Chapter 16

Advanced Data Structures in PL/SQL

Mayko Freitas da Silva

Introduction

Imagine you're a database developer tasked with creating a system to manage complex, interconnected data. You need to efficiently store and manipulate various pieces of information that are logically related but have different data types. How would you organize this data in a way that's both intuitive and efficient? This is where advanced data structures in PL/SQL come to your rescue.

In this chapter, you will embark on a journey into the world of complex data organization in PL/SQL. You'll discover powerful techniques that allow you to work with related information in a structured, efficient, and intuitive manner. These skills are crucial for any PL/SQL developer looking to build robust, maintainable, and high-performance database applications across various domains.

In this chapter, you will learn about:

1. User-Defined Records: Custom-designed containers for holding related data of different types in a single unit.
2. Nested Records: Records within records, allowing for more complex data relationships and hierarchical data structures.
3. Collections of Records: Organizing multiple records in a single structure, enabling efficient handling of sets of complex data.

By mastering these concepts, you'll be able to create more sophisticated and efficient PL/SQL programs. You'll learn how to organize data in a way that mirrors real-world relationships, making your code more intuitive to write and easier to maintain.

As we delve into each topic, we'll use practical examples based on the HR Demo Schema objects. However, the concepts and techniques you'll learn are universally applicable across various database scenarios and industries. This approach will help you see how these advanced data structures can be applied to real-world database challenges you might face in any project.

Let's begin our exploration of advanced PL/SQL data structures. By the end of this chapter, you'll have a toolkit of techniques to handle complex data relationships with ease and confidence, enabling you to develop more efficient and powerful database solutions for any application domain.

Lab 16.1: Building Your Own Record Types

After this lab, you will be able to:

- Create custom record types to suit your specific data needs
- Initialize and manipulate fields within user-defined records
- Apply constraints to ensure data integrity in your records

Understanding User-Defined Records

Think of a user-defined record as a custom form you create to collect specific information. Just as you might design a form to gather details about employees, you can create a record type in PL/SQL to store related data in a structured way.

Let's start by looking at the general syntax for creating a user-defined record:

```
TYPE type_name IS RECORD
(
  field_name1 datatype1 [NOT NULL] [:= default_expression1],
  field_name2 datatype2 [NOT NULL] [:= default_expression2],
  ...
  field_nameN datatypeN [NOT NULL] [:= default_expressionN]
);

record_name type_name;
```

This structure allows you to define a record with multiple fields, each with its own data type and optional constraints or default values.

Creating Your First Custom Record

Let's create a custom record to store information about employees in the HR schema. We'll call it emp_info_type:

```
DECLARE
  TYPE emp_info_type IS RECORD
  (
    emp_id    employees.employee_id%TYPE,
    full_name VARCHAR2(100),
    job_title jobs.job_title%TYPE,
    hire_date DATE := SYSDATE
  );

  emp_record emp_info_type;
BEGIN
  -- We'll populate this record later
  NULL;
END;
/
```

In this example, you've created a record type emp_info_type with four fields:

- emp_id: Uses the same data type as the employee_id column in the employees table
- full_name: A VARCHAR2 field to store the employee's full name
- job_title: Matches the data type of the job_title column in the jobs table
- hire_date: A DATE field with a default value of the current date

Initializing and Manipulating Record Fields

Now that you've created your custom record type, let's see how to work with it:

```
DECLARE
  TYPE emp_info_type IS RECORD
  (
    emp_id    employees.employee_id%TYPE,
    full_name VARCHAR2(100),
    job_title jobs.job_title%TYPE,
    hire_date DATE := SYSDATE
  );

  emp_record emp_info_type;
BEGIN
  -- Populate the record
  SELECT e.employee_id, e.first_name || ' ' || e.last_name, j.job_title, e.hire_date
  INTO emp_record.emp_id, emp_record.full_name, emp_record.job_title, emp_record.hire_date
  FROM employees e
  JOIN jobs j ON e.job_id = j.job_id
  WHERE e.employee_id = 103;

  -- Display the record information
  DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_record.emp_id);
  DBMS_OUTPUT.PUT_LINE('Name: ' || emp_record.full_name);
  DBMS_OUTPUT.PUT_LINE('Job Title: ' || emp_record.job_title);
  DBMS_OUTPUT.PUT_LINE('Hire Date: ' || TO_CHAR(emp_record.hire_date, 'DD-MON-YYYY'));
END;
/
```

This script demonstrates how to populate your custom record with data from the HR schema tables and then display the information.

Applying Constraints to Record Fields

To ensure data integrity, you can apply constraints to your record fields. Let's modify our emp_info_type to include a NOT NULL constraint:

```
DECLARE
  TYPE emp_info_type IS RECORD
  (
    emp_id    employees.employee_id%TYPE NOT NULL := 0,  -- Default value
    full_name VARCHAR2(100) NOT NULL := 'Unknown',       -- Default value
    job_title jobs.job_title%TYPE,
    hire_date DATE := SYSDATE
  );

  emp_record emp_info_type;
BEGIN
  -- Now we can assign values
  emp_record.emp_id := 104;
  emp_record.full_name := 'John Doe';
  emp_record.job_title := 'Manager';

  DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_record.emp_id);
  DBMS_OUTPUT.PUT_LINE('Name: ' || emp_record.full_name);
  DBMS_OUTPUT.PUT_LINE('Job Title: ' || emp_record.job_title);
  DBMS_OUTPUT.PUT_LINE('Hire Date: ' || TO_CHAR(emp_record.hire_date, 'DD-MON-YYYY'));
END;
/
```

In this example, we've added NOT NULL constraints to the emp_id and full_name fields. This ensures that these fields must always have a value, improving data integrity in your custom record.

There's a crucial point we need to understand about NOT NULL constraints in record types. When we declare a record type with NOT NULL fields, these fields must be initialized at the point of declaration. This is different from regular variables, where we can initialize NOT NULL fields after declaration.

Exercise 16.1

Create a custom record type called dept_summary_type that includes the following fields:

- dept_id: Same type as department_id in the departments table
- dept_name: Same type as department_name in the departments table
- manager_name: VARCHAR2(100)
- employee_count: NUMBER
- avg_salary: NUMBER(10,2)

Use this record type to create a PL/SQL block that retrieves and displays summary information for a specific department (e.g., department_id = 60).

By completing this lab, you've taken your first steps into the world of custom data structures in PL/SQL. You've learned how to create, initialize, and manipulate user-defined records, as well as how to apply constraints to ensure data integrity. These skills form the foundation for building more complex and efficient database applications.

Qualified Expressions with Records

Qualified expressions provide a concise and elegant way to assign values to all fields of a record at once. This technique can make your code more readable and reduce the number of lines needed to populate a record. Here's how you can use qualified expressions with records:

```
DECLARE
  TYPE employee_rec_type IS RECORD (
    employee_id NUMBER,
    full_name   VARCHAR2(100),
    job_title   VARCHAR2(35),
    hire_date   DATE
  );

  emp_info employee_rec_type;
BEGIN
  -- Using a qualified expression to assign values
  emp_info := employee_rec_type(
    employee_id => 101,
    full_name   => 'John Smith',
    job_title   => 'Software Engineer',
    hire_date   => DATE '2020-03-15'
  );

  -- Display the information
  DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_info.employee_id);
  DBMS_OUTPUT.PUT_LINE('Name: ' || emp_info.full_name);
  DBMS_OUTPUT.PUT_LINE('Job Title: ' || emp_info.job_title);
  DBMS_OUTPUT.PUT_LINE('Hire Date: ' || TO_CHAR(emp_info.hire_date, 'YYYY-MM-DD'));
END;
```

In this example, we use a qualified expression to assign values to all fields of the emp_info record in a single operation. This approach is particularly useful when you have records with many fields or when you want to make your code more self-documenting.

You can also use qualified expressions with records in SQL statements:

```
DECLARE
  -- Our record type definition
  TYPE employee_rec_type IS RECORD (
    employee_id NUMBER,
    full_name   VARCHAR2(100),
    job_title   VARCHAR2(35),
    hire_date   DATE
  );

  emp_info employee_rec_type;
BEGIN
  -- Here's the correct way to do it
  SELECT e.employee_id,
       e.first_name || ' ' || e.last_name,
       j.job_title,
       e.hire_date
  INTO   emp_info.employee_id,
       emp_info.full_name,
       emp_info.job_title,
       emp_info.hire_date
  FROM   employees e
  JOIN   jobs j ON e.job_id = j.job_id
  WHERE  e.employee_id = 103;

  -- Display the information
  DBMS_OUTPUT.PUT_LINE('Employee ID: ' || emp_info.employee_id);
  DBMS_OUTPUT.PUT_LINE('Name: ' || emp_info.full_name);
  DBMS_OUTPUT.PUT_LINE('Job Title: ' || emp_info.job_title);
  DBMS_OUTPUT.PUT_LINE('Hire Date: ' || TO_CHAR(emp_info.hire_date, 'YYYY-MM-DD'));
END;
```

This example demonstrates how to use a qualified expression within a SELECT statement to populate a record directly from a query result.

Understanding Record Compatibility

Record compatibility is an important concept when working with user-defined records in PL/SQL. Two records are considered compatible if they have the same structure, meaning they have the same number of fields, in the same order, with compatible data types. However, it's crucial to note that even if two records have identical structures, they may not be directly assignable if they are declared using different record types.

Let's explore this concept with some examples:

```
DECLARE
  -- Define two record types with the same structure
  TYPE employee_type1 IS RECORD (
    id   NUMBER,
    name VARCHAR2(100)
  );

  TYPE employee_type2 IS RECORD (
    id   NUMBER,
    name VARCHAR2(100)
  );

  -- Declare variables of each type
  emp1 employee_type1;
  emp2 employee_type2;

  -- Declare a third variable using %ROWTYPE
  emp3 employees%ROWTYPE;
BEGIN
  -- Assign values to emp1
  emp1.id := 101;
  emp1.name := 'Alice Johnson';

  -- This will cause a compilation error:
  -- emp2 := emp1;

  -- Instead, we need to assign values individually:
  emp2.id := emp1.id;
  emp2.name := emp1.name;

  -- We can assign values from a %ROWTYPE record to our user-defined record
  SELECT * INTO emp3 FROM employees WHERE employee_id = 103;

  emp1.id := emp3.employee_id;
  emp1.name := emp3.first_name || ' ' || emp3.last_name;

  -- Display the information
  DBMS_OUTPUT.PUT_LINE('emp1 - ID: ' || emp1.id || ', Name: ' || emp1.name);
  DBMS_OUTPUT.PUT_LINE('emp2 - ID: ' || emp2.id || ', Name: ' || emp2.name);
END;
```

In this example:

1. **We define two record types (**employee_type1 and employee_type2) with identical structures.

2. Despite having the same structure, we cannot directly assign emp1 to emp2 because they are of different types.
3. To copy data between these records, we need to assign values field by field.
4. We can, however, assign values from a %ROWTYPE record (like emp3) to our user-defined record, as long as the field names and types are compatible.

Understanding record compatibility is crucial when working with complex data structures in PL/SQL. It helps you avoid common pitfalls and write more robust code, especially when dealing with different record types or when interfacing between user-defined records and table-based or cursor-based records.

By mastering qualified expressions and understanding record compatibility, you can write more efficient, readable, and maintainable PL/SQL code when working with complex data structures.

Lab 16.2: Records Within Records

After this lab, you will be able to:

- Use Nested Records
- Create complex data structures with nested records
- Access data in nested records
- Combine collections and records in nested structures

Nested Records

Nested records are a powerful feature in PL/SQL that allow you to create more complex and hierarchical data structures. A nested record is simply a record that contains another record as one of its fields. This concept is particularly useful when you need to represent data that has a natural hierarchy or when you want to group related data at different levels.

Let's start with a basic example of a nested record:

```
DECLARE
  -- Define a record type for name
  TYPE name_rec_type IS RECORD (
    first_name  VARCHAR2(50),
    last_name   VARCHAR2(50)
  );

  -- Define a record type that includes the name record
  TYPE person_rec_type IS RECORD (
    person_id   NUMBER,
    name        name_rec_type,
    birth_date  DATE
  );

  -- Declare a variable of the nested record type
  person_info person_rec_type;
BEGIN
  -- Assign values to the nested record
  person_info.person_id := 1;
  person_info.name.first_name := 'John';
  person_info.name.last_name := 'Doe';
  person_info.birth_date := DATE '1990-01-15';

  -- Display the information
  DBMS_OUTPUT.PUT_LINE('Person ID: ' || person_info.person_id);
  DBMS_OUTPUT.PUT_LINE('Name: ' || person_info.name.first_name || ' ' || person_info.name.last_name);
  DBMS_OUTPUT.PUT_LINE('Birth Date: ' || TO_CHAR(person_info.birth_date, 'YYYY-MM-DD'));
END;
```

In this example, name_rec_type is a record type that contains first and last names. The person_rec_type is a nested record that includes the name_rec_type as one of its fields, along with other information about the person.

Accessing Data in Nested Records

To access fields in a nested record, you use dot notation, extending it to as many levels as necessary. In the example above, we accessed the first name using person_info.name.first_name.

Creating Complex Data Structures with Nested Records

Nested records allow you to create more complex data structures that can represent real-world relationships. Let's expand our example to include more nested levels:

```
DECLARE
   TYPE address_rec_type IS RECORD (
      street  VARCHAR2(100),
      city    VARCHAR2(50),
      state   VARCHAR2(25),
      zip     VARCHAR2(10)
   );

   TYPE name_rec_type IS RECORD (
      first_name  VARCHAR2(50),
      last_name   VARCHAR2(50)
   );

   TYPE person_rec_type IS RECORD (
      person_id   NUMBER,
      name        name_rec_type,
      birth_date  DATE,
      address     address_rec_type
   );

   person_info person_rec_type;
BEGIN
   -- Assign values to the nested record
   person_info.person_id := 1;
   person_info.name.first_name := 'John';
   person_info.name.last_name := 'Doe';
   person_info.birth_date := DATE '1990-01-15';
   person_info.address.street := '123 Main St';
   person_info.address.city := 'Anytown';
   person_info.address.state := 'CA';
   person_info.address.zip := '12345';

   -- Display the information
   DBMS_OUTPUT.PUT_LINE('Person ID: ' || person_info.person_id);
   DBMS_OUTPUT.PUT_LINE('Name: ' || person_info.name.first_name || ' ' || person_info.name.last_name);
   DBMS_OUTPUT.PUT_LINE('Birth Date: ' || TO_CHAR(person_info.birth_date, 'YYYY-MM-DD'));
   DBMS_OUTPUT.PUT_LINE('Address: ' || person_info.address.street || ', ' ||
                        person_info.address.city || ', ' || person_info.address.state ||
                        ' ' || person_info.address.zip);
END;
```

This example demonstrates how you can use nested records to create a more complex data structure that includes a person's name, birth date, and address information.

Combining Collections and Records in Nested Structures

You can take nested records a step further by combining them with collections. This allows you to create even more powerful and flexible data structures. Here's an example that uses a nested table of records:

```
DECLARE
  TYPE phone_rec_type IS RECORD (
    phone_type  VARCHAR2(10),
    phone_number VARCHAR2(20)
  );

  TYPE phone_table_type IS TABLE OF phone_rec_type;

  TYPE person_rec_type IS RECORD (
    person_id   NUMBER,
    name        name_rec_type,
    phones      phone_table_type
  );

  person_info person_rec_type;
BEGIN
  -- Initialize the nested table
  person_info.phones := phone_table_type();

  -- Assign values to the nested record
  person_info.person_id := 1;
  person_info.name.first_name := 'John';
  person_info.name.last_name := 'Doe';

  -- Add phone numbers to the nested table
  person_info.phones.EXTEND(2);
  person_info.phones(1) := phone_rec_type('Home', '555-1234');
  person_info.phones(2) := phone_rec_type('Mobile', '555-5678');

  -- Display the information
  DBMS_OUTPUT.PUT_LINE('Person ID: ' || person_info.person_id);
  DBMS_OUTPUT.PUT_LINE('Name: ' || person_info.name.first_name || ' ' || person_info.name.last_name);

  -- Display phone numbers
  FOR i IN 1..person_info.phones.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(person_info.phones(i).phone_type || ': ' ||
                         person_info.phones(i).phone_number);
  END LOOP;
END;
```

In this example, we've created a nested table of phone records within our person record. This allows us to store multiple phone numbers for each person, demonstrating how collections and nested records can be combined to create flexible, complex data structures.

Nested records provide a powerful way to organize and manage complex data in PL/SQL. By understanding how to create and use nested records, you can design more efficient and intuitive data structures for your database applications, regardless of the specific domain or industry you're working in.

Lab 16.3: Collections of Records

After this lab, you will be able to:

- Create and use Collections of Records
- Work with different types of Record Collections
- Populate and access Record Collections

Collections of Records

Collections of records allow you to store and manipulate multiple records in a single structure. This is particularly useful when you need to work with sets of complex data. In PL/SQL, you can create collections of records using three main types: associative arrays, nested tables, and varrays.

Let's explore each type with examples:

1. Associative Arrays of Records

Associative arrays (also known as PL/SQL tables or index-by tables) are the most flexible type of collection in PL/SQL. They can have a numeric or string index and can grow dynamically.

```
DECLARE
    -- Record type for employee information
    TYPE employee_rec_type IS RECORD (
        employee_id NUMBER,
        full_name   VARCHAR2(100),
        salary      NUMBER
    );

    -- Associative array type to store multiple employees
    TYPE employee_aa_type IS TABLE OF employee_rec_type
        INDEX BY PLS_INTEGER;

    -- Declare the associative array variable
    employee_aa employee_aa_type;

    -- Cursor to get first 5 employees
    CURSOR emp_cursor IS
        SELECT
            employee_id,
            first_name || ' ' || last_name AS full_name,
            salary
        FROM employees
        WHERE ROWNUM <= 5;

    -- Variable for counting records
    v_count NUMBER := 0;

BEGIN
    -- Store employee records in the associative array
    FOR emp_rec IN emp_cursor LOOP
        employee_aa(emp_rec.employee_id) := emp_rec;
        v_count := v_count + 1;
    END LOOP;

    -- Display header
    DBMS_OUTPUT.PUT_LINE('Employee List Report');
    DBMS_OUTPUT.PUT_LINE('-------------------');

    -- Check if we have any records
    IF employee_aa.COUNT > 0 THEN
        -- Display all stored employees
        FOR i IN employee_aa.FIRST . employee_aa.LAST LOOP
            IF employee_aa.EXISTS(i) THEN
                DBMS_OUTPUT.PUT_LINE(
                    'Employee ID: ' || employee_aa(i).employee_id ||
                    ', Name: ' || employee_aa(i).full_name ||
                    ', Salary: $' || TO_CHAR(employee_aa(i).salary, '99,999.
00')
            );
         END IF;
      END LOOP;

      -- Display total count
      DBMS_OUTPUT.PUT_LINE('-------------------');
      DBMS_OUTPUT.PUT_LINE('Total Employees: ' || v_count);
   ELSE
      DBMS_OUTPUT.PUT_LINE('No employees found.');
   END IF;

EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('No data found in employees table.');
   WHEN VALUE_ERROR THEN
      DBMS_OUTPUT.PUT_LINE('Error: Value too large for defined data type.');
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
END;
/
```

In this example, we create an associative array of employee records, indexed by the employee ID. We populate it using a cursor and then iterate through the array to display its contents.

2. Nested Tables of Records

Nested tables are similar to associative arrays but can be stored in the database and have some additional features.

```
DECLARE
    -- Create our department record structure
    TYPE department_rec_type IS RECORD (
        department_id   NUMBER,
        department_name VARCHAR2(30),
        manager_id      NUMBER
    );

    -- Create our nested table type
    TYPE department_nt_type IS TABLE OF department_rec_type;

    -- Initialize our nested table
    department_nt department_nt_type := department_nt_type();

    -- Variables for manager information
    v_manager_name VARCHAR2(100);
    v_dept_count  NUMBER := 0;

BEGIN
    -- Display header
    DBMS_OUTPUT.PUT_LINE('Department Summary Report');
    DBMS_OUTPUT.PUT_LINE('------------------------');

    -- Get department data with manager names
    FOR dept_rec IN (
        SELECT
            d.department_id,
            d.department_name,
            d.manager_id
        FROM departments d
        WHERE ROWNUM <= 5
        ORDER BY d.department_id
    ) LOOP
        -- Extend the nested table
        department_nt.EXTEND;

        -- Store the department record
        department_nt(department_nt.
LAST) := dept_rec;

    -- Count departments
    v_dept_count := v_dept_count + 1;

    -- Get manager name if exists
    BEGIN
      SELECT first_name || ' ' || last_name
      INTO v_manager_name
      FROM employees
      WHERE employee_id = dept_rec.manager_id;
    EXCEPTION
      WHEN NO_DATA_FOUND THEN
        v_manager_name := 'No Manager Assigned';
    END;

    -- Display department information
    DBMS_OUTPUT.PUT_LINE(
      'Department ID: ' || dept_rec.department_id || CHR(10) ||
      'Name: ' || dept_rec.department_name || CHR(10) ||
      'Manager ID: ' || NVL(TO_CHAR(dept_rec.manager_id), 'None') || CHR(10) ||
      'Manager Name: ' || v_manager_name || CHR(10) ||
      '-----------------------'
    );
  END LOOP;

  -- Display summary
  DBMS_OUTPUT.PUT_LINE(CHR(10) || 'Summary:');
  DBMS_OUTPUT.PUT_LINE('Total Departments: ' || v_dept_count);

  -- Display contents of nested table
  DBMS_OUTPUT.PUT_LINE(CHR(10) || 'Nested Table Contents:');
  DBMS_OUTPUT.PUT_LINE('-----------------------');

  IF department_nt.COUNT > 0 THEN
    FOR i IN 1 . department_nt.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE(
        'Index ' || i || ': ' ||
        'Dept ID: ' || department_nt(i).department_id || ', ' ||
        'Name: ' || department_nt(i).department_name
      );
    END LOOP;
  ELSE
    DBMS_OUTPUT.PUT_LINE('No departments found.
');
  END IF;

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No departments found in the database.');
  WHEN VALUE_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('Error: Value too large for defined data type.');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
    -- Rollback any changes if necessary
    ROLLBACK;
END;
/
```

In this example, we create a nested table of department records. We populate it using a cursor and then iterate through the table to display its contents.

3. Varrays of Records

Varrays (variable-size arrays) are similar to nested tables but have a maximum size and maintain their order.

```
DECLARE
  -- Define a record type
  TYPE job_rec_type IS RECORD (
    job_id     VARCHAR2(10),
    job_title  VARCHAR2(35),
    min_salary NUMBER
  );

  -- Define a varray type of records
  TYPE job_va_type IS VARRAY(10) OF job_rec_type;

  -- Declare a variable of the varray type
  job_va job_va_type := job_va_type();

  -- Cursor to fetch job data
  CURSOR job_cursor IS
    SELECT job_id, job_title, min_salary
    FROM jobs
    WHERE ROWNUM <= 5;

BEGIN
  -- Populate the varray
  FOR job_rec IN job_cursor LOOP
    job_va.EXTEND;
    job_va(job_va.LAST) := job_rec;
  END LOOP;

  -- Display the contents of the varray
  FOR i IN 1 .. job_va.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('Job ID: ' || job_va(i).job_id ||
                         ', Title: ' || job_va(i).job_title ||
                         ', Min Salary: ' || job_va(i).min_salary);
  END LOOP;
END;
```

In this example, we create a varray of job records with a maximum size of 10. We populate it using a cursor and then iterate through the varray to display its contents.

Choosing the Right Collection Type

- Use associative arrays when you need a flexible, memory-only collection with arbitrary indexes.
- Use nested tables when you need to store the collection in the database or when the number of elements can vary significantly.
- Use varrays when you have a fixed maximum number of elements and want to preserve their order.

Collections of records provide a powerful way to work with sets of complex data in PL/SQL. By combining the flexibility of collections with the structure of records, you can create efficient and intuitive data structures for a wide range of database applications.

Advanced Data Structures in PL/SQL