Pract 1

Writing PL/SQL Blocks with basic programming constructs by including following:
a. Sequential Statements
b. unconstrained loop

PL/SQL Block Example

```
DECLARE
   v_sum NUMBER := 0;  -- Variable to store the sum
   v_counter INTEGER;   -- Counter for the loop
BEGIN
   -- Sequential Statement: Initialize counter
   v_counter := 1;

   -- Unconstrained Loop: Calculate the sum of the first 10 natural numbers
   LOOP
      v_sum := v_sum + v_counter;  -- Add counter to sum

      IF v_counter = 10 THEN
         EXIT;  -- Exit the loop when the counter reaches 10
      END IF;

      v_counter := v_counter + 1;  -- Increment the counter
   END LOOP;

   -- Sequential Statement: Output the result
   DBMS_OUTPUT.PUT_LINE('The sum of the first 10 natural numbers is: ' || v_sum);
END;
/
```

Output

```
The sum of the first 10 natural numbers is: 55
```

Pract 2

Sequences:
a. Creating simple Sequences with clauses like START WITH, INCREMENT BY, MAXVALUE, MINVALUE, CYCLE | NOCYCLE, CACHE | NOCACHE, ORDER | NOORECER.
b. Creating and using Sequences for tables.

1. Creating a Sequence

Here's an example of how to create a sequence using various clauses:

sql

```
CREATE SEQUENCE my_sequence
   START WITH 1          -- Starting value
   INCREMENT BY 1        -- Increment value
   MINVALUE 1            -- Minimum value
   MAXVALUE 1000         -- Maximum value
   CYCLE                 -- Allow cycling back to the minimum value
   CACHE 10;             -- Cache 10 sequence numbers for performance
```

2. Explanation of Clauses

START WITH**: Specifies the first number that will be generated.

INCREMENT BY**: Defines the interval between consecutive numbers in the sequence.

MINVALUE**: Sets the minimum value the sequence can generate.

MAXVALUE**: Sets the maximum value the sequence can generate.

CYCLE**: If specified, the sequence will start again from the `MINVALUE` once the `MAXVALUE` is reached.

NOCYCLE**: Prevents cycling, which is the default behavior.

CACHE**: Preallocates a specified number of sequence numbers in memory for performance.

NOCACHE**: Does not cache sequence numbers, which can be slower.

ORDER**: Guarantees that numbers are generated in the order requested (not typically needed).

NOORDER**: Allows the numbers to be generated out of order, which is the default behavior.

B. Creating and Using Sequences for Tables

1. Creating a Table

First, let’s create a simple table that will use the sequence:
sql

```
CREATE TABLE employees (
   employee_id NUMBER PRIMARY KEY,
   first_name VARCHAR2(50),
   last_name VARCHAR2(50)
);
```

2. Inserting Data Using the Sequence

You can use the sequence to generate unique `employee_id` values when inserting new records:

sql

~~~
INSERT INTO employees (employee_id, first_name, last_name)
VALUES (my_sequence.NEXTVAL, 'John', 'Doe');

INSERT INTO employees (employee_id, first_name, last_name)
VALUES (my_sequence.NEXTVAL, 'Jane', 'Smith');

INSERT INTO employees (employee_id, first_name, last_name)
VALUES (my_sequence.NEXTVAL, 'Alice', 'Johnson');
```
~~~

3. Selecting Data from the Table

To see the records you've inserted, run:

`sql

```
SELECT * FROM employees;
```

Output

If you run the above `SELECT` statement after inserting three records, you might see:

```
EMPLOYEE_ID  FIRST_NAME  LAST_NAME

-----------  -----------  ----------

1          John       Doe

2          Jane       Smith

3          Alice      Johnson
```

Pract 3

Writing PL/SQL Blocks with basic programming constructs by including following:

a. If...then...Else, IF...ELSIF...ELSE... END IF

b. Case statement

```
DECLARE
   v_score NUMBER := 85;  -- Example score
   v_grade CHAR(1);       -- Variable to store the grade
   v_feedback VARCHAR2(100); -- Variable for feedback
```

```
BEGIN
   -- Using IF...THEN...ELSE
   IF v_score >= 90 THEN
      v_grade := 'A';
      v_feedback := 'Excellent work!';
   ELSIF v_score >= 80 THEN
      v_grade := 'B';
      v_feedback := 'Good job!';
   ELSIF v_score >= 70 THEN
      v_grade := 'C';
      v_feedback := 'You passed.';
   ELSIF v_score >= 60 THEN
      v_grade := 'D';
      v_feedback := 'Needs improvement.';
   ELSE
      v_grade := 'F';
      v_feedback := 'Failed. Please try again.';
   END IF;

   -- Output the grade and feedback
   DBMS_OUTPUT.PUT_LINE('Score: ' || v_score);
   DBMS_OUTPUT.PUT_LINE('Grade: ' || v_grade);
   DBMS_OUTPUT.PUT_LINE('Feedback: ' || v_feedback);

   -- Using CASE statement
   CASE v_grade
      WHEN 'A' THEN
         DBMS_OUTPUT.PUT_LINE('Outstanding performance!');
      WHEN 'B' THEN
         DBMS_OUTPUT.PUT_LINE('Well done!');
      WHEN 'C' THEN
         DBMS_OUTPUT.PUT_LINE('You did it!');
      WHEN 'D' THEN
         DBMS_OUTPUT.PUT_LINE('Keep working at it!');
      WHEN 'F' THEN
         DBMS_OUTPUT.PUT_LINE('Don’t give up!');
      ELSE
         DBMS_OUTPUT.PUT_LINE('Invalid grade.');
   END CASE;

END;
/
```

Output

```
Score: 85
Grade: B
Feedback: Good job!
Well done!
```

Explanation

1. Variable Declarations:

   - `v_score`: This variable holds the score that will be evaluated.
   - `v_grade`: This variable will store the corresponding letter grade.
   - `v_feedback`: This variable will hold feedback based on the score.
2. IF...THEN...ELSE Structure:
   - The block checks the value of `v_score` and assigns a grade and feedback accordingly.
   - It uses `ELSIF` to evaluate multiple conditions.
3. Output the Results:
   - The grade and feedback are displayed using `DBMS_OUTPUT.PUT_LINE`.
4. CASE Statement:
   - After determining the grade, a `CASE` statement is used to print a message based on the grade.
   - Each `WHEN` clause corresponds to a specific grade and provides a different message.

Pract 4

Writing PL/SQL Blocks with basic programming constructs for following Iterative

Structure:

a. While-loop Statements

b. For-loop Statements.

A. WHILE Loop

use a `WHILE` loop to calculate the factorial of a number.
Plsql

```
DECLARE
   v_number NUMBER := 5;      -- Number to calculate the factorial of
   v_factorial NUMBER := 1;   -- Variable to hold the factorial result
   v_counter NUMBER := 1;     -- Counter for the loop
BEGIN
   -- While loop to calculate factorial
   WHILE v_counter <= v_number LOOP
      v_factorial := v_factorial * v_counter; -- Multiply factorial by counter
      v_counter := v_counter + 1;             -- Increment the counter
   END LOOP;

   -- Output the result
   DBMS_OUTPUT.PUT_LINE('Factorial of ' || v_number || ' is: ' || v_factorial);
END;
/
```

B. FOR Loop Example

use a `FOR` loop to calculate the sum of the first 10 natural numbers.

```
plsql
DECLARE
   v_sum NUMBER := 0; -- Variable to hold the sum
BEGIN
   -- For loop to calculate the sum of the first 10 natural numbers
   FOR v_counter IN 1..10 LOOP
      v_sum := v_sum + v_counter; -- Add the counter to the sum
   END LOOP;

   -- Output the result
   DBMS_OUTPUT.PUT_LINE('The sum of the first 10 natural numbers is: ' || v_sum);
END;
/
```

Explanation

1.WHILE Loop
   - The `WHILE` loop continues executing as long as the condition `v_counter <= v_number` is true.
   - Inside the loop, we multiply the current factorial value by the counter and then increment the counter until it exceeds the specified number.

2. FOR Loop
   - The `FOR` loop iterates over a defined range (from 1 to 10 in this case).
   - Each iteration adds the current counter value to the sum.

3. Output
   - Both blocks output the results using `DBMS_OUTPUT.PUT_LINE`.

Outputs

1. WHILE Loop Output(for `v_number` = 5):

```
Factorial of 5 is: 120
```

2.FOR Loop Output:

```
The sum of the first 10 natural numbers is: 55
```

Pract 5

Writing PL/SQL Blocks with basic programming constructs by including a GoTO to

jump out of a loop and NULL as a statement inside IF

```
plsql
DECLARE
   v_counter NUMBER := 1;      -- Counter for the loop
   v_exit_condition NUMBER := 5; -- Condition to exit the loop
```

~~~
BEGIN
   -- Loop to count from 1 to 10
   LOOP
      IF v_counter = v_exit_condition THEN
         GOTO exit_loop;  -- Jump out of the loop if the condition is met
      ELSE
         DBMS_OUTPUT.PUT_LINE('Current counter: ' || v_counter);
      END IF;

      v_counter := v_counter + 1;  -- Increment the counter
   END LOOP;

exit_loop:
   NULL;  -- This is a placeholder statement; it does nothing

   -- Output the final message
   DBMS_OUTPUT.PUT_LINE('Exited the loop at counter: ' || v_counter);
END;
/
```
~~~

Explanation

1. Variable Declarations:
   - `v_counter`: A counter initialized to 1.
   - `v_exit_condition`: The value at which the loop will exit (set to 5 in this case).

2. Loop:
   - The `LOOP` statement runs indefinitely until a `GOTO` statement is executed.
   - Inside the loop, we check if `v_counter` equals `v_exit_condition`.
   - If it does, we use `GOTO exit_loop` to jump to the `exit_loop` label, effectively exiting the loop.

3. Using `NULL:
   - The `NULL` statement is included after the `exit_loop` label. This is a placeholder that performs no action but is syntactically valid.

4. Output:
   - The loop prints the current counter value until it reaches the exit condition.
   - After exiting the loop, a final message is printed showing the counter value at the time of exit.

Output

Current counter: 1

Current counter: 2

Current counter: 3

Current counter: 4

Exited the loop at counter: 5

Pract 6

Writing Procedures in PL/SQL Block

a. Create an empty procedure, replace a procedure and call procedure

b. Create a stored procedure and call it

c. Define procedure to insert data

d. A forward declaration of procedure

a. Create an Empty Procedure, Replace a Procedure, and Call a Procedure

1. Create an Empty Procedure:**

sql

```
CREATE OR REPLACE PROCEDURE empty_procedure IS
BEGIN
   NULL; -- This does nothing
END empty_procedure;
/
```

****2. Replace a Procedure:****

sql

```
CREATE OR REPLACE PROCEDURE empty_procedure IS
BEGIN
   DBMS_OUTPUT.PUT_LINE('This is an updated procedure.');
END empty_procedure;
/
```

****3. Call the Procedure:****

```sql
BEGIN
   empty_procedure;
END;
/
```

b. Create a Stored Procedure and Call It

1. Create a Stored Procedure:

```sql
CREATE OR REPLACE PROCEDURE greet_user(p_username IN VARCHAR2) IS
BEGIN
   DBMS_OUTPUT.PUT_LINE('Hello, ' || p_username || '!');
END greet_user;
/
```

2. Call the Procedure:

```sql
BEGIN
   greet_user('Alice');
END;
/
```

c. Define a Procedure to Insert Data

1. Create a Procedure to Insert Data:
Assuming there's a table named `employees` with columns `id` and `name`.
sql

```
CREATE OR REPLACE PROCEDURE insert_employee(p_id IN NUMBER, p_name IN
VARCHAR2) IS
BEGIN
   INSERT INTO employees (id, name) VALUES (p_id, p_name);
   COMMIT; -- Don't forget to commit the transaction!
END insert_employee;
/
```

2. Call the Insert Procedure:**
sql

```
BEGIN
   insert_employee(1, 'John Doe');
END;
/
```

d. A Forward Declaration of Procedure

1. Forward Declaration:**
This allows you to declare a procedure before its definition.

sql

```
CREATE OR REPLACE PROCEDURE main_procedure;
CREATE OR REPLACE PROCEDURE helper_procedure IS
BEGIN
   DBMS_OUTPUT.PUT_LINE('Helper procedure called.');
END helper_procedure;
/
CREATE OR REPLACE PROCEDURE main_procedure IS
BEGIN
   helper_procedure; -- Call the helper procedure
   DBMS_OUTPUT.PUT_LINE('Main procedure called.');
END main_procedure;
/
```

2. Call the Main Procedure:
sql

```
BEGIN
   main_procedure;
END;
/
```

Pract 7

Writing Functions in PL/SQL Block.
a. Define and call a function
b. Define and use function in select clause,
c. Call function in dbms_output.put_line
d. recursive function
e count employee from a function and return value to a variable

f. call function and store the return value to a variable

.

a. Define and Call a Function

```
CREATE OR REPLACE FUNCTION add_numbers(
   p_num1 NUMBER,
   p_num2 NUMBER
) RETURN NUMBER IS
BEGIN
   RETURN p_num1 + p_num2;
END add_numbers;
/
```

For the addition function:

```
DECLARE
   v_result NUMBER;
BEGIN
   v_result := add_numbers(10, 20);
   DBMS_OUTPUT.PUT_LINE('The result is: ' || v_result);
END;
```
```

**Output**

```
The result is: 30
```

b. Define and Use Function in SELECT Clause

```
CREATE TABLE numbers (
   num1 NUMBER,
   num2 NUMBER
);

INSERT INTO numbers (num1, num2) VALUES (10, 20);
INSERT INTO numbers (num1, num2) VALUES (30, 40);
INSERT INTO numbers (num1, num2) VALUES (50, 60);
COMMIT;
```

sql

```
SELECT num1, num2, add_numbers(num1, num2) AS sum
FROM numbers;
```

Output

```
NUM1    NUM2    SUM
5       15      20
10      20      30
```

c. Call Function in DBMS_OUTPUT.PUT_LINE

sql

```
DECLARE
```
```

```
    v_num1 NUMBER := 30;
    v_num2 NUMBER := 50;
BEGIN
    DBMS_OUTPUT.PUT_LINE('The sum of ' || v_num1 || ' and ' || v_num2 || ' is: ' ||
add_numbers(v_num1, v_num2));
END;
```

Output

The sum of 30 and 50 is: 80

d. Recursive Function

For the factorial function:

sql

```
DECLARE
    v_factorial NUMBER;
BEGIN
    v_factorial := factorial(5);
    DBMS_OUTPUT.PUT_LINE('The factorial of 5 is: ' || v_factorial);
END;
```

Output

The factorial of 5 is: 120

e. Count Employees from a Function

```
DECLARE
    v_emp_count NUMBER;
BEGIN
    v_emp_count := count_employees();
    DBMS_OUTPUT.PUT_LINE('Total number of employees: ' || v_emp_count);
END;
```

Output(if there are, for example, 10 employees)

Total number of employees: 10

f. Call Function and Store the Return Value to a Variable

```sql
DECLARE
    v_result NUMBER;
BEGIN
    v_result := add_numbers(15, 25);
    DBMS_OUTPUT.PUT_LINE('The result of addition is: ' || v_result);
END;
```

Output

The result of addition is: 40

Pract 8

Creating and working with Insert/Update/Delete Trigger using Before/After clause.

1. Insert Triggers

BEFORE Insert Trigger

*Name**: `before_insert_employee`

*Purpose**: Validate that a new employee's salary is at least 30,000.

*Action**: Raises an error if the condition is not met.

```
CREATE TRIGGER before_insert_employee
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
    IF NEW.salary < 30000 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Salary must be at least 30,000';
    END IF;
END;
```

AFTER Insert Trigger

*Name**: `after_insert_employee`

*Purpose**: Log the insertion of a new employee.

```
CREATE TRIGGER after_insert_employee
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
    INSERT INTO employee_log (action, employee_id)
    VALUES ('INSERT', NEW.id);
END;
```

2. Update Triggers

- **BEFORE Update Trigger**
  - **Name**: `before_update_employee`
  - **Purpose**: Prevent the decrease of an employee's salary.
  - **Action**: Raises an error if the new salary is lower than the old salary.

```
CREATE TRIGGER before_update_employee
BEFORE UPDATE ON employees
FOR EACH ROW
BEGIN
    IF NEW.salary < OLD.salary THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Salary cannot be decreased';
    END IF;
END;
```

AFTER Update Trigger**

- **Name**: `after_update_employee`
- **Purpose**: Log the update of an employee's information.

```sql
CREATE TRIGGER after_update_employee
```

```
 AFTER UPDATE ON employees
 FOR EACH ROW
 BEGIN
    INSERT INTO employee_log (action, employee_id)
    VALUES ('UPDATE', NEW.id);
 END;
```

3. Delete Triggers

- **BEFORE Delete Trigger**
  - **Name**: `before_delete_employee`
  - **Purpose**: Prevent deletion of an employee if they are part of a department.
  - **Action**: Raises an error if the employee is found in the department table.

```
 CREATE TRIGGER before_delete_employee
 BEFORE DELETE ON employees
 FOR EACH ROW
 BEGIN
    IF EXISTS (SELECT 1 FROM department WHERE employee_id = OLD.id) THEN
       SIGNAL SQLSTATE '45000'
       SET MESSAGE_TEXT = 'Cannot delete employee; they are part of a department';
    END IF;
 END;
```

AFTER Delete Trigger**
  - **Name**: `after_delete_employee`
  - **Purpose**: Log the deletion of an employee.

```
 CREATE TRIGGER after_delete_employee
 AFTER DELETE ON employees
 FOR EACH ROW
 BEGIN
    INSERT INTO employee_log (action, employee_id)
    VALUES ('DELETE', OLD.id);
 END;
```

pract9

Write an Implicit and explicit cursor to complete the task.

Example of Implicit Cursor

```
DECLARE
   employee_name VARCHAR(100);
BEGIN
   FOR employee_rec IN (SELECT name FROM employees) LOOP
      employee_name := employee_rec.name;
      DBMS_OUTPUT.PUT_LINE('Employee Name: ' || employee_name);
   END LOOP;
END;
```

Explicit Cursor

Example of Explicit Cursor

```
DECLARE
   CURSOR employee_cursor IS
      SELECT name, salary FROM employees;
   employee_rec employee_cursor%ROWTYPE;  -- Variable to hold fetched record
BEGIN
   OPEN employee_cursor;  -- Open the cursor
   LOOP
      FETCH employee_cursor INTO employee_rec;  -- Fetch each record
      EXIT WHEN employee_cursor%NOTFOUND;  -- Exit if no more records

      DBMS_OUTPUT.PUT_LINE('Employee Name: ' || employee_rec.name ||
                           ', Salary: ' || employee_rec.salary);
   END LOOP;

   CLOSE employee_cursor;  -- Close the cursor
END;
```

Output

Sample Data in `employees` Table

Name	Salary
Alice	60000
Bob	50000
Charlie	70000

Output

```
Employee Name: Alice, Salary: 60000

Employee Name: Bob, Salary: 50000

Employee Name: Charlie, Salary: 70000
```

Explanation of the Output

1. **DBMS_OUTPUT.PUT_LINE**: This statement is used to display the concatenated string that includes each employee's name and salary.

2. **Loop Through Records**: The loop iterates through all records fetched from the `employee_cursor`, and each record's details are printed until there are no more records to fetch.If the `employees` table contains no records, the output will be empty. You can also customize the output or add more formatting based on your requirements

Pract 10

create packages and use it in SQL black to complete the task.

Step 1: Create a Package Specification and Body

Package Specification

```
CREATE OR REPLACE PACKAGE number_operations AS
   FUNCTION add_numbers(p_num1 NUMBER, p_num2 NUMBER) RETURN NUMBER;
   PROCEDURE display_result(v_result NUMBER);
END number_operations;
```

Package Body

```
CREATE OR REPLACE PACKAGE BODY number_operations AS
   FUNCTION add_numbers(p_num1 NUMBER, p_num2 NUMBER) RETURN NUMBER IS
   BEGIN
      RETURN p_num1 + p_num2;
   END add_numbers;
   PROCEDURE display_result(v_result NUMBER) IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE('The result is: ' || v_result);
   END display_result;
END number_operations;
/
```

Step 2: Use the Package in a PL/SQL Block

```
DECLARE
   v_result NUMBER;
BEGIN
   v_result := number_operations.add_numbers(10, 20);
   number_operations.display_result(v_result);
END;
/
```

Explanation

1. **Package Specification**:
   - **Function**: `add_numbers` takes two `NUMBER` parameters and returns their sum.
   - **Procedure**: `display_result` takes a `NUMBER` parameter and prints the result.

2. **Package Body**:
   - Implements the logic for both the function and the procedure.

3. **PL/SQL Block**:
   - Declares a variable `v_result`.
   - Calls `add_numbers` to compute the sum of 10 and 20.
   - Calls `display_result` to print the result.

Output

The result is: 30

Pract 11

Write a SQL block to handle exception by writing:

a. Predefined Exceptions,

b. User-Defined Exceptions,

c. Redeclared Predefined Exceptions,

```
DECLARE
   -- Predefined Exception
   no_data_found EXCEPTION;

   -- User-Defined Exception
   user_defined_error EXCEPTION;

   -- Redeclared Predefined Exception
   zero_division EXCEPTION;

   v_number1 NUMBER := 10;
   v_number2 NUMBER := 0; -- This will cause a division by zero
   v_result NUMBER;
BEGIN
   -- Simulate a situation that raises a predefined exception
   BEGIN
      SELECT salary INTO v_result FROM employees WHERE id = 999; -- Assuming this ID
doesn't exist
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         DBMS_OUTPUT.PUT_LINE('Predefined Exception: No data found for the specified ID.');
   END;

   -- Simulate a user-defined exception
   IF v_number2 = 0 THEN
      RAISE user_defined_error; -- Raising user-defined exception
   END IF;

   -- Handle the user-defined exception
   EXCEPTION
      WHEN user_defined_error THEN
         DBMS_OUTPUT.PUT_LINE('User-Defined Exception: Division by zero is not allowed.');

   -- Simulate a situation that raises a redeclared predefined exception
   BEGIN
      v_result := v_number1 / v_number2; -- This will cause a division by zero
   EXCEPTION
      WHEN zero_division THEN
         DBMS_OUTPUT.PUT_LINE('Redeclared Predefined Exception: Division by zero error.');
   END;

END;
/
```

Explanation

1. **Predefined Exception**:
   - `NO_DATA_FOUND` is a predefined exception raised when a query does not return any rows. In this block, it’s simulated by trying to select a salary for a non-existent employee ID.

2. **User-Defined Exception**:
   - `user_defined_error` is defined and raised when attempting to divide by zero. It checks if `v_number2` is zero and raises the exception accordingly.

3. **Redeclared Predefined Exception**:
   - `zero_division` is defined as an exception that redeclares the built-in division-by-zero error. The division by zero occurs when calculating `v_result`, which is caught in the inner block.

Output

Predefined Exception: No data found for the specified ID.
User-Defined Exception: Division by zero is not allowed.
Redeclared Predefined Exception: Division by zero error.

Pract 12

Create nested tables and work with nested tables

Step 1: Create a Nested Table Type

```
CREATE OR REPLACE TYPE employee_type AS OBJECT (
    id NUMBER,
    name VARCHAR2(100),
    salary NUMBER
);
CREATE OR REPLACE TYPE employee_table AS TABLE OF employee_type;
```

Step 2: Create a Table that Uses the Nested Table

```
CREATE TABLE department (
    dept_id NUMBER,
    dept_name VARCHAR2(100),
    employees employee_table -- Nested table column
) NESTED TABLE employees STORE AS employees_nt;  -- Physical storage for the nested
table
```

Step 3: Insert Data into the Nested Table

```
INSERT INTO department (dept_id, dept_name, employees) VALUES (
    1,
    'Sales',
    employee_table(employee_type(1, 'Alice', 60000), employee_type(2, 'Bob', 50000))
);

INSERT INTO department (dept_id, dept_name, employees) VALUES (
    2,
    'HR',
    employee_table(employee_type(3, 'Charlie', 70000), employee_type(4, 'Diana', 55000))
);
```

Step 4: Querying the Nested Table

```
SELECT d.dept_id, d.dept_name, e.id, e.name, e.salary
FROM department d, TABLE(d.employees) e;
```

Step 5: Updating Nested Table Entries

```
UPDATE department d
```

```
SET d.employees := employee_table(employee_type(1, 'Alice', 65000), employee_type(2, 'Bob',
50000))
WHERE d.dept_id = 1;
```

Step 6: Deleting from the Nested Table

```
DELETE FROM department d
WHERE EXISTS (
   SELECT 1
   FROM TABLE(d.employees) e
   WHERE e.id = 1 AND d.dept_id = 1
);
```

Complete Example

Step 1: Create Types

```
CREATE OR REPLACE TYPE employee_type AS OBJECT (
   id NUMBER,
   name VARCHAR2(100),
   salary NUMBER
);
/
CREATE OR REPLACE TYPE employee_table AS TABLE OF employee_type;
/
```

Step 2: Create Department Table

```
CREATE TABLE department (
   dept_id NUMBER,
   dept_name VARCHAR2(100),
   employees employee_table
) NESTED TABLE employees STORE AS employees_nt;
/
```

Step 3: Insert Data

```
INSERT INTO department (dept_id, dept_name, employees) VALUES (
   1,
   'Sales',
   employee_table(employee_type(1, 'Alice', 60000), employee_type(2, 'Bob', 50000))
);
INSERT INTO department (dept_id, dept_name, employees) VALUES (
   2,
   'HR',
   employee_table(employee_type(3, 'Charlie', 70000), employee_type(4, 'Diana', 55000))
);
```

Step 4: Query Data

```
SELECT d.dept_id, d.dept_name, e.id, e.name, e.salary
FROM department d, TABLE(d.employees) e;
```

Step 5: Update Data

```
UPDATE department d
SET d.employees := employee_table(employee_type(1, 'Alice', 65000), employee_type(2, 'Bob',
50000))
WHERE d.dept_id = 1;
```

Step 6: Delete Data

```
DELETE FROM department d
```

```
WHERE EXISTS (
   SELECT 1
   FROM TABLE(d.employees) e
   WHERE e.id = 1 AND d.dept_id = 1
);
```