
11. Write the syntax of any two looping statements in PL/SQL.
12. Write about GOTO statement.

Five Mark Questions :

1. Implement sub queries with two examples.
2. Explain the types of SQL JOIN statements with examples.
3. Describe the steps of managing indexes.
4. Explain the management of sequences with an example.
5. Explain about synonyms in detail with an example.
6. Write about views in detail with an example?
7. Explain DDL statements in SQL with one example.
8. Develop a PL/SQL program to reverse a given number.
9. Develop a PL/SQL program to find largest number among given three numbers.
10. Develop a PL/SQL program to check whether the given character is vowel or not using ASE statement.
11. Write a PL/SQL program to display prime numbers between 1 to 100 using FOR loop.
12. Develop a PL/SQL program using GOTO statement.

CHAPTER

Six

ADVANCED PL/SQL

CHAPTER OUTLINE

6.1 PL/SQL RECORDS

A **record** is a data structure that can hold data items of different kinds. Records consist of different fields, similar to a row of a database table.

For example, you want to keep track of your books in a library. You might want to track the following attributes about each book, such as Title, Author, Subject, BookID. A record containing a field for each of these items allows treating a BOOK as a logical unit and allows you to organize and represent its information in a better way.

PL/SQL can handle the following types of records "

- Table-based
- Cursor-based records
- User-defined records

Table-Based Records : The %ROWTYPE attribute enables a programmer to create table-based and cursorbased records.

The following example illustrates the concept of table-based records. We will be using the CUSTOMERS table.

```
DECLARE
  customer_rec customers%rowtype;
BEGIN
  SELECT * into customer_rec
  FROM customers
  WHERE id = 5;
  dbms_output.put_line('Customer ID: ' || customer_rec.id);
  dbms_output.put_line('Customer Name: ' || customer_rec.name);
  dbms_output.put_line('Customer Address: ' || customer_rec.address);
  dbms_output.put_line('Customer Salary: ' || customer_rec.salary);
END;
/
```


Output :

```
Customer ID: 5
Customer Name: Hardik
Customer Address: Bhopal
Customer Salary: 9000
```

Cursor-Based Records : The following example illustrates the concept of cursor-based records. We will be using the CUSTOMERS table.

```
DECLARE
  CURSOR customer_cur is
    SELECT id, name, address
    FROM customers;
  customer_rec customer_cur%rowtype;
BEGIN
  OPEN customer_cur;
  LOOP
    FETCH customer_cur into customer_rec;
    EXIT WHEN customer_cur%notfound;
    dbms_output.put_line(customer_rec.id || ' ' || customer_rec.name);
  END LOOP;
END;
/
```

Output:

```
1 Ramesh
2 Khilan
3 kaushik
4 Chaitali
5 Hardik
6 Komal
```

User-Defined Records : PL/SQL provides a user-defined record type that allows you to define the different record structures. These records consist of different fields. Suppose

you want to keep track of your books in a library. You might want to track the following attributes about each book

- Title
- Author
- Subject
- Book ID
- Defining a Record

The record type is defined as :

```
TYPE
type_name IS RECORD
 ( field_name1  datatype1  [NOT NULL]  [:= DEFAULT EXPRESSION],
  field_name2   datatype2   [NOT NULL]  [:= DEFAULT EXPRESSION],
  ...
  field_nameN  datatypeN  [NOT NULL]  [:= DEFAULT EXPRESSION);
record-name  type_name;
```

The Book record is declared in the following way -

```
DECLARE
TYPE books IS RECORD
     (title varchar(50),
     author varchar(50),
     subject varchar(100),
     book_id number);
book1 books;
book2 books;
```

Accessing Fields : To access any field of a record, we use the dot (.) operator. The member access operator is coded as a period between the record variable name and the field that we wish to access.

Following is an example to explain the usage of record

```
DECLARE
type books is record
            (title varchar(50),
            author varchar(50),
            subject varchar(100),
            book_id number);
 book1 books;
 book2 books;
BEGIN
 —— Book 1 specification
 book1.title  := 'C Programming';
 book1.author := 'Dennis Ritchie';
 book1.subject := 'C Programming Tutorial';
 book1.book_id := 6495407;
 - — Book 2 specification
 book2.title := 'C++ Programming';
 book2.author := 'Bjarne Stroustrup';
 book2.subject := ''C++ Programming Tutorial';
 book2.book_id := 6495700;
 - — Print book 1 record
 dbms_output.put_line('Book 1 title: '|| book1.title);
 dbms_output.put_line('Book 1 author : '|| book1.author);
 dbms_output.put_line('Book 1 subject: '|| book1.subject);
 dbms_output.put_line('Book 1 book_id: ' || book1.book_id);
 - — Print book 2 record
 dbms_output.put_line('Book 2 title: '|| book2.title);
 dbms_output.put_line('Book 2 author: '|| book2.author);
 dbms_output.put_line('Book 2 subject: '|| book2.subject);
 dbms_output.put_line('Book 2 book_id: '|| book2.book_id);
END;
/
```

Output :

```
Book 1 title: C Programming
Book 1 author: Dennis Ritchie
Book 1 subject: C Programming Tutorial
Book 1 book_id: 6495407
Book 2 title: C++ Programming
Book 2 author: Bjarne Stroustrup
Book 2 subject: C++ Programming Tutorial
Book 2 book_id: 6495700
```

6.2 SUBPROGRAMS

Subprograms are named PL/SQL blocks that can be called with a set of parameters. PL/SQL has two types of subprograms, **procedures**, and **functions**. Procedures and Functions are the subprograms which can be created and saved in the database as database objects. They can be called or referred inside the other blocks also.

Subprograms have :

- A declarative part, with declarations of types, cursors, constants, variables, exceptions, and nested subprograms.
- An executable part, with statements that assign values, control execution and manipulate Oracle data.
- An optional exception-handling part, which deals with runtime error conditions.

Advantages of PL/SQL Subprograms

- Subprograms let you extend the PL/SQL language. Procedures act like new statements. Functions act like new expressions and operators.
- Subprograms break a program down into manageable, well-defined modules.
- Subprograms promote reusability.
- Subprograms promote maintainability.

- Dummy subprograms (stubs) let you defer the definition of procedures and functions until after testing the main program.

Before we learn about PL/SQL subprograms, we will discuss the various terminologies that are the part of these subprograms.

Parameter : The parameter is variable or placeholder of any valid PL/SQL datatype through which the PL/SQL subprogram exchange the values with the main code.

For each parameter, you specify :

- Its name.
- Its parameter mode (IN, OUT, or IN OUT). If you omit the mode, the default is IN. The optional NOCOPY keyword speeds up the processing of large OUT or IN OUT parameters.
- Its datatype. You specify only the type, not any length or precision constraints.
- Optionally, its default value.

Based on their purpose parameters are classified as

1. IN Parameter
2. OUT Parameter
3. IN OUT Parameter

These parameter types should be mentioned at the time of creating the subprograms.

RETURN : RETURN is the keyword that instructs the compiler to switch the control from the subprogram to the calling statement. In subprogram RETURN simply means that the control needs to exit from the subprogram. Once the controller finds RETURN keyword in the subprogram, the code after this will be skipped.

6.3 PROCEDURES

PL/SQL Procedures : A **procedure** is a subprogram that performs a specific action. You specify the name of the procedure, its parameters, its local variables, and the BEGIN-END block that contains its code and handles any exceptions.

The parameter is variable or placeholder of any valid PL/SQL datatype through which the PL/SQL subprogram exchange the values with the main code. This parameter allows to give input to the subprograms and to extract from these subprograms.

- These parameters should be defined along with the subprograms at the time of creation.
- These parameters are included in the calling statement of these subprograms to interact the values with the subprograms.
- The datatype of the parameter in the subprogram and the calling statement should be same.
- The size of the datatype should not mention at the time of parameter declaration, as the size is dynamic for this type.

Based on their purpose parameters are classified as,

1. IN parameter.
2. OUT parameter.
3. IN OUT parameter.

1. IN Parameter :

- This parameter is used for giving input to the subprograms.
- It is a read-only variable inside the subprograms. Their values cannot be changed inside the subprograms.
- In the calling statement, these parameters can be a variable or or literal value or an expression, for example, it could be the arithmetic expression like '5×8' or 'a/b' where 'a' and 'b' are variables.
- By default, the parameters are of IN type.

2. OUT Parameter :

- This parameter is used for getting output from the subprograms.
- It is a read-write variable inside the subprograms. Their values can be changed inside the subprograms.
- In the calling statement, these parameters should always be a variable to hold the value from the current subprograms.

3. IN OUT Parameter :

- This parameter is used for both giving input and for getting output from the subprograms.
- It is a read-write variable inside the subprograms. Their values can be changed inside the subprograms.

- In the calling statement, these parameters should always be a variable to hold the value from the subprograms.

These parameter types should be mentioned at the time of creating the subprograms.

IN Parameter : It is used to send values to procedure and it is a read only parameter.

```
CREATE [OR REPLACE] PROCEDURE
    procedure_name(
    Parametername1 IN datatype,
    Parametername2 IN datatype,
    .....)
    [IS/AS]
    declaration section;
    BEGIN
    execution section;
    EXCEPTION
    exception handing section;
    END;
```

(or)

```
CREATE [OR REPLACE] PROCEDURE
    procedurename(
    IN parametername1 datatype,
    IN parametername2 datatype,
    .....)
    [IS/AS]
    declaration section;
    BEGIN
    execution section;
    EXCEPTION
    exception handling section;
    END;
```

Example :
```
CREATE PROCEDURE addition(
    a IN integer,
    b IN integer)
> CREATE PROCEDURE addition(
    IN a integer,
    IN b integer)
```

Out/Parameter : These are used to get values from procedure, this is a write only parameter and are used to send the output from a procedure or function i.e., we cannot pass values to outparameter while executing.

SQL Syntax :

```
CREATE PROCEDURE procedurename(
    parametername1 OUT datatype
    parametername2 OUT datatype
            ⋮
)
AS
declaration section;
begin
execution section;
exception handling section;
end;
```

Syntax : MYSQL

```
> CREATE PROCEDURE procedurename(
OUT parametername1 datatype,
 OUT parametername2 datatype,
 .....)
BEGIN
 excution section;
EXCEPTION
exception handling section;
END;
```



```
> CREATE PROCEDURE addition(
  a OUT integer,
  b OUT integer)
 > CREATE PROCEDURE addition(
  OUT a integer,
  OUT b integer)
 > CREATE PROCEDURE procedurename(
  parametername1 IN datatype,
  parametername2 OUT datatype,
            ⋮
  )
  BEGIN
 Execution section;
 Exception handling section;
 end;
```

Ex :

```
> CREATE PROCEDURE add(
 IN a int,
 OUT b int)
 /
```

IN OUT Parameter :

```
> CREATE PROCEDURE procedurename(
  parametrename1 IN OUT datatype,
  parametername2 IN OUT datatype)
 .....)
```

Ex :

```
> CREATE OR REPLACE PROCEDURE name( )
 AS
  BEGIN
  dbms_output.put_line('Bahubali');
  END;
   /
```

Execute procedure :

```
EXECUTE  name( );
          (or)
EXEC  name ( );
```

Deleting procedure :

```
> DROP  PROCEDURE  procedurename;
```

EXAMPLE-1

Write a procedure to find minimum number

```
CREATE 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;

DECLARE
a number;
b number;
c number;
BEGIN
a : = 23;
b : = 45;
findMin(a, b, c);
dbms_output.put_line('Minimum of (23, 45) :' ||c);
END;
/
```

Output :

```
Minimum of (23, 45) : 23
```


EXAMPLE-2

Write a PL/SQL procedure to find square of number using IN & OUT mode

```
CREATE PROCEDURE squareNum(x IN OUT number) IS
BEGIN
x := x*x;
END;

DECLARE
a number(5);
BEGIN
a := 16;
squareNum(a);
dbms_output.put_line('Square of (16) :' ||a);
END;
/
```

Output :

```
Square of (16) : 256
```

EXAMPLE-3

Write a PL/SQL procedure in MYSQL to add two numbers

```
> DELIMETER @
> Create PROCEDURE addition ( )
BEGIN
DECLARE  a  int;
DECLARE  b  int;
DECLARE  c  int;
SET  a = 10;
SET  b = 20;
SET  c = a + b;
SELECT C;
END;
@
```

Output :

```
> call addition ( )
@ C/30
```

EXAMPLE-4

Write a PL/SQL program in MYSQL for parameter modes (IN)

```
> DELIMITER //
> CREATE PROCEDURE getpin (IN rollno varchar(30))
BEGIN
SELECT * FROM student WHERE
pin = rollno;
END;
//
> call getpin ('17001_CM_230');
```

Pin	Address
17001 - CM - 230	Mumbai

EXAMPLE-5

Write a PL/SQL program in MYSQL for parameter modes (out)

```
> DELIMITER //
> CREATE PROCEDURE outdemo (IN sbranch varchar(30), OUT total int)
BEGIN
SELECT count (*) INTO total FROM
Student where branch = sbranch;
END;
//
> call outdemo ('CME', @ Total);
> Select @total;
```

Output :

```
@ total/6
```


6.4 FUNCTIONS

A **function** is a subprogram that computes and returns a single value. Functions and procedures are structured alike, except that functions have a RETURN clause.

Like a procedure, a function has two parts: the specification and the body. The function specification begins with the keyword FUNCTION and ends with the RETURN clause, which specifies the datatype of the return value. Parameter declarations are optional. Functions that take no parameters are written without parentheses.

The function body begins with the keyword IS and ends with the keyword END followed by an optional function name.

The function body has three parts: a declarative part, an executable part, and an optional exception-handling part.

The RETURN statement immediately ends the execution of a subprogram and returns control to the caller. A subprogram can contain several RETURN statements.

Using the RETURN Statement

The RETURN statement immediately completes the execution of a subprogram and returns control to the caller. Execution then resumes with the statement following the subprogram call. (Do not confuse the RETURN statement with the RETURN clause in a function specification, which specifies the datatype of the return value.)

A subprogram can contain several RETURN statements, none of which need be the last lexical statement. Executing any of them completes the subprogram immediately. However, to have multiple exit points in a subprogram is a poor programming practice.

In procedures, a RETURN statement cannot contain an expression. The statement simply returns control to the caller before the normal end of the procedure is reached.

However, in functions, a RETURN statement must contain an expression, which is evaluated when the RETURN statement is executed. The resulting value is assigned to the function identifier, which acts like a variable of the type specified in the RETURN clause.

SQL Syntax :

```
> CREATE OR REPLACE FUNCTION FUNCTIONNAME(
[list of parameters])
RETURN return_datatype;
[IS | AS]
```

```
declaration section;
BEGIN
execution section;
EXCEPTION
exception section;
END;
```

Let's see a simple example to create a function.

```
CREATE OR REPLACE FUNCTION adder(n1 IN number, n2 IN number)
RETURN number
AS
    n3 number(8);
BEGIN
    n3:=n1+n2;
RETURN n3;
END;
/
```

Now write another program to call the function.

```
DECLARE
    n3 number(2);
BEGIN
    n3 = adder(11,22);
dbms_output.put_line('Addition is: ' || n3);
END;
/
```

Output ;

Addition is :33

EXAMPLE PROGRAMS USING FUNCTIONS

EXAMPLE-1

Write a PL/SQL function program using IN parameter

```
> DELIMITER //
> CREATE FUNCTION calcincome(value INT)
```



```
RETURNS INT
BEGIN
DECLARE income INT;
SET income = 0;
WHILE (income < = 20000) DO
SET income = income + value;
END WHILE;
RETURN income;
END; //
DELIMITER;
> select calcincome (8000);
```

Output:

16000

EXAMPLE-2

Write a PL/SQL function program in MYSQL without parameters

```
> CREATE FUNCTION fdemo()
RETURN int
AS
BEGIN
RETURN (SELECT SUM(marks) FROM student);
END;
/
```

EXAMPLE-3

Write a PL/SQL function program SQL without parameters

```
> CREATE FUNCTION totalemp()
    RETURN number IS
    total number(2) := 0;
BEGIN
    SELECT COUNT(*) INTO total
FROM employee;
```

```
    RETURN total;
    END;
    /
```

Output :

Function created

Calling a function :

```
> DECLARE
    c number(2);
BEGIN
    c: = totalemp(); —Calling function
    dbms_output.put_line('total employees =' || c);
END;
/
```

EXAMPLE-4

Write a PL/SQL function program SQL with parameters

```
CREATE OR REPLACE FUNCTION welcome_msg(p_name IN VARCHAR2) RETURN
VARCHAR2
IS
BEGIN
RETURN ('Welcome '|| p_name);
END;
/
```

Output :

Function created

Calling a function:

```
> DECLARE
    lv_msg VARCHAR2(30);
BEGIN
    lv_msg:= welcome_msg_func('Suresh');
    dbms_output.put_line(lv_msg);
END;
/
```


Output:

Welcome Suresh

Similarities between Procedure and Function

- Both can be called from other PL/SQL blocks.
- Both can have as many parameters as required.
- Both are treated as database objects in PL/SQL.

Comparing Procedures and Functions

S.No.	Datatype	Description
1	Execute as a PL/SQL statement	Invoke as part of an expression
2.	No RETURN datatype	Must contain a RETURN datatype
3.	Can return none, one or many values	Must return a single value
4.	Cannot use in SQL Statement	Can use in SQL Statements
5.	Used mainly to execute certain process	Used mainly to perform some calculation
6.	Use OUT parameter to return the value	Use RETURN to return the value
7.	It is not mandatory to return the value	It is mandatory to return the value
8.	RETURN will simply exit the control from subprogram.	RETURN will exit the control from subprogram and also returns the value
9.	Return datatype will not be specified at the time of creation	Return datatype is mandatory at the time of creation

Benefits of Stored Procedures and Functions

In addition to modularizing application development, stored procedures and functions have the following benefits:

Improved performance

- Avoid reparsing for multiple users by exploiting the shared SQL area.
- Avoid PL/SQL parsing at run-time by parsing at compile time.
- Reduce the number of calls to the database and decrease network traffic by bundling commands.

Improved maintenance

- *Modify routines online without interfering with other users.*
- *Modify one routine to affect multiple applications.*
- *Modify one routine to eliminate duplicate testing.*

Improved data security and integrity

- *Control indirect access to database objects from non privileged users with security privileges.*
- *Ensure that related actions are performed together, or not at all, by funneling activity for related tables through a single path.*

6.5 RECURSION WITH EXAMPLE PROGRAMS

PL/SQL does support recursion via function calls. Recursion is the act of a function calling itself, and a recursive call requires PL/SQL to create local copies of its memory structures for each call.

Recursion is a powerful technique for simplifying the design of algorithms. Basically, recursion means self-reference. In a recursive mathematical sequence, each term is derived by applying a formula to preceding terms. The Fibonacci sequence (0, 1, 1, 2, 3, 5, 8, 13, 21, ...), is an example. Each term in the sequence is the sum of the two terms that immediately precede it.

0+1=1 (3rd place)

1+1=2 (4th place)

2+1=3 (5th place) and So on

Sequence F(n) of Fibonacci series will have recurrence relation defined as "

F(n) = F(n-1) + F(n-2)

Where, F(0)=0 and F(1)=1 are always fixed

In a recursive definition, something is defined as simpler versions of itself. Consider the definition of n factorial (n!), the product of all integers from 1 to n:

n! = n * (n - 1)!

A recursive subprogram is one that calls itself. Each recursive call creates a new instance of any items declared in the subprogram, including parameters, variables, cursors, and exceptions.

EXAMPLE PROGRAMS

EXAMPLE-1

Find factorial of a number using Recursive function

```
CREATE OR REPLACE FUNCTION fact(x number)
RETURN number
IS
  f number;
BEGIN
  IF x=0 THEN
    f := 1;
ELSE
    f := x * fact(x-1);
  END IF;
RETURN f;
END;
```

Calling a function :

```
DECLARE
  num number;
  factorial number;
BEGIN
  num:= 6;
  factorial := fact(num);
  dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END;
```

Output :

```
Factorial 6 is 720
> SELECT fact(5) FROM DUAL;
> /
fact(5)
120
```

EXAMPLE-2

Find Fibonacci series without using Recursive function

```
-- declare variable a = 0 for first digit in a series
-- declare variable b = 0 for Second digit in a series
-- declare variable temp for first digit in a series
DECLARE
a number := 0;
b number := 1;
temp number;
n number := 10;
i number;
BEGIN
  dbms_output.put_line('Fibonacci series is: ');
  dbms_output.put_line(a);
  dbms_output.put_line(b);
  FOR i IN 2..n
  LOOP
    temp:= a + b;
    a := b;
    b := temp;
    dbms_output.put_line(temp);
  END LOOP;
LOOP;
/
```

Output :

```
Fibonacci series is: 0 1 1 2 3 5 8 13 21 34
```

EXAMPLE-3

Find Fibonacci series using Recursive function

```
CREATE OR REPLACE FUNCTION fib(n INTEGER) RETURN INTEGER IS
BEGIN
```



```
IF (n=1) OR (n=2) THEN -- terminating condition
RETURN 1;
ELSE
RETURN fib(n-1) + fib(n-2); -- recursive call
END IF;
END;
/
```

Output :

```
> SELECT fib(1), fib(2), fib(3), fib(4), fib(5) FROM DUAL;
1 2 3 5 8
```

6.6 STORED PROCEDURES

A **stored procedure** is one which performs one or more specific tasks in PL/SQL block. This is similar to a procedure in other programming languages. A stored procedure in PL/SQL is nothing but a series of declarative SQL statements which can be stored in the database catalogue. A procedure can be thought of as a function or a method. They can be invoked through triggers, other procedures, or applications on Java, PHP etc.

A procedure consists of header and a body.

- **Header** consists of name of the procedure, parameters (or) variables passed to the procedure.
- **Body** consists of declaration section, execution section and exception section similar to a general PL/SQL block.

All the statements of a block are passed to Oracle engine all at once which increases processing speed and decreases the traffic.

Advantages :

- They result in performance improvement of the application. If a procedure is being called frequently in an application in a single connection, then the compiled version of the procedure is delivered.
- They reduce the traffic between the database and the application, since the lengthy statements are already fed into the database and need not be sent again and again via the application.

- They add to code reusability, similar to how functions and methods work in other languages such as C/C++ and Java.

Disadvantages :

- Stored procedures can cause a lot of memory usage. The database administrator should decide an upper bound as to how many stored procedures are feasible for a particular application.
- MySQL does not provide the functionality of debugging the stored procedures.

Syntax for creating stored procedure :

```
CREATE [OR REPLACE] PROCEDURE procedure_name[(parameter [,parameter])]
[IS | AS]
   [declaration_section]
BEGIN
   executable_section
[EXCEPTION
   exception_section]
END [procedure_name];
```

Create stored procedure example :

In this example, we are going to insert record in user table. So you need to create user table first.

Table creation:

```
CREATE TABLE User(id number(10) primary key, name varchar2(100));
```

Now write the procedure code to insert record in user table.

```
CREATE OR REPLACE PROCEDURE insertuser
(id IN NUMBER,
name IN VARCHAR2)
IS
BEGIN
INSERT INTO User VALUES(id, name);
END;
/
```


Output :

```
Procedure created.
```

PL/SQL program to call procedure

```
BEGIN
  insertuser(101,'Suresh');
  dbms_output.put_line('record inserted successfully');
END;
/
```

Now, see the "User" table, you will see one record is inserted.

ID	Name
101	Suresh

PL/SQL Drop Stored procedure

Syntax

```
DROP PROCEDURE procedure_name;
```

Example :

```
DROP PROCEDURE insertuser;
```

6.7 EXCEPTION HANDLING

An error that occurs during the program execution is called **Exception** in PL/SQL. PL/SQL facilitates programmers to catch such conditions using exception block in the program and an appropriate action is taken against the error condition.

There are two types of exceptions :

1. System-defined (pre-defined) Exceptions
2. User-defined Exceptions

Syntax for exception handling : Following is a general syntax for exception handling:

```
DECLARE
    <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
```

```
    <exception handling goes here >
    WHEN exception1 THEN
      exception1-handling-statements
    WHEN exception2  THEN
      exception2-handling-statements
    WHEN exception3 THEN
      exception3-handling-statements
    .....
    WHEN others THEN
      exception3-handling-statements
END;
```

EXAMPLE-1

```
SET SERVEROUTPUT ON;
DECLARE
  a int;
  b int;
  c int;
BEGIN
  a := &a;
  b := &b;
  c := a/b;
  dbms_output.put_line('RESULT=' || c);
EXCEPTION
WHEN ZERO_DIVIDE THEN
  dbms_output.put_line('Division by 0 is not possible');
END;
```

Output :

```
Enter the value for a: 10
Enter the value for b: 0
Division by 0 is not possible
```


EXAMPLE-2

Let's take another example to demonstrate the concept of exception handling. Here we are using the already created CUSTOMERS table.

SELECT* FROM COUSTOMERS;

ID	NAME	AGE	ADDRESS	SALARY
1.	Ramesh	23	Allahabad	20000
2.	Suresh	22	Kampur	22000
3.	Mahesh	24	Ghaziabad	24000
4.	Chandan	25	Noida	26000
5.	Alex	21	Paris	28000
6.	Sunita	20	Delhi	30000

```
DECLARE
    c_id customers.id%type := 8;
    c_name  customers.name%type;
    c_addr customers.address%type;
BEGIN
  SELECT name, address INTO c_name, c_addr
  FROM customers
  WHERE id = c_id;
DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
 DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
 EXCEPTION
   WHEN no_data_found THEN
     dbms_output.put_line('No such customer!');
   WHEN others THEN
     dbms_output.put_line('Error!');
 END;
 /
```

After the execution of above code at SQL Prompt, it produces the following result:

```
No such customer!
```

The above program should show the name and address of a customer as result whose ID is given. But there is no customer with ID value 8 in our database, so the program raises the run-time exception NO_DATA_FOUND, which is captured in EXCEPTION block.

If you use the id defined in the above table (i.e. 1 to 6), you will get a certain result. For a demo example: here, we are using the id 5. In above example just change c_id customers.id%type; = 2.

After the execution of above changed code at SQL prompt, you will get the following result:

```
Name: Suresh
Address: Kanpur
```

Raising Exceptions :

In the case of any internal database error, exceptions are raised by the database server automatically. But it can also be raised explicitly by programmer by using command RAISE.

User-defined Exceptions :

PL/SQL facilitates their users to define their own exceptions according to the need of the program. A user-defined exception can be raised explicitly, using either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

User defined exceptions are in general defined to handle special cases where our code can generate exception due to our code logic. Also, in your code logic, you can explicitly specify to generate an exception using the RAISE keyword and then handle it using the EXCEPTION block.

Syntax for user defined exceptions :

```
DECLARE
  <exception_name> EXCEPTION
BEGIN
  <sql sentence>
  IF <test_condition> THEN
    RAISE <exception_name>;
  END IF;
```



```
EXCEPTION
  WHEN <exception_name> THEN
  statement;
END;
```

EXAMPLE-3

User defined exception

Let's take an example to understand how to use user-defined exception. Below we have a simple example,

RollNo	SNAME	Total Courses
11.	Anu	2
12.	Asha	1
13.	Arpit	3
14.	Chtan	1

In the PL/SQL program below, we will be using the above table student to demonstrate the use of User-defined Exception.

```
SET SERVEROUTPUT ON;
DECLARE
  sno student.rollno%type;
  snm student.sname%type;
  crno student.total_course%type;
  invalid_total EXCEPTION;
BEGIN
  sno := &rollno;
  snm := '&sname';
  crno:=total_courses;
  IF (crno > 3) THEN
    RAISE invalid_total;
  END IF;
  INSERT into student values(sno, snm, crno);
```

```
EXCEPTION
  WHEN invalid_total THEN
  dbms_output.put_line ('Total number of courses cannot be more than 3 ');
END;
```

Output:

```
Enter the value for eno: 16
Enter the value for enm: Akash
Enter the value for cno: 5
Total number of courses cannot be more than 3
```

In the above program, User-defined exception called invalid_total is used which is generated when total number of courses is greater than 3 (when a student can be enrolled maximum in 3 courses).

PL/SQL Pre-defined Exceptions :

There are many pre-defined exceptions in PL/SQL which are executed when any database rule is violated by the programs.

For Example : NO_DATA_FOUND is a pre-defined exception which is raised when a SELECT INTO statement returns no rows.

Following is a list of some important pre-defined exceptions :

S.No.	Exception	Oracle Error	SQL Code	Description
1	ACCESS_INTO_NULL	06530	-6530	It is raised when a NULL object is automatically assigned a value
2	CASE_NOT_FOUND	06592	-6592	It is raised when none of the choices in the "WHEN" clauses of a CASE statement is selected, and there is no else clause
3	DUP_VAL_ON_INDEX	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index
4	INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor

5	INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number
6	LOGIN_DENIED	01017	-1017	It is raised when program attempts to log on to the database with an invalid username or password
7	NO_DATA_FOUND	01403	+100	It is raised when a select into statement returns no rows
8	NOT_LOGGED_ON	01012	-1012	It is raised when a database call is issued without being connected to the database
9	PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an internal problem
10	ROWTYPE_MISMATCH	06504	-6504	It is raised when a cursor fetches value in a variable having incompatible data type
11	STORAGE_ERROR	06500	-6500	It is raised when PL/SQL ran out of memory or memory was corrupted
12	TOO_MANY_ROWS	01422	-1422	It is raised when a SELECT INTO statement returns more than one row
13	VALUE_ERROR	06502	-6502	It is raised when an arithmetic, conversion, truncation, or size-constraint error occurs
14	ZERO_DIVIDE	01476	-1476	It is raised when an attempt is made to divide a number by zero

6.8 CURSOR WITH EXAMPLE PROGRAMS

When an SQL statement is processed, Oracle creates a memory area known as context area. A **cursor** is a pointer to this context area. It contains all information needed for processing the statement. In PL/SQL, the context area is controlled by Cursor. A cursor contains information on a select statement and the rows of data accessed by it.

A cursor is used to referred to a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors :

- Implicit Cursors
- Explicit Cursors.

1. **PL/SQL Implicit Cursors :** The implicit cursors are automatically generated by Oracle while an SQL statement is executed, if you don't use an explicit cursor for the statement.

These are created by default to process the statements when DML statements like INSERT, UPDATE, DELETE etc. are executed.

Oracle provides some attributes known as implicit cursors attributes to check the status of DML operations. Some of them are: %FOUND, %NOTFOUND, %ROWCOUNT and %ISOPEN.

For example : When you execute the SQL statements like INSERT, UPDATE, DELETE then the cursor attributes tell whether any rows are affected and how many have been affected. If you run a SELECT INTO statement in PL/SQL block, the implicit cursor attribute can be used to find out whether any row has been returned by the SELECT statement. It will return an error if there no data is selected.

The following table specifies the status of the cursor with each of its attribute.

Attribute	Description
%FOUND	Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect at least one row or more rows or a SELECT INTO statement returned one or more rows. Otherwise it returns FALSE.
%NOTFOUND	Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect no row, or a SELECT INTO statement return no rows. Otherwise it returns FALSE. It is a just opposite of %FOUND.
%ISOPEN	It always returns FALSE for implicit cursors, because the SQL cursor is automatically closed after executing its associated SQL statements.
%ROWCOUNT	It returns the number of rows affected by DML statements like INSERT, DELETE, and UPDATE or returned by a SELECT INTO statement.

PL/SQL Implicit Cursor Example :

Create customers table and have records :

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Let's execute the following program to update the table and increase salary of each customer by 5000. Here, SQL%ROWCOUNT attribute is used to determine the number of rows affected :

Create procedure :

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE  customers
  SET salary = salary + 5000;
  IF sql%notfound THEN
    dbms_output.put_line('no customers updated');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' customers updated ');
  END IF;
END;
/
```

Output :

```
6 customers updated
PL/SQL procedure successfully completed.
```

Now, if you check the records in customer table, you will find that the rows are updated.

select * from customers;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	25000
2	Suresh	22	Kanpur	27000
3	Mahesh	24	Ghaziabad	29000
4	Chandan	25	Noida	31000
5	Alex	21	Paris	33000
6	Sunita	20	Delhi	35000

2. **PL/SQL Explicit Cursors :** The Explicit cursors are defined by the programmers to gain more control over the context area. These cursors should be defined in the declaration section of the PL/SQL block. It is created on a SELECT statement which returns more than one row.

Following is the syntax to create an explicit cursor :

Syntax of explicit cursor

Following is the syntax to create an explicit cursor:

CURSOR cursor_name IS select_statement;

Steps : You must follow these steps while working with an explicit cursor.

1. Declare the cursor to initialize in the memory.
2. Open the cursor to allocate memory.
3. Fetch the cursor to retrieve data.
4. Close the cursor to release allocated memory.

(i) **Declare the cursor :** It defines the cursor with a name and the associated SELECT statement.

Syntax for explicit cursor declaration "

```
CURSOR name IS
    SELECT statement;
```

(ii) **Open the cursor :** It is used to allocate memory for the cursor and make it easy to fetch the rows returned by the SQL statements into it.

Syntax for cursor open :

```
OPEN cursor_name;
```

(iii) **Fetch the cursor :** It is used to access one row at a time. You can fetch rows from the above-opened cursor as follows :

Syntax for cursor fetch :

```
FETCH cursor_name INTO variable_list;
```

(iv) **Close the cursor :** It is used to release the allocated memory. The following syntax is used to close the above-opened cursors.

Syntax for cursor close :

```
Close cursor_name;
```

PL/SQL Explicit Cursor Example : Explicit cursors are defined by programmers to gain more control over the context area. It is defined in the declaration section of the PL/SQL block. It is created on a SELECT statement which returns more than one row.

Let's take an example to demonstrate the use of explicit cursor. In this example, we are using the already created CUSTOMERS table.

Create customers table and have records :

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Create Procedure : Execute the following program to retrieve the customer name and address.

```
DECLARE
  c_id customers.id%type;
  c_name customers.name%type;
  c_addr customers.address%type;
CURSOR c_customers is
    SELECT id, name, address FROM customers;
```

```
BEGIN
  OPEN c_customers;
  LOOP
    FETCH c_customers into c_id, c_name, c_addr;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
  CLOSE c_customers;
END;
/
```

Output :

```
1. Ramesh Allahabad        2. Suresh Kanpur
3. Mahesh Ghaziabad        4. Chandan Noida
5. Alex Paris              6. Sunita Delhi.
PL/SQL procedure successfully completed.
```

EXAMPLE PROGRAMS

Implicit Cursors Example Programs

Consider EMP table with schema EMP (EMPNO, ENAME, JOB, SAL)

EXAMPLE-1

Print no. of rows deleted from emp table.

```
DECLARE
  ROW_DEL_NO NUMBER;
BEGIN
  DELETE FROM EMP;
  ROW_DEL_NO:= SQL%ROWCOUNT;
  dbms_output.put_line(No. of rows deleted are:|| ROW_DEL_NO);
END;
/
```

Output :

```
SQL> /
No. of rows deleted are: 5
```


EXAMPLE-2

Accept empno and print its details (using cursor)

```
DECLARE
  V_NO EMP.EMPNO%TYPE:=&V_NO;
  V_NAME EMP.ENAME%TYPE;
  V_JOB EMP.JOB%TYPE;
  V_SAL EMP.SAL%TYPE;
BEGIN
  SELECT ename, job, sal INTO V_NAME,V_JOB,V_SAL
  FROM EMP WHERE empno=V_NO;
  IF SQL%FOUND THEN —SQL%FOUND is true if empno=v_no
  dbms_output.put_line(V_NAME ||''||V_JOB||''||V_SAL);
EXCEPTION
WHEN NO_DATA_FOUND THEN
  dbms_output.put_line('Empno does not exists');
END;
```

Output :

```
SQL>/
Enter value for v_no: 34
Empno does not exists
SQL>/
Enter value for v_no: 7801
SMITH CLERK 8000
```

Explicit Cursor Example Program

```
DECLARE
  CURSOR c_deptno IS SELECT ename, sal, deptno
  FROM EMP;
  v_name emp.ename%type;
  v_sal emp.sal%type;
  v_deptno emp.deptno%type;
BEGIN
  OPEN c_deptno;
```

```
    FETCH c_deptno INTO v_name,v_sal,v_deptno;
    dbms_output.put_line(v_name ||''||v_deptno||''||v_sal);
    CLOSE c_deptno;
  END;
```

Output:

```
SQL > /
SMITH 20 8000
```

Using Cursor FOR.... LOOP

In the cursor FOR LOOP, the result of SELECT query is used to determine the number of times the loop is executed. In a Cursor FOR loop, the opening, fetching and closing of cursors is performed implicitly. When you use it, Oracle automatically declares a variable with the same name as that is used as a counter in the FOR LOOP. Just precede the name of the selected field with the name of this variable to access its contents.

For example :

```
DECLARE
  CURSOR c_deptno IS SELECT ename, sal, deptno
  FROM EMP;
BEGIN
  FOR x IN c_deptno
  LOOP
  dbms_output.put_line(x.ename ||' '||x.deptno||' '||x.sal);
  END LOOP;
End;
```

In above example a Cursor FOR loop is used, there is no open and fetch command. The command FOR x IN c_deptno implicitly opens the c_deptno cursor and fetches a value into the x variable. Note that x is not explicitly declared in the block. When no more records are in the cursor, the loop is exited and cursor is closed.

There is no need to check the cursor%NOTFOUND attribute-that is automated via the cursor FOR loop. And also there is no need of close command.

Explicit Cursor in MySQL using functions :

```
mysql>delimiter #;
mysql> CREATE TABLE SITE(SITE_ID INT, SITE_NAME VARCHAR(20));
-> #
```



```
Query OK, 0 rows affected (0.20 sec)
mysql> INSERT INTO SITE VALUES(26,'MASAB TANK'),(48,'HUDA
COLONY'),(1,'HYDERABAD');
-> #
Query OK, 3 rows affected (0.04 sec) Records: 3 Duplicates: 0 Warnings: 0

mysql> SELECT * FROM SITE;
-> #
+ ———+————————+
| SITE_ID | SITE_NAME      |
+ ———+————————+
|      26 | MASAB TANK |
|      48 | HUDA COLONY |
|       1 | HYDERABAD    |
+ ———+————————+
3 rows in set (0.00 sec)
mysql> create function findsiteid(name_in varchar(50))
-> returns int
-> begin
-> declare siteid int default 0;
-> declare c1 cursor for
-> select site_id from site
-> where site_name=name_in;
-> open c1;
-> fetch c1 into siteid;
-> close c1;
-> return siteid;
-> end;
-> #
Query OK, 0 rows affected (0.57 sec)
mysql> select findsiteid('hyderabad');
-> #
```

```
+-----------------------+
| findsiteid('hyderabad') |
+-----------------------+
|                     1 |
+-----------------------+
mysql> select findsiteid('hyder');
-> #
+-----------------------+
| findsiteid('hyder') |
+-----------------------+
|                     0 |
+-----------------------+
mysql> select findsiteid('masabtank');
-> #
+-----------------------+
| findsiteid('masabtank') |
+-----------------------+
|                     0 |
+-----------------------+
mysql> select findsiteid('MASAB TANK');
-> #
+-----------------------+
| findsiteid('MASAB TANK') |
+-----------------------+
|                    26 |
+-----------------------+
```

Advantages of cursors :

1. Allowing to position at specific rows of the result set.
2. Returning one row or block of rows from the current position in the result set.
3. Supporting data modification to the rows at the current position in the result set.

6.9 TRIGGERS WITH EXAMPLE PROGRAMS

Trigger is invoked by Oracle engine automatically whenever a specified event occurs. Trigger is stored into database and invoked repeatedly, when specific condition match.

Triggers are stored programs, which are automatically executed or fired when some event occurs.

Triggers are written to be executed in response to any of the following events.

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
- Database definition (DDL) statements (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers could be defined on the table, view, schema, or database with which the event is associated.

Advantages of Triggers : These are the following advantages of Triggers,

- Trigger generates some derived column values automatically.
- Enforces referential integrity.
- Event logging and storing information on table access.
- Auditing.
- Synchronous replication of tables.
- Imposing security authorizations.
- Preventing invalid transactions.

Creating a trigger :

Syntax for creating trigger :

```
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
  Declaration-statements
BEGIN
  Executable-statements
EXCEPTION
  Exception-handling-statements
END;
```

Here :

- **CREATE [OR REPLACE] TRIGGER trigger_name :** It creates or replaces an existing trigger with the trigger_name.
- **{BEFORE | AFTER | INSTEAD OF} :** This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
- **{INSERT [OR] | UPDATE [OR] | DELETE}** : This specifies the DML operation.
- **[OF col_name] :** This specifies the column name that would be updated.
- **[ON table_name] :** This specifies the name of the table associated with the trigger.
- **[REFERENCING OLD AS o NEW AS n] :** This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
- **[FOR EACH ROW] :** This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- **WHEN (condition) :** This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

PL/SQL Trigger Example : Let's take a simple example to demonstrate the trigger. In this example, we are using the following CUSTOMERS table :

Create table and have records :

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Create Trigger : Let's take a program to create a row level trigger for the CUSTOMERS table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values :

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
sal_diff number;
  BEGIN
sal_diff := :NEW.salary  - :OLD.salary;
dbms_output.put_line('Old salary: ' || :OLD.salary);
dbms_output.put_line('New salary: ' || :NEW.salary);
dbms_output.put_line('Salary difference: ' || sal_diff);
  END;
  /
```

After the execution of the above code at SQL Prompt, it produces the following result.

```
Trigger created.
```

Check the salary difference by procedure :

Use the following code to get the old salary, new salary and salary difference after the trigger created.

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE  customers
  SET salary = salary + 5000;
  IF sql%notfound THEN
    dbms_output.put_line('no customers updated');
  ELSIF sql%found THEN
    total_rows := sql%rowcount;
    dbms_output.put_line( total_rows || ' customers updated ');
  END IF;
END;
/
```

Output :

```
Old salary: 20000
New salary: 25000
Salary difference: 5000
Old salary: 22000
New salary: 27000
Salary difference: 5000
Old salary: 24000
New salary: 29000
Salary difference: 5000
Old salary: 26000
New salary: 31000
Salary difference: 5000
Old salary: 28000
New salary: 33000
Salary difference: 5000
Old salary: 30000
```



```
New salary: 35000
Salary difference: 5000
6 customers updated
```

Note : As many times you executed this code, the old and new both salary is incremented by 5000 and hence the salary difference is always 5000.

After the execution of above code again, you will get the following result.

```
Old salary: 25000
New salary: 30000
Salary difference: 5000
Old salary: 27000
New salary: 32000
Salary difference: 5000
Old salary: 29000
New salary: 34000
Salary difference: 5000
Old salary: 31000
New salary: 36000
Salary difference: 5000
Old salary: 33000
New salary: 38000
Salary difference: 5000
Old salary: 35000
New salary: 40000
Salary difference: 5000
6 customers updated
```

Important Points : Following are the two very important point and should be noted carefully.

- OLD and NEW references are used for record level triggers these are not available for table level triggers.

- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.

EXAMPLE PROGRAMS

EXAMPLE-1

Access the value of column inside a trigger

```
SQL> CREATE TRIGGER tr_sal
     BEFORE INSERT ON EMP
     FOR EACH ROW
     BEGIN
     IF :NEW.sal = 0 THEN
     RAISE_APPLICATION_ERROR('-20010','Salary should be greater than 0');
     END IF;
     END;
```

Output:

```
SQL> /
Trigger created.
```

When you insert data into an emp table with salary 0 at that time this trigger will get executed.

Modifying a Trigger :

A trigger can be modified using OR REPLACE clause of CREATE TRIGGER command.

EXAMPLE-2

SQL> CREATE OR REPLACE TRIGGER tr_sal

```
     BEFORE INSERT ON EMP
     FOR EACH ROW
     BEGIN
     IF :NEW.sal <= 0 THEN
     RAISE_APPLICATION_ERROR('-20010','Salary should be greater than 0');
     END IF;
     END;
```


Output :

```
SQL> /
Trigger created.
```

When you insert data into an emp table with salary 0 or less than 0 at that time this trigger will get executed.

Enabling or Disabling a Trigger :

To enable or disable a specific trigger, ALTER TRIGGER command is used.

Syntax :

```
ALTER TRIGGER trigger_name ENABLE/DISABLE;
```

When a trigger is created, it is automatically enabled and it gets executed according to the triggering command. To disable the trigger, use DISABLE option as:

```
ALTER TRIGGER tr_sal DISABLE;
```

To enable or disable all the triggers of a table, ALTER TABLE command is used.

Syntax :

```
ALTER TABLE table_name ENABLE / DISABLE ALL TRIGGERS;
```

Ex : ALTER TABLE emp DISABLE ALL TRIGGERS;

Deleting a Trigger

To delete a trigger, use DROP TRIGGER command.

Syntax :

```
DROP TRIGGER trigger_name;
```

Ex: DROP TRIGGER tr_sal;

Triggers in MySQL

```
mysql> CREATE TABLE Account(num int, amount decimal(10,2));
mysql> CREATE TRIGGER ins_sum
    -> BEFORE INSERT ON Account
    -> FOR EACH ROW
    -> @sum=@sum+NEW.amount;
mysql> SET @sum=0;
mysql>INSERT INTO  Account  VALUES(100,20000.50),(200,30000.50),(300,40000.80);
Query OK, 3 rows affected (0.18 sec)
```

```
mysql> SELECT @sum;
+----------+
| @sum     |
+----------+
| 90001.80 |
+----------+
mysql> DELIMITER #
mysql> CREATE TRIGGER update_check
    -> BEFORE UPDATE ON Account
    -> FOR EACH ROW
    -> BEGIN
    -> IF NEW.amount<0 THEN
    -> SET NEW.amount=0;
    -> ELSEIF NEW.amount>1000 THEN
    -> SET NEW.amount=40000;
    -> END IF;
    -> END;
    -> #
mysql> SELECT * FROM Account;
    -> #
+-----+----------+
| num | amount   |
+-----+----------+
| 100 | 20000.50 |
| 200 | 30000.50 |
| 300 | 40000.80 |
+-----+----------+
mysql> UPDATE Account SET amount=2000 WHERE num=100;
    -> #
Query OK, 1 row affected (0.17 sec)
mysql> SELECT * FROM Account;
    -> #
```



```
+-----+----------+
| num | amount   |
+-----+----------+
| 100 | 40000.00 |
| 200 | 30000.50 |
| 300 | 40000.80 |
+-----+----------+
mysql> UPDATE Account SET amount=-2000 WHERE num=100;
    -> #
Query OK, 1 row affected (0.08 sec)
mysql> SELECT * FROM Account;
    -> #
+-----+----------+
| num | amount   |
+-----+----------+
| 100 |     0.00 |
| 200 | 30000.50 |
| 300 | 40000.80 |
+-----+----------+
```

6.10 PACKAGES WITH EXAMPLE PROGRAMS

Packages : Packages are schema objects that groups logically related PL/SQL types, variables and subprograms. A package will have two mandatory parts,

(i) Package specification.

(ii) Package body or definition.

Package Specification : It is the interface to the package. It just declares the types, variables, constant, exceptions, cursors and sub-programs that can be referenced from outside the package. In otherwords, it contains all information about the content of the package. All objects placed in specification are public objects.

Syntax :

(Creating a package specification)

```
> CREATE [OR REPLACE] PACKAGE package_name
[IS/AS]
SUB program and public element declaration
.....
END package_name;
```

Ex :

```
> CREATE [OR REPLACE] PACKAGE emp–sal
AS
PROCEDURE final_sal (sid employee,id%type);
.....
END emp_sal;
```

Advantages of Packages :

(i) Modularity
(ii) Easier application design
(iii) Information hiding
(iv) Added functionality
(v) Better performance.

Package body : The package body has the codes for various methods declared in the package specification.

The characteristics of a package body are :

(i) It should contain definition for all the subprogram or cursors that have been declared in the specification.

(ii) It can also have more subprograms or other element that are not declared in specification. These are called private elements.

(iii) The first part of the package is global dedaration part. The last part of package is package initialization part that execute one time whenever a package is reffered first time in the session

Syntax :

```
> CREATE [OR REPLACE] PACKAGE BODY package_name
  [IS/AS]
```



```
< global_declaration part>
< private element definition>
< sub program and public element definition>
.....
EXCEPTION
.....
< package Initialization>
END package_name;
```

Example :

```
> CREATE OR REPLACE PACKAGE BODY emp_sal
AS
PROCEDURE find_sal (sid employee.id%type)
IS
esal employee.salary%type;
BEGIN
SELECT salary INTO esal
FROM employee WHERE id = sid;
dbms_output.put_line('salary =' || esal);
END find_sal;
END emp_sal;
```

Output :

```
Package body created
```

Using the package element : The package element (variables, procedures on functions) are accussed with the following syntax

```
> PACKAGE name.elementname;
```

Ex :

```
> DECLARE
Code employee.id % type : = &sid;
BEGIN
emp_sal.find_sal(code);
END;
/
```

Output :

```
Enter value for sid : 1          Old 2 : code employee, id% type = & sid;
Salary = 20000                   new 2 : code employee id% type : = 1;
```

PL/SQL Procedure successfully completed.

EXAMPLE PROGRAMS

SQL> SELECT * FROM students;

ID	NAME	MARKS
1	Rishi	789
2.	Jashith	998
3.	Tarun	886
4.	Bhargavi	965

Package Specification :

```
SQL> CREATE OR REPLACE PACKAGE students_marks AS
 2    PROCEDURE find_marks(sid int);
 3    END students_marks;
 4    /
```

Package created.

Package body

```
SQL> CREATE OR REPLACE PACKAGE BODY students_marks AS
 2    PROCEDURE find_marks(sid int)
 3    IS
 4    smarks int;
 5    BEGIN
 6    SELECT marks INTO smarks FROM students WHERE id=sid;
 7    dbms_output.put_line('Smarks = '|| smarks);
 8    END find_marks;
 9    END students_marks;
10    /
```


Package body created.

Using the Package element

```
SQL> DECLARE
 2  sis int:=&sid;
 3  BEGIN
 4   students_marks.find_marks(sis);
 5  END;
 6  /
```

Output :

```
Enter value for sid : 1
Smarks = 789
```