Chapter 15

Mastering PL/SQL Collections

Mayko Freitas da Silva

Introduction

Imagine you're organizing a company picnic for the HR department. You need to keep track of all the employees attending, their food preferences, and the activities they're interested in. How would you manage this information efficiently in your PL/SQL program? This is where collections come to the rescue.

In the world of PL/SQL programming, you'll often find yourself working with groups of related data. Whether it's a list of employee names, a set of department codes, or a series of salary figures, you need a way to store and manipulate multiple values of the same type. This is exactly what PL/SQL collections are designed to do.

A collection in PL/SQL is like a digital container that can hold multiple items of the same type. Think of it as a smart, expandable box where each item has its own unique label. This powerful feature allows you to work with groups of data more efficiently and write cleaner, more organized code.

In this chapter, you'll embark on an exciting journey through the landscape of PL/SQL collections. You'll discover how to create, populate, and manipulate different types of collections, each with its own special characteristics and use cases.

Here's what you'll learn in this chapter:

1. The fundamentals of PL/SQL collections and why they're so useful
2. How to work with associative arrays (also known as index-by tables) and nested tables
3. The ins and outs of varrays (variable-size arrays) and when to use them
4. How to create and manage multidimensional collections for complex data structures
5. Advanced techniques using collection iteration controls and qualified expressions

By the end of this chapter, you'll have a solid understanding of PL/SQL collections and be able to use them effectively in your own programs. You'll see how collections can simplify your code, improve performance, and provide elegant solutions to common programming challenges.

So, are you ready to unlock the power of PL/SQL collections? Let's dive in and start exploring these versatile data structures.

Lab 15.1: Understanding PL/SQL Tables

After this lab, you will be able to:

- Create and use associative arrays
- Work with nested tables
- Apply collection methods to manipulate data

Introduction to PL/SQL Tables

Imagine you're tasked with creating a digital employee directory for the HR department. You need a way to store and quickly access employee information. This is where PL/SQL tables come in handy. PL/SQL tables are collection types that allow you to store multiple elements of the same data type, much like a single-column database table.

There are two main types of PL/SQL tables we'll explore: associative arrays (also known as index-by tables) and nested tables. Let's dive into each of these.

Associative Arrays

An associative array is like a customizable filing cabinet where you can choose your own labeling system. Each drawer (element) can be accessed using a unique label (index) that you define.

The general syntax for creating an associative array is:

```
TYPE type_name IS TABLE OF element_type [NOT NULL]
  INDEX BY index_type;
table_name TYPE_NAME;
```

Let's create an associative array to store employee last names:

```
DECLARE
  TYPE last_name_type IS TABLE OF employees.last_name%TYPE
    INDEX BY PLS_INTEGER;
  last_name_tab last_name_type;
BEGIN
  -- We'll populate this array later
  NULL;
END;
```

In this example, we've created an associative array type called last_name_type based on the last_name column of the employees table. We then declared a variable last_name_tab of this type.

Now, let's populate this array with some employee last names:

```
DECLARE
  TYPE last_name_type IS TABLE OF employees.last_name%TYPE
    INDEX BY PLS_INTEGER;
  last_name_tab last_name_type;
  v_index PLS_INTEGER := 0;
BEGIN
  FOR rec IN (SELECT last_name FROM employees WHERE ROWNUM <= 5)
  LOOP
    v_index := v_index + 1;
    last_name_tab(v_index) := rec.last_name;
    DBMS_OUTPUT.PUT_LINE('last_name(' || v_index || '): ' || last_name_tab(v_index));
  END LOOP;
END;
```

This script will output something like:

```
last_name(1): King
last_name(2): Kochhar
last_name(3): De Haan
last_name(4): Hunold
last_name(5): Ernst
```

Nested Tables

A nested table is similar to an associative array, but it doesn't require an index type and can be stored in a database column. Think of it as a box of numbered items where you can add or remove items as needed.

Here's how you declare a nested table:

```
TYPE type_name IS TABLE OF element_type [NOT NULL];
table_name TYPE_NAME;
```

Let's create a nested table to store department names:

```
DECLARE
  TYPE dept_name_type IS TABLE OF departments.department_name%TYPE;
  dept_name_tab dept_name_type := dept_name_type();
BEGIN
  -- We'll populate this table later
  NULL;
END;
```

Notice that we initialize the nested table using a constructor: dept_name_type(). This creates an empty, non-NULL nested table.

Now, let's populate and print the contents of this nested table:

```
DECLARE
  TYPE dept_name_type IS TABLE OF departments.department_name%TYPE;
  dept_name_tab dept_name_type := dept_name_type();
BEGIN
  FOR rec IN (SELECT department_name FROM departments WHERE ROWNUM <= 5)
  LOOP
    dept_name_tab.EXTEND;
    dept_name_tab(dept_name_tab.LAST) := rec.department_name;
  END LOOP;

  FOR i IN 1..dept_name_tab.COUNT
  LOOP
    DBMS_OUTPUT.PUT_LINE('dept_name(' || i || '): ' || dept_name_tab(i));
  END LOOP;
END;
```

This script will output something like:

```
dept_name(1): Administration
dept_name(2): Marketing
dept_name(3): Purchasing
dept_name(4): Human Resources
dept_name(5): Shipping
```

Collection Methods

PL/SQL provides several built-in methods to manipulate and retrieve information about collections. Let's explore some of the most commonly used methods:

1. EXISTS: Checks if an element exists at a specific index.
2. COUNT: Returns the number of elements in the collection.
3. FIRST and LAST: Return the first and last index of the collection.
4. PRIOR and NEXT: Return the previous and next index relative to a given index.
5. DELETE: Removes elements from the collection.
6. EXTEND: Increases the size of a nested table (not used with associative arrays).
7. TRIM: Removes elements from the end of a nested table.

Let's see these methods in action with an associative array of employee salaries:

```
DECLARE
  TYPE salary_type IS TABLE OF employees.salary%TYPE
    INDEX BY PLS_INTEGER;
  salary_tab salary_type;
BEGIN
  -- Populate the array
  FOR rec IN (SELECT employee_id, salary FROM employees WHERE ROWNUM <= 5)
  LOOP
    salary_tab(rec.employee_id) := rec.salary;
  END LOOP;

  -- EXISTS
  IF salary_tab.EXISTS(100) THEN
    DBMS_OUTPUT.PUT_LINE('Salary for employee 100 exists: ' || salary_tab(100));
  END IF;

  -- COUNT
  DBMS_OUTPUT.PUT_LINE('Number of salaries stored: ' || salary_tab.COUNT);

  -- FIRST and LAST
  DBMS_OUTPUT.PUT_LINE('First employee ID: ' || salary_tab.FIRST);
  DBMS_OUTPUT.PUT_LINE('Last employee ID: ' || salary_tab.LAST);

  -- PRIOR and NEXT
  DBMS_OUTPUT.PUT_LINE('Employee ID before 102: ' || salary_tab.PRIOR(102));
  DBMS_OUTPUT.PUT_LINE('Employee ID after 102: ' || salary_tab.NEXT(102));

  -- DELETE
  salary_tab.DELETE(103);
  DBMS_OUTPUT.PUT_LINE('Number of salaries after deletion: ' || salary_tab.COUNT);

  -- Try to access deleted element
  IF NOT salary_tab.EXISTS(103) THEN
    DBMS_OUTPUT.PUT_LINE('Salary for employee 103 no longer exists');
  END IF;
END;
```

Exercise 15.1

Create a PL/SQL block that uses a nested table to store the job titles of the first 10 employees (based on employee_id) from the employees table. Use collection methods to:

1. Print the number of job titles stored
2. Print the first and last job titles
3. Remove the last two job titles
4. Print the new count of job titles

Remember to initialize your nested table and use the EXTEND method when adding elements.

By mastering PL/SQL tables and collection methods, you've taken a significant step in efficiently managing groups of data in your PL/SQL programs. In the next lab, we'll explore another type of collection: varrays.

Lab 15.2: Exploring Varrays

After this lab, you will be able to:

- Create and initialize varrays
- Access and modify varray elements
- Compare varrays to other collection types

Introduction to Varrays

Imagine you're designing a system to track the top five performers in each department. You know you'll never need to track more than five, but you might sometimes track fewer. This scenario is perfect for a varray, or variable-size array.

A varray is like a expandable bookshelf with a maximum capacity. You decide the maximum number of shelves when you build it, but you don't have to use all the shelves right away. You can add books (elements) as needed, up to the maximum.

Creating and Initializing Varrays

The general syntax for creating a varray is:

```
TYPE type_name IS {VARRAY | VARYING ARRAY} (size_limit) OF element_type [NOT NULL];
varray_name TYPE_NAME;
```

Let's create a varray to store the top five performer names in a department:

```
DECLARE
  TYPE top_performers_type IS VARRAY(5) OF employees.last_name%TYPE;
  top_performers top_performers_type;
BEGIN
  -- We'll initialize this varray later
  NULL;
END;
```

In this example, we've created a varray type called top_performers_type that can hold up to 5 last names. We then declared a variable top_performers of this type.

Like nested tables, varrays must be initialized before use. Let's initialize and populate our varray:

```
DECLARE
  TYPE top_performers_type IS VARRAY(5) OF employees.last_name%TYPE;
  top_performers top_performers_type := top_performers_type();
BEGIN
  -- Populate the varray with some sample data
  top_performers.EXTEND(3);
  top_performers(1) := 'King';
  top_performers(2) := 'Kochhar';
  top_performers(3) := 'De Haan';

  -- Print the contents of the varray
  FOR i IN 1..top_performers.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('Top performer ' || i || ': ' || top_performers(i));
  END LOOP;
END;
```

This script will output:

```
Top performer 1: King
Top performer 2: Kochhar
Top performer 3: De Haan
```

Accessing and Modifying Varray Elements

Accessing and modifying varray elements is similar to working with other collection types. You use the index to access or modify individual elements.

Let's update our top performers list and add two more performers:

```
DECLARE
  TYPE top_performers_type IS VARRAY(5) OF employees.last_name%TYPE;
  top_performers top_performers_type := top_performers_type('King', 'Kochhar', 'De Haan');
BEGIN
  -- Modify an existing element
  top_performers(2) := 'Hunold';

  -- Add new elements
  top_performers.EXTEND(2);
  top_performers(4) := 'Ernst';
  top_performers(5) := 'Austin';

  -- Print the updated varray
  FOR i IN 1..top_performers.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('Top performer ' || i || ': ' || top_performers(i));
  END LOOP;

  -- Try to add one more element (this will raise an exception)
  BEGIN
    top_performers.EXTEND;
    top_performers(6) := 'Pataballa';
  EXCEPTION
    WHEN SUBSCRIPT_BEYOND_COUNT THEN
      DBMS_OUTPUT.PUT_LINE('Cannot add more elements. Varray is full.');
  END;
END;
```

This script demonstrates how to modify existing elements, add new elements, and what happens when you try to exceed the varray's size limit.

Comparing Varrays to Other Collection Types

Let's compare varrays with associative arrays and nested tables:

1. Size Limit:
    - Varrays have a maximum size defined at declaration.
    - Associative arrays and nested tables can grow dynamically without a predefined limit.
2. Index Type:
    - Varrays are always indexed by integers starting from 1.
    - Associative arrays can be indexed by PLS_INTEGER or string types.
    - Nested tables are indexed by integers, but the index may not always start at 1 if elements are deleted.
3. Storage:
    - Varrays and nested tables can be stored in database tables.
    - Associative arrays exist only in PL/SQL memory.
4. Sparse or Dense:
    - Varrays are always dense (no gaps in index values).
    - Associative arrays and nested tables can be sparse (may have gaps in index values).

Here's an example that illustrates some of these differences:

```
DECLARE
  -- Our varray
  TYPE dept_names_varray IS VARRAY(5) OF departments.department_name%TYPE;
  v_dept_names dept_names_varray := dept_names_varray();

  -- A nested table
  TYPE dept_names_nested IS TABLE OF departments.department_name%TYPE;
  n_dept_names dept_names_nested := dept_names_nested();

  -- An associative array
  TYPE dept_names_assoc IS TABLE OF departments.department_name%TYPE
    INDEX BY PLS_INTEGER;
  a_dept_names dept_names_assoc;

BEGIN
  -- Let's fill 'em up (but only with 4 for the varray)
  FOR rec IN (SELECT department_name FROM departments WHERE ROWNUM <= 4) LOOP
    v_dept_names.EXTEND;
    v_dept_names(v_dept_names.LAST) := rec.department_name;
    n_dept_names.EXTEND;
    n_dept_names(n_dept_names.LAST) := rec.department_name;
    a_dept_names(a_dept_names.COUNT + 1) := rec.department_name;
  END LOOP;

  -- Try to add up to the VARRAY limit
  IF v_dept_names.COUNT <= v_dept_names.LIMIT THEN
    v_dept_names.EXTEND;
    v_dept_names(v_dept_names.LAST) := 'New Department 1';
  END IF;

  IF v_dept_names.COUNT < v_dept_names.LIMIT THEN
    v_dept_names.EXTEND;
    v_dept_names(v_dept_names.LAST) := 'New Department 2';
  END IF;

  -- Extend others without issue
  n_dept_names.EXTEND(2);
  n_dept_names(n_dept_names.LAST - 1) := 'New Department 1';
  n_dept_names(n_dept_names.LAST) := 'New Department 2';
  a_dept_names(a_dept_names.COUNT + 1) := 'New Department 1';
  a_dept_names(a_dept_names.COUNT + 1) := 'New Department 2';

  -- Let's delete the middle ones and see what happens
  n_dept_names.DELETE(3);
  a_dept_names.
```

```
DELETE(3);

  -- Time to check our results
  DBMS_OUTPUT.PUT_LINE('Varray has: ' || v_dept_names.COUNT);
  DBMS_OUTPUT.PUT_LINE('Nested table has: ' || n_dept_names.COUNT);
  DBMS_OUTPUT.PUT_LINE('Associative array has: ' || a_dept_names.COUNT);
  DBMS_OUTPUT.PUT_LINE('Nested table still has index 3? ' || CASE WHEN n_dept_names.EXISTS(3) THEN
'Yep' ELSE 'Nope' END);
  DBMS_OUTPUT.PUT_LINE('Associative array still has index 3? ' || CASE WHEN a_dept_names.EXISTS(3) TH
EN 'Yep' ELSE 'Nope' END);
END;
```

This script demonstrates the size limitations of varrays, the ability to add elements beyond the initial size for nested tables and associative arrays, and the sparse nature of nested tables and associative arrays after element deletion.

Exercise 15.2

Create a PL/SQL block that uses a varray to store the first names of the top 3 highest-paid employees in the employees table. Your block should:

1. Declare and initialize the varray
2. Populate the varray with the first names of the top 3 highest-paid employees
3. Print the contents of the varray
4. Try to add a 4th employee and handle the exception
5. Update the second employee's name to uppercase and print the updated varray

Remember to use the appropriate varray methods and handle any exceptions that may occur.

By mastering varrays, you've added another powerful tool to your PL/SQL collection toolkit. You now understand when to use varrays and how they differ from other collection types. In the next lab, we'll explore how to work with multidimensional collections for more complex data structures.

Lab 15.3: Working with Multidimensional Collections

After this lab, you will be able to:

- Create multidimensional collections
- Access and manipulate elements in nested collections
- Understand the practical applications of multidimensional collections

Introduction to Multidimensional Collections

Imagine you're designing a system to track employee skills across different departments. Each department has multiple employees, and each employee has multiple skills. This complex data structure is a perfect use case for multidimensional collections.

A multidimensional collection in PL/SQL is like a set of nesting dolls, each containing smaller sets of items. It allows you to create collections of collections, providing a way to represent complex, hierarchical data structures.

Creating Multidimensional Collections

Let's start by creating a two-dimensional collection to represent departments and their employees:

```
DECLARE
  TYPE employee_list IS TABLE OF employees.employee_id%TYPE;
  TYPE dept_emp_type IS TABLE OF employee_list INDEX BY PLS_INTEGER;
  dept_employees dept_emp_type;
BEGIN
  -- We'll populate this collection later
  NULL;
END;
```

In this example, we've created a nested table type employee_list to hold employee IDs, and then used this type to create an associative array dept_emp_type indexed by department ID.

Now, let's populate this multidimensional collection with some data:

```
DECLARE
  TYPE employee_list IS TABLE OF employees.employee_id%TYPE;
  TYPE dept_emp_type IS TABLE OF employee_list INDEX BY PLS_INTEGER;
  dept_employees dept_emp_type;
BEGIN
  -- Populate the collection
  FOR dept IN (SELECT department_id FROM departments WHERE department_id IN (10, 20, 30)) LOOP
    dept_employees(dept.department_id) := employee_list();
    FOR emp IN (SELECT employee_id FROM employees WHERE department_id = dept.department_id) LOOP
      dept_employees(dept.department_id).EXTEND;
      dept_employees(dept.department_id)(dept_employees(dept.department_id).LAST) := emp.employee_id;
    END LOOP;
  END LOOP;

  -- Print the contents
  FOR dept IN dept_employees.FIRST..dept_employees.LAST LOOP
    IF dept_employees.EXISTS(dept) THEN
      DBMS_OUTPUT.PUT_LINE('Department ' || dept || ' has ' || dept_employees(dept).COUNT || ' employees:');
      FOR i IN 1..dept_employees(dept).COUNT LOOP
        DBMS_OUTPUT.PUT_LINE('  Employee ID: ' || dept_employees(dept)(i));
      END LOOP;
    END IF;
  END LOOP;
END;
```

This script creates a two-dimensional collection where the outer collection is indexed by department ID, and each inner collection contains the employee IDs for that department.

Accessing and Manipulating Elements

Accessing elements in a multidimensional collection requires multiple index values. Let's modify our previous example to update and access specific elements:

```
DECLARE
  TYPE employee_list IS TABLE OF employees.employee_id%TYPE;
  TYPE dept_emp_type IS TABLE OF employee_list INDEX BY PLS_INTEGER;
  dept_employees dept_emp_type;
BEGIN
  -- Populate the collection (same as before)
  FOR dept IN (SELECT department_id FROM departments WHERE department_id IN (10, 20, 30)) LOOP
    dept_employees(dept.department_id) := employee_list();
    FOR emp IN (SELECT employee_id FROM employees WHERE department_id = dept.department_id) LOOP
      dept_employees(dept.department_id).EXTEND;
      dept_employees(dept.department_id)(dept_employees(dept.department_id).LAST) := emp.employee_id;
    END LOOP;
  END LOOP;

  -- Print initial state of department 20
  DBMS_OUTPUT.PUT_LINE('Initial state of Department 20:');
  FOR i IN 1..dept_employees(20).COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('  Employee ID: ' || dept_employees(20)(i));
  END LOOP;

  -- Add a new employee to department 20
  dept_employees(20).EXTEND;
  dept_employees(20)(dept_employees(20).LAST) := 999; -- Assuming 999 is a valid employee ID

  -- Remove the first employee from department 20
  dept_employees(20).DELETE(1);

  -- Print updated state of department 20
  DBMS_OUTPUT.PUT_LINE('Updated state of Department 20:');
  FOR i IN 1..dept_employees(20).COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('  Employee ID: ' || dept_employees(20)(i));
  END LOOP;

  -- Try to access a non-existent department
  IF NOT dept_employees.EXISTS(40) THEN
    DBMS_OUTPUT.PUT_LINE('Department 40 does not exist in the collection.');
  END IF;
END;
```

This script demonstrates how to add and remove elements from the inner collection, as well as how to handle attempts to access non-existent elements.

Practical Applications of Multidimensional Collections

Multidimensional collections are particularly useful in scenarios where you need to represent complex, hierarchical data structures. Here are a few practical applications:

1. Organizational Hierarchies: Representing company structures with departments, teams, and employees.
2. Skill Matrices: Tracking skills for employees across different departments or projects.
3. Sales Data: Organizing sales figures by region, product category, and individual products.
4. Educational Systems: Managing courses, students, and grades across different academic years.

Let's implement a simple skill matrix using a multidimensional collection:

```
DECLARE
  TYPE skill_level_type IS RECORD (
    skill_name VARCHAR2(50),
    skill_level NUMBER(1)
  );
  TYPE skill_list IS TABLE OF skill_level_type;
  TYPE emp_skills_type IS TABLE OF skill_list INDEX BY PLS_INTEGER;
  emp_skills emp_skills_type;
BEGIN
  -- Add skills for employees
  emp_skills(100) := skill_list();
  emp_skills(100).EXTEND(2);
  emp_skills(100)(1).skill_name := 'Leadership';
  emp_skills(100)(1).skill_level := 5;
  emp_skills(100)(2).skill_name := 'Communication';
  emp_skills(100)(2).skill_level := 4;

  emp_skills(101) := skill_list();
  emp_skills(101).EXTEND(2);
  emp_skills(101)(1).skill_name := 'Java Programming';
  emp_skills(101)(1).skill_level := 4;
  emp_skills(101)(2).skill_name := 'Database Design';
  emp_skills(101)(2).skill_level := 3;

  -- Print skills for employees
  FOR emp_id IN emp_skills.FIRST..emp_skills.LAST LOOP
    IF emp_skills.EXISTS(emp_id) THEN
      DBMS_OUTPUT.PUT_LINE('Skills for Employee ' || emp_id || ':');
      FOR i IN 1..emp_skills(emp_id).COUNT LOOP
        DBMS_OUTPUT.PUT_LINE('  ' || emp_skills(emp_id)(i).skill_name || ': ' || emp_skills(emp_id)(i).skill_leve
l);
      END LOOP;
    END IF;
  END LOOP;
END;
```

This example creates a multidimensional collection to represent a skill matrix for employees. It demonstrates how to add skills to the collection and how to print the skills for each employee.

Exercise 15.3

Create a PL/SQL block that uses a multidimensional collection to represent a project assignment system. Your collection should store project IDs as the outer index, and for each project, store a list of assigned employee IDs along with their role on the project (e.g., 'Manager', 'Developer', 'Tester'). Your block should:

1. Declare and initialize the multidimensional collection
2. Assign at least three employees to two different projects with specific roles
3. Remove an employee from one project
4. Print all employees and their roles for both projects

Use employees and their IDs from the employees table in the HR schema for this exercise. You can invent project IDs as needed.

By mastering multidimensional collections, you've gained the ability to represent complex data structures in PL/SQL. This powerful feature allows you to organize and manipulate hierarchical data efficiently, opening up new possibilities for data modeling in your PL/SQL programs. In the next lab, we'll explore advanced techniques for working with collections using iteration controls and qualified expressions.

Lab 15.4: Collection Iteration Controls and Qualified Expressions

Introduction

In this lab, you will learn about advanced ways to iterate through collections and use qualified expressions to populate them efficiently. PL/SQL provides powerful iteration controls that make working with collections more flexible and intuitive.

Learning Objectives

After completing this lab, you will be able to:

- Use VALUES OF, INDICES OF, and PAIRS OF iteration controls
- Implement qualified expressions for collection population
- Apply aggregate qualified expressions effectively
- Work with both named and positional associations

Collection Iteration Controls

PL/SQL offers three specialized iteration controls for collections:

- VALUES OF: Iterates through collection elements
- INDICES OF: Iterates through collection indexes
- PAIRS OF: Iterates through both indexes and elements simultaneously

Let's explore each type with practical examples:

VALUES OF Example

```
DECLARE
  TYPE salary_list_t IS TABLE OF employees.salary%TYPE;
  v_salaries salary_list_t := salary_list_t();
BEGIN
  -- Populate collection with some employee salaries
  SELECT salary BULK COLLECT INTO v_salaries
  FROM employees
  WHERE department_id = 80;

  -- Print all salaries using VALUES OF
  FOR salary IN VALUES OF v_salaries LOOP
    DBMS_OUTPUT.PUT_LINE('Salary: ' || salary);
  END LOOP;
END;
/
```

INDICES OF Example

```
DECLARE
  TYPE emp_salary_t IS TABLE OF NUMBER
    INDEX BY PLS_INTEGER;
  v_emp_salaries emp_salary_t;
BEGIN
  -- Populate sparse collection
  v_emp_salaries(100) := 24000;
  v_emp_salaries(200) := 17000;
  v_emp_salaries(300) := 9000;

  -- Print indices using INDICES OF
  FOR i IN INDICES OF v_emp_salaries LOOP
    DBMS_OUTPUT.PUT_LINE('Index: ' || i);
  END LOOP;
END;
/
```

PAIRS OF Example

```
DECLARE
  TYPE emp_name_t IS TABLE OF employees.last_name%TYPE
    INDEX BY PLS_INTEGER;
  v_emp_names emp_name_t;
BEGIN
  -- Populate with some employee names
  SELECT last_name BULK COLLECT INTO v_emp_names
  FROM employees
  WHERE ROWNUM <= 5;

  -- Print both index and value using PAIRS OF
  FOR i, name IN PAIRS OF v_emp_names LOOP
    DBMS_OUTPUT.PUT_LINE('Employee ' || i || ': ' || name);
  END LOOP;
END;
/
```

Qualified Expressions

Starting from Oracle 18c, qualified expressions provide a concise way to populate collections. Let's explore both named and positional associations:

Named Association

```
DECLARE
  TYPE dept_salary_t IS TABLE OF NUMBER;
  -- Using qualified expression with named association
  v_dept_salaries dept_salary_t :=
    dept_salary_t(1 => 8000, 2 => 9000, 3 => 10000);
BEGIN
  FOR i, salary IN PAIRS OF v_dept_salaries LOOP
    DBMS_OUTPUT.PUT_LINE('Position ' || i || ': ' || salary);
  END LOOP;
END;
/
```

Positional Association

```
DECLARE
  TYPE emp_id_t IS TABLE OF NUMBER;
  -- Using qualified expression with positional association
  v_emp_ids emp_id_t := emp_id_t(100, 101, 102, 103);
BEGIN
  FOR id IN VALUES OF v_emp_ids LOOP
    DBMS_OUTPUT.PUT_LINE('Employee ID: ' || id);
  END LOOP;
END;
/
```

Aggregate Qualified Expressions

You can use aggregate functions with qualified expressions to create more complex collections:

Index Iterator Example

```
DECLARE
  TYPE tab_type IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
  v_tab     tab_type;
  v_tab_new tab_type;
BEGIN
  v_tab := tab_type(FOR i IN 1..4 => i/2);

  DBMS_OUTPUT.PUT_LINE('v_tab:');
  FOR i, v IN PAIRS OF v_tab
  LOOP
    DBMS_OUTPUT.PUT_LINE('index = '||i||', value = '||v);
  END LOOP;

  v_tab_new :=
    tab_type(FOR i, v IN PAIRS OF v_tab INDEX i+10 => v+10);

  DBMS_OUTPUT.PUT_LINE('v_tab_new:');
  FOR i, v IN PAIRS OF v_tab_new
  LOOP
    DBMS_OUTPUT.PUT_LINE('index = '||i||', value = '||v);
  END LOOP;
END;
/
```

Sequence Iterator Example

```
DECLARE
  TYPE tab_type IS TABLE OF NUMBER;
  v_tab tab_type;
BEGIN
  -- Initialize with iterator
  v_tab := tab_type(FOR i IN 1..4 INDEX i => i/2);

  DBMS_OUTPUT.PUT_LINE('v_tab:');
  FOR i, v IN PAIRS OF v_tab
  LOOP
    DBMS_OUTPUT.PUT_LINE('index = '||i||', value = '||v);
  END LOOP;

  -- Update v_tab collection using sequence
  v_tab := tab_type(FOR i IN 10..14 SEQUENCE => i/2);

  DBMS_OUTPUT.PUT_LINE('v_tab updated:');
  FOR i, v IN PAIRS OF v_tab
  LOOP
    DBMS_OUTPUT.PUT_LINE('index = '||i||', value = '||v);
  END LOOP;
END;
/
```

Mastering PL/SQL Collections