Chapater 19

Bulk Operations in PL/SQL

Mayko Freitas da Silva

Introduction to PL/SQL Bulk Processing Features

PL/SQL bulk processing features represent a significant advancement in how Oracle databases handle large-scale data operations. This content explores three fundamental aspects of bulk processing: FORALL statements, BULK COLLECT clauses, and collection binding in SQL statements. These features are essential for developers who need to optimize their PL/SQL code for performance when working with large datasets.

The content is structured into three main labs:

1. Lab 18.1 focuses on FORALL statements, demonstrating how to perform bulk DML operations efficiently and handle exceptions during these operations.
2. Lab 18.2 covers the BULK COLLECT clause, showing how to fetch multiple rows at once and manage memory effectively when retrieving large result sets.
3. Lab 18.3 explores binding collections in SQL statements, particularly in the context of dynamic SQL and cursor operations.

Throughout these labs, you'll learn how to move beyond traditional row-by-row processing to more efficient bulk operations, significantly improving your application's performance. The examples use the HR schema to demonstrate practical, real-world applications of these concepts.

Whether you're optimizing existing code or developing new applications, understanding these bulk processing features is crucial for writing efficient PL/SQL programs that can handle large volumes of data effectively.

18.1 Understanding FORALL Statements

Database performance often hinges on how efficiently we handle bulk operations. When updating multiple records, the difference between single-row processing and bulk operations can be dramatic. The FORALL statement in PL/SQL provides a powerful solution for this common challenge.

The Problem with Traditional Loops

Consider a scenario where you need to update salaries for 1000 employees. Using a traditional FOR loop, each update requires a separate round trip to the database:

```
FOR i IN 1..1000 LOOP
  UPDATE employees
  SET salary = new_salaries(i)
  WHERE employee_id = emp_ids(i);
END LOOP;
```

This approach, while straightforward, creates significant overhead by executing 1000 separate SQL statements.

Introducing FORALL

The FORALL statement transforms multiple DML operations into a single bulk operation. Here's how to implement it:

```
DECLARE
  -- Define collection types
  TYPE emp_id_type IS TABLE OF employees.employee_id%TYPE
    INDEX BY PLS_INTEGER;
  TYPE salary_type IS TABLE OF employees.salary%TYPE
    INDEX BY PLS_INTEGER;

  l_emp_ids    emp_id_type;
  l_salaries   salary_type;
```

The code above creates two collection types to store employee IDs and salaries. Using PLS_INTEGER as the index type optimizes performance for large collections.

Implementing Bulk Operations

Let's compare traditional and FORALL approaches:

```
-- Populate collections
BEGIN
  FOR i IN 1..100 LOOP
    l_emp_ids(i) := i + 1000;
    l_salaries(i) := 5000;
  END LOOP;

  -- FORALL implementation
  FORALL i IN 1..100
    UPDATE employees_copy
    SET salary = l_salaries(i)
    WHERE employee_id = l_emp_ids(i);
END;
```

The FORALL statement processes all updates in a single database operation, significantly reducing network traffic and processing overhead.

Performance Comparison

To illustrate the performance difference, we can measure execution times:

```
DECLARE
  l_start_time NUMBER;
  l_end_time   NUMBER;
BEGIN
  -- Measure FORALL performance
  l_start_time := DBMS_UTILITY.GET_TIME;

  FORALL i IN 1..100
    UPDATE employees_copy
    SET salary = l_salaries(i)
    WHERE employee_id = l_emp_ids(i);

  l_end_time := DBMS_UTILITY.GET_TIME;
  DBMS_OUTPUT.PUT_LINE('FORALL duration: ' ||
    TO_CHAR(l_end_time - l_start_time) || ' hsecs');
END;
```

Key Benefits of FORALL

1. **Reduced Network Traffic**: Single round trip to the database
2. **Improved Performance**: Especially noticeable with large datasets
3. **Memory Efficiency**: Better utilization of system resources
4. **Simplified Code**: Cleaner implementation for bulk operations

Best Practices

- Use FORALL for DML operations affecting multiple rows
- Consider collection size limitations in your Oracle version
- Implement error handling for bulk operations
- Test performance with representative data volumes

18.2 Error Handling with SAVE EXCEPTIONS

In real-world applications, data operations don't always proceed perfectly. The SAVE EXCEPTIONS clause provides a robust mechanism for handling errors in bulk operations while maintaining processing efficiency.

Setting Up Error Handling

```
-- Enable DBMS_OUTPUT in your client
-- Example command for SQL*Plus:
-- SET SERVEROUTPUT ON;

DECLARE
  -- Define collection types for employee data
  TYPE emp_id_type IS TABLE OF employees.employee_id%TYPE INDEX BY PLS_INTEGER;
  TYPE salary_type IS TABLE OF employees.salary%TYPE INDEX BY PLS_INTEGER;

  l_emp_ids    emp_id_type;   -- Collection for employee IDs
  l_salaries   salary_type;   -- Collection for salaries

  -- Define exception for bulk errors
  l_bulk_errors EXCEPTION;
  PRAGMA EXCEPTION_INIT(l_bulk_errors, -24381);

BEGIN
  -- Populate collections with some invalid data
  FOR i IN 1.5 LOOP
    l_emp_ids(i) := i + 100; -- Employee IDs (e.g., 101, 102, etc.)
    l_salaries(i) := CASE
                       WHEN i = 2 THEN -1000  -- Invalid salary
                       WHEN i = 4 THEN -2000  -- Invalid salary
                       ELSE 5000              -- Valid salary for others
                     END;
  END LOOP;

  -- Attempt bulk update with SAVE EXCEPTIONS
  BEGIN
    FORALL i IN 1.5 SAVE EXCEPTIONS
      UPDATE employees_copy
      SET salary = l_salaries(i)
      WHERE employee_id = l_emp_ids(i);

    COMMIT;  -- Commit valid updates
EXCEPTION
    WHEN l_bulk_errors THEN
      DBMS_OUTPUT.PUT_LINE('Exception block entered.');
      DBMS_OUTPUT.PUT_LINE('Number of failures: ' || SQL%BULK_EXCEPTIONS.COUNT);

      FOR i IN 1.SQL%BULK_EXCEPTIONS.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(
          'Error ' || i || ' occurred during iteration ' ||
          SQL%BULK_EXCEPTIONS(i).ERROR_INDEX
        );
        DBMS_OUTPUT.
```

```
PUT_LINE(
            'Error message: ' ||
            SQLERRM(-SQL%BULK_EXCEPTIONS(i).ERROR_CODE)
          );
      END LOOP;

   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Unexpected error: ' || SQLERRM); -- Catch any other unforeseen errors
END;
END;
```

Let's break this down.

First, we need to establish the foundation for catching bulk operation errors:

```
-- Enable DBMS_OUTPUT in your client
-- Example command for SQL*Plus:
-- SET SERVEROUTPUT ON;

DECLARE
 -- Define collection types for employee data
 TYPE emp_id_type IS TABLE OF employees.employee_id%TYPE INDEX BY PLS_INTEGER;
 TYPE salary_type IS TABLE OF employees.salary%TYPE INDEX BY PLS_INTEGER;

 l_emp_ids    emp_id_type;   -- Collection for employee IDs
 l_salaries   salary_type;   -- Collection for salaries

 -- Define exception for bulk errors
 l_bulk_errors EXCEPTION;
 PRAGMA EXCEPTION_INIT(l_bulk_errors, -24381);
```

The PRAGMA EXCEPTION_INIT statement links our custom exception to Oracle's bulk error code (-24381), enabling structured error handling.

Implementing SAVE EXCEPTIONS

Here's how to use SAVE EXCEPTIONS in a bulk operation:

```
BEGIN
 -- Populate collections with some invalid data
 FOR i IN 1..5 LOOP
  l_emp_ids(i) := i + 100; -- Employee IDs (e.g., 101, 102, etc.)
  l_salaries(i) := CASE
                    WHEN i = 2 THEN -1000  -- Invalid salary
                    WHEN i = 4 THEN -2000  -- Invalid salary
                    ELSE 5000              -- Valid salary for others
                   END;
 END LOOP;

 -- Attempt bulk update with SAVE EXCEPTIONS
 BEGIN
  FORALL i IN 1..5 SAVE EXCEPTIONS
   UPDATE employees_copy
   SET salary = l_salaries(i)
   WHERE employee_id = l_emp_ids(i);

  COMMIT;  -- Commit valid updates
```

Comprehensive Error Handling

When errors occur, SAVE EXCEPTIONS allows us to capture and process them systematically:

```
EXCEPTION
  WHEN l_bulk_errors THEN
    DBMS_OUTPUT.PUT_LINE('Exception block entered.');
    DBMS_OUTPUT.PUT_LINE('Number of failures: ' || SQL%BULK_EXCEPTIONS.COUNT);

    FOR i IN 1..SQL%BULK_EXCEPTIONS.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE(
        'Error ' || i || ' occurred during iteration ' ||
        SQL%BULK_EXCEPTIONS(i).ERROR_INDEX
      );
      DBMS_OUTPUT.PUT_LINE(
        'Error message: ' ||
        SQLERRM(-SQL%BULK_EXCEPTIONS(i).ERROR_CODE)
      );
    END LOOP;

  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Unexpected error: ' || SQLERRM); -- Catch any other unforeseen errors
END;
END;
/
```

Key Components of Error Handling

1. **SQL%BULK_EXCEPTIONS**: A built-in structure containing error information
2. **ERROR_INDEX**: Identifies the iteration where the error occurred
3. **ERROR_CODE**: Provides the Oracle error code
4. **SQLERRM**: Converts error codes to readable messages

Practical Applications

The SAVE EXCEPTIONS mechanism enables several essential capabilities:

- **Partial Success**: Valid records are processed even when some fail
- **Detailed Error Reporting**: Comprehensive information about each failure
- **Efficient Recovery**: Process only failed records in subsequent runs
- **Audit Trail**: Track and document processing issues

Best Practices

Always use SAVE EXCEPTIONS for production bulk operationsImplement appropriate logging of errorsConsider automated error handling proceduresPlan for error recovery scenariosTest with various error conditions

Error Processing Strategies

When handling bulk operation errors, consider implementing:

- Automated error notification systems
- Error logging to persistent storage
- Recovery procedures for common error types
- Business rules for handling partial successes
- Validation procedures to prevent common errors

Common Use Cases

1. **Data Migration**: Handle incomplete or invalid source data
2. **Batch Updates**: Process large datasets with potential inconsistencies
3. **Integration Operations**: Manage errors in cross-system data operations
4. **Data Cleanup**: Process potentially problematic records safely

Using the INDICES OF Option

When working with collections, you may encounter situations where some elements have been deleted, creating what's known as a sparse collection. The INDICES OF option allows FORALL to process only the existing elements in such collections efficiently.

Let's explore this feature step by step:

First, let's declare our collection types and variables:

```
DECLARE
  TYPE emp_id_type IS TABLE OF employees.employee_id%TYPE
    INDEX BY PLS_INTEGER;
  TYPE salary_type IS TABLE OF employees.salary%TYPE
    INDEX BY PLS_INTEGER;

  l_emp_ids    emp_id_type;
  l_salaries   salary_type;
```

These collections will store employee IDs and salaries, but some elements will be deliberately removed.

Next, let's populate our collections:

```
  -- Populate collections
  FOR i IN 1..5 LOOP
    l_emp_ids(i) := i + 100;
    l_salaries(i) := 5000;
  END LOOP;
```

Now, let's create sparse collections by deleting some elements:

```
  -- Delete some elements to create sparse collections
  l_emp_ids.DELETE(2);
  l_salaries.DELETE(2);
  l_emp_ids.DELETE(4);
  l_salaries.DELETE(4);
```

After these deletions:

- Element 2 is removed (originally index 102)
- Element 4 is removed (originally index 104)
- Remaining elements are at positions 1, 3, and 5

Finally, we use INDICES OF to process only the existing elements:

```
-- Use INDICES OF to process only existing elements
FORALL i IN INDICES OF l_emp_ids
  UPDATE employees_copy
  SET salary = l_salaries(i)
  WHERE employee_id = l_emp_ids(i);

DBMS_OUTPUT.PUT_LINE('Number of rows updated: ' || SQL%ROWCOUNT);
```

The INDICES OF clause ensures that:

- Only existing elements are processed
- Deleted elements are skipped
- The collection indexes are properly maintained

Here's the complete code with example output:

```
DECLARE
  TYPE emp_id_type IS TABLE OF employees.employee_id%TYPE
    INDEX BY PLS_INTEGER;
  TYPE salary_type IS TABLE OF employees.salary%TYPE
    INDEX BY PLS_INTEGER;

  l_emp_ids    emp_id_type;
  l_salaries   salary_type;
BEGIN
  -- Populate collections
  FOR i IN 1..5 LOOP
    l_emp_ids(i) := i + 100;
    l_salaries(i) := 5000;
  END LOOP;

  -- Delete some elements to create sparse collections
  l_emp_ids.DELETE(2);
  l_salaries.DELETE(2);
  l_emp_ids.DELETE(4);
  l_salaries.DELETE(4);

  -- Use INDICES OF to process only existing elements
  FORALL i IN INDICES OF l_emp_ids
    UPDATE employees_copy
    SET salary = l_salaries(i)
    WHERE employee_id = l_emp_ids(i);

  DBMS_OUTPUT.PUT_LINE('Number of rows updated: ' || SQL%ROWCOUNT);
END;
/
```

Key points about INDICES OF:

- Perfect for working with sparse collections
- Automatically skips deleted elements
- Maintains proper index relationships
- More efficient than checking for existence in a loop
- Particularly useful when processing filtered datasets

A practical example of when to use INDICES OF:

- When processing a filtered subset of employees
- After removing invalid or unwanted records
- When working with partially populated collections
- In data cleanup operations where some records are skipped

Using the VALUES OF Option

The VALUES OF option provides precise control over which collection elements to process in a FORALL statement. Instead of processing all elements or only existing ones, you can specify exactly which indices should be included in the operation.

Let's break down how to use this powerful feature:

First, we'll declare our collection types, including a new type for indices:

```
DECLARE
  TYPE emp_id_type IS TABLE OF employees.employee_id%TYPE
    INDEX BY PLS_INTEGER;
  TYPE salary_type IS TABLE OF employees.salary%TYPE
    INDEX BY PLS_INTEGER;
  TYPE index_type IS TABLE OF PLS_INTEGER
    INDEX BY PLS_INTEGER;

  l_emp_ids    emp_id_type;
  l_salaries   salary_type;
  l_indices    index_type;
```

Note the new index_type collection that will control which elements we process.

Next, let's populate our employee data collections:

```
  -- Populate collections
  FOR i IN 1..5 LOOP
    l_emp_ids(i) := i + 100;
    l_salaries(i) := 5000;
  END LOOP;
```

Now, we'll specify which indices we want to process:

```
-- Specify which indices to process
l_indices(1) := 1;  -- Will process first element
l_indices(2) := 3;  -- Will process third element
l_indices(3) := 5;  -- Will process fifth element
```

This selective approach allows us to:

- Choose specific elements to process
- Skip elements without deleting them
- Process elements in a different order if needed

Finally, we use VALUES OF to process only the specified indices:

```
-- Use VALUES OF to process only specified indices
FORALL i IN VALUES OF l_indices
  UPDATE employees_copy
  SET salary = l_salaries(i)
  WHERE employee_id = l_emp_ids(i);

DBMS_OUTPUT.PUT_LINE('Number of rows updated: ' || SQL%ROWCOUNT);
```

Here's the complete code with example output:

```
DECLARE
  TYPE emp_id_type IS TABLE OF employees.employee_id%TYPE
    INDEX BY PLS_INTEGER;
  TYPE salary_type IS TABLE OF employees.salary%TYPE
    INDEX BY PLS_INTEGER;
  TYPE index_type IS TABLE OF PLS_INTEGER
    INDEX BY PLS_INTEGER;

  l_emp_ids    emp_id_type;
  l_salaries   salary_type;
  l_indices    index_type;
BEGIN
  -- Populate collections
  FOR i IN 1..5 LOOP
    l_emp_ids(i) := i + 100;
    l_salaries(i) := 5000;
  END LOOP;

  -- Specify which indices to process
  l_indices(1) := 1;
  l_indices(2) := 3;
  l_indices(3) := 5;

  -- Use VALUES OF to process only specified indices
  FORALL i IN VALUES OF l_indices
    UPDATE employees_copy
    SET salary = l_salaries(i)
    WHERE employee_id = l_emp_ids(i);

  DBMS_OUTPUT.PUT_LINE('Number of rows updated: ' || SQL%ROWCOUNT);
END;
/
```

Key points about VALUES OF:

- Provides precise control over which elements to process
- Allows processing elements in a specific order
- Can be used to implement custom processing logic
- More flexible than INDICES OF for selective processing
- Useful when you need to process a specific subset of elements

Practical applications:

- Processing priority records first
- Implementing custom batch processing logic
- Handling special cases separately
- Following specific business rules for data processing
- Creating staged updates based on certain criteria

Lab 18.2 Using BULK COLLECT with SELECT Statements

BULK COLLECT is a powerful feature that enables you to fetch multiple rows at once into collections, significantly improving performance compared to row-by-row processing. This is particularly useful when dealing with large result sets.

Let's break down how to implement BULK COLLECT effectively:

First, let's declare our collection types:

```
DECLARE
 -- Define collection types for employee data
 TYPE t_emp_ids IS TABLE OF employees.employee_id%TYPE;
 TYPE t_emp_names IS TABLE OF employees.last_name%TYPE;
 TYPE t_emp_sals IS TABLE OF employees.salary%TYPE;

 l_emp_ids    t_emp_ids;
 l_emp_names  t_emp_names;
 l_emp_sals   t_emp_sals;
```

Note that we're using:

- Nested tables (not index-by tables) as they're better suited for BULK COLLECT
- %TYPE to ensure our collections match the database column types
- Separate collections for each column we want to fetch

Next, let's use BULK COLLECT to fetch the data:

```
-- Fetch multiple columns into separate collections
SELECT employee_id, last_name, salary
  BULK COLLECT INTO l_emp_ids, l_emp_names, l_emp_sals
  FROM employees
  WHERE department_id = 80;
```

This single statement:

- Fetches all matching rows at once
- Populates all three collections simultaneously
- Reduces context switching between SQL and PL/SQL engines

Finally, we'll display the results:

```
-- Display results
FOR i IN 1..l_emp_ids.COUNT LOOP
  DBMS_OUTPUT.PUT_LINE(
    'Employee: ' || l_emp_ids(i) ||
    ', Name: ' || l_emp_names(i) ||
    ', Salary: ' || l_emp_sals(i));
END LOOP;
```

Here's the complete code with example output:

```
DECLARE
  -- Define collection types for employee data
  TYPE t_emp_ids IS TABLE OF employees.employee_id%TYPE;
  TYPE t_emp_names IS TABLE OF employees.last_name%TYPE;
  TYPE t_emp_sals IS TABLE OF employees.salary%TYPE;

  l_emp_ids    t_emp_ids;
  l_emp_names  t_emp_names;
  l_emp_sals   t_emp_sals;
BEGIN
  -- Fetch multiple columns into separate collections
  SELECT employee_id, last_name, salary
    BULK COLLECT INTO l_emp_ids, l_emp_names, l_emp_sals
    FROM employees
    WHERE department_id = 80;

  -- Display results
  FOR i IN 1..l_emp_ids.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(
      'Employee: ' || l_emp_ids(i) ||
      ', Name: ' || l_emp_names(i) ||
      ', Salary: ' || l_emp_sals(i));
  END LOOP;
END;
/

-- Example Output:
Employee: 145, Name: Russell, Salary: 14000
Employee: 146, Name: Partners, Salary: 13500
Employee: 147, Name: Errazuriz, Salary: 12000
Employee: 148, Name: Cambrault, Salary: 11000
Employee: 149, Name: Zlotkey, Salary: 10500
```

Key points about BULK COLLECT:

- Significantly improves performance for large result sets
- Reduces context switching overhead
- Can fetch multiple columns simultaneously
- Collections are automatically sized to match the result set
- Works well with both simple and complex queries

Best practices:

1. Consider using LIMIT clause for very large result sets
2. Match collection types to column types using %TYPE
3. Ensure sufficient memory for the expected result set
4. Use meaningful collection names that reflect their content
5. Consider error handling for no data found situations

When to use BULK COLLECT:

- Fetching multiple rows for processing
- Loading lookup tables into memory
- Performing batch operations
- Building temporary result sets
- Improving performance of data-intensive operations

Using Record Types with BULK COLLECT

Record types provide a more structured and maintainable way to handle multiple columns when using BULK COLLECT. Instead of managing separate collections for each column, you can group related data into a single record structure.

Let's break down this approach:

First, we'll define our record and table types:

```
DECLARE
 -- Define record type based on employees table
 TYPE emp_rec_type IS RECORD (
  emp_id    employees.employee_id%TYPE,
  emp_name  employees.last_name%TYPE,
  emp_sal   employees.salary%TYPE
 );

 -- Define table type of records
 TYPE emp_tab_type IS TABLE OF emp_rec_type;

 l_emp_data emp_tab_type;
```

This structure provides:

- A custom record type (emp_rec_type) that groups related fields
- A table type (emp_tab_type) to hold multiple records
- Type safety through %TYPE declarations

Next, let's fetch the data using BULK COLLECT:

```
-- Fetch multiple rows into collection of records
SELECT employee_id, last_name, salary
  BULK COLLECT INTO l_emp_data
  FROM employees
  WHERE department_id = 60;
```

Notice how:

- Multiple columns are fetched into a single collection
- The column order matches the record structure
- Each row becomes one record in the collection

Finally, we'll display the results using record notation:

```
-- Display results
FOR i IN 1..l_emp_data.COUNT LOOP
  DBMS_OUTPUT.PUT_LINE(
    'Employee: ' || l_emp_data(i).emp_id ||
    ', Name: ' || l_emp_data(i).emp_name ||
    ', Salary: ' || l_emp_data(i).emp_sal);
END LOOP;
```

Here's the complete code with example output:

```
DECLARE
 -- Define record type based on employees table
 TYPE emp_rec_type IS RECORD (
  emp_id    employees.employee_id%TYPE,
  emp_name  employees.last_name%TYPE,
  emp_sal   employees.salary%TYPE
 );

 -- Define table type of records
 TYPE emp_tab_type IS TABLE OF emp_rec_type;

 l_emp_data emp_tab_type;
BEGIN
 -- Fetch multiple rows into collection of records
 SELECT employee_id, last_name, salary
  BULK COLLECT INTO l_emp_data
  FROM employees
  WHERE department_id = 60;

 -- Display results
 FOR i IN 1..l_emp_data.COUNT LOOP
  DBMS_OUTPUT.PUT_LINE(
   'Employee: ' || l_emp_data(i).emp_id ||
   ', Name: ' || l_emp_data(i).emp_name ||
   ', Salary: ' || l_emp_data(i).emp_sal);
 END LOOP;
END;
/

-- Example Output:
Employee: 103, Name: Hunold, Salary: 9000
Employee: 104, Name: Ernst, Salary: 6000
Employee: 105, Name: Austin, Salary: 4800
Employee: 106, Name: Pataballa, Salary: 4800
Employee: 107, Name: Lorentz, Salary: 4200
```

Key advantages of using record types:

1. Better code organization
2. Reduced variable declarations
3. More maintainable code structure
4. Logical grouping of related data
5. Easier to pass data between procedures

Best practices:

- Use meaningful field names in record types
- Keep related fields together in the record structure
- Consider adding documentation comments for complex records
- Use %TYPE to ensure type compatibility
- Consider creating package-level types for reuse

Common use cases:

- Processing employee data
- Handling transaction records
- Working with related column sets
- Building data transformation logic
- Creating temporary result structures

Using BULK COLLECT with Cursors and LIMIT Clause

When working with large datasets, it's crucial to manage memory efficiently. The combination of cursors, BULK COLLECT, and the LIMIT clause provides a powerful way to process large result sets in manageable chunks.

Let's break down this advanced approach:

First, let's declare our cursor and types:

```
DECLARE
  CURSOR emp_cur IS
    SELECT employee_id, last_name, salary
    FROM employees
    WHERE department_id = 50;

  TYPE emp_rec_type IS RECORD (
    emp_id    employees.employee_id%TYPE,
    emp_name  employees.last_name%TYPE,
    emp_sal   employees.salary%TYPE
  );

  TYPE emp_tab_type IS TABLE OF emp_rec_type;

  l_emp_data emp_tab_type;
  l_batch_size PLS_INTEGER := 10; -- Process 10 rows at a time
```

Key components:

- A cursor defining the result set
- A record type to hold row data
- A table type for the collection
- A batch size to control memory usage

Now, let's implement the batch processing logic:

```
BEGIN
  OPEN emp_cur;

  LOOP
    FETCH emp_cur
    BULK COLLECT INTO l_emp_data
    LIMIT l_batch_size;

    EXIT WHEN l_emp_data.COUNT = 0;

    -- Process current batch
    FOR i IN 1..l_emp_data.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE(
        'Processing Employee: ' || l_emp_data(i).emp_id ||
        ', Name: ' || l_emp_data(i).emp_name);
    END LOOP;
  END LOOP;

  CLOSE emp_cur;
```

Here's the complete code with example output:

```
DECLARE
  CURSOR emp_cur IS
    SELECT employee_id, last_name, salary
    FROM employees
    WHERE department_id = 50;

  TYPE emp_rec_type IS RECORD (
    emp_id    employees.employee_id%TYPE,
    emp_name  employees.last_name%TYPE,
    emp_sal   employees.salary%TYPE
  );

  TYPE emp_tab_type IS TABLE OF emp_rec_type;

  l_emp_data emp_tab_type;
  l_batch_size PLS_INTEGER := 10; -- Process 10 rows at a time
BEGIN
  OPEN emp_cur;

  LOOP
    FETCH emp_cur
    BULK COLLECT INTO l_emp_data
    LIMIT l_batch_size;

    EXIT WHEN l_emp_data.COUNT = 0;

    -- Process current batch
    FOR i IN 1..l_emp_data.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE(
        'Processing Employee: ' || l_emp_data(i).emp_id ||
        ', Name: ' || l_emp_data(i).emp_name);
    END LOOP;
  END LOOP;

  CLOSE emp_cur;
END;
/

-- Example Output:
Processing Employee: 120, Name: Weiss
Processing Employee: 121, Name: Fripp
Processing Employee: 122, Name: Kaufling
Processing Employee: 123, Name: Vollman
Processing Employee: 124, Name: Mourgos
-- (Additional rows up to batch size)
```

Key benefits of this approach:

1. Controlled memory usage
2. Efficient processing of large datasets
3. Better resource management
4. Reduced system impact
5. Improved scalability

Best practices for batch processing:

- Choose appropriate batch sizes based on:
    - Available memory
    - Record size
    - Processing requirements
    - System resources
- Consider adding error handling
- Monitor performance metrics
- Include progress logging for long-running operations
- Implement transaction control if needed

Determining optimal batch size:

- Start with a moderate size (e.g., 100-1000)
- Monitor memory usage
- Test with representative data volumes
- Adjust based on performance metrics
- Consider system-wide impact

Example batch size guidelines:

```
-- Small records, lots of memory
l_batch_size := 1000;

-- Large records or limited memory
l_batch_size := 100;

-- Very large records or complex processing
l_batch_size := 50;
```

Using BULK COLLECT with the RETURNING Clause

The RETURNING clause combined with BULK COLLECT provides a powerful way to capture the results of DML operations (INSERT, UPDATE, DELETE) without requiring additional queries. This feature helps maintain data consistency and improve performance.

Let's break down this functionality:

First, let's declare our collection types:

```
DECLARE
  TYPE num_tab_type IS TABLE OF employees.employee_id%TYPE;
  TYPE name_tab_type IS TABLE OF employees.last_name%TYPE;

  l_emp_ids    num_tab_type;
  l_emp_names  name_tab_type;
```

Now, let's use RETURNING with BULK COLLECT in an UPDATE statement:

```
BEGIN
  -- Update salaries and collect affected employee information
  UPDATE employees
  SET salary = salary * 1.1
  WHERE department_id = 30
  RETURNING employee_id, last_name
  BULK COLLECT INTO l_emp_ids, l_emp_names;
```

This statement:

- Updates employee salaries
- Immediately captures affected records
- Stores results in collections
- Performs everything in a single operation

Here's the complete code with example output:

```
DECLARE
  TYPE num_tab_type IS TABLE OF employees.employee_id%TYPE;
  TYPE name_tab_type IS TABLE OF employees.last_name%TYPE;

  l_emp_ids    num_tab_type;
  l_emp_names  name_tab_type;
BEGIN
  -- Update salaries and collect affected employee information
  UPDATE employees
  SET salary = salary * 1.1
  WHERE department_id = 30
  RETURNING employee_id, last_name
  BULK COLLECT INTO l_emp_ids, l_emp_names;

  -- Display affected employees
  FOR i IN 1..l_emp_ids.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(
      'Updated Employee: ' || l_emp_ids(i) ||
      ', Name: ' || l_emp_names(i));
  END LOOP;

  ROLLBACK; -- Rollback changes for demonstration
END;
/

-- Example Output:
Updated Employee: 114, Name: Raphaely
Updated Employee: 115, Name: Khoo
Updated Employee: 116, Name: Baida
Updated Employee: 117, Name: Tobias
Updated Employee: 118, Name: Himuro
Updated Employee: 119, Name: Colmenares
```

Key benefits of RETURNING clause with BULK COLLECT:

1. Single-pass operation
2. Atomic data capture
3. Improved performance
4. Guaranteed data consistency
5. Reduced network traffic

Common use cases:

```
-- Capture inserted data
INSERT INTO employees (employee_id, last_name, salary)
VALUES (...)
RETURNING employee_id, last_name
BULK COLLECT INTO l_emp_ids, l_emp_names;

-- Capture deleted data
DELETE FROM employees
WHERE department_id = 30
RETURNING employee_id, last_name
BULK COLLECT INTO l_emp_ids, l_emp_names;

-- Capture multiple columns
UPDATE employees
SET salary = salary * 1.1
RETURNING
  employee_id,
  last_name,
  salary,
  hire_date
BULK COLLECT INTO
  l_emp_ids,
  l_emp_names,
  l_salaries,
  l_hire_dates;
```

Best practices:

1. Match collection types to returned columns
2. Include error handling
3. Consider transaction management
4. Document the expected results
5. Validate collection contents when needed

Error handling example:

```
DECLARE
   -- Define collection types
   TYPE emp_id_type IS TABLE OF employees.employee_id%TYPE;
   TYPE name_type IS TABLE OF employees.last_name%TYPE;

   -- Declare collections
   l_emp_ids    emp_id_type;
   l_emp_names  name_type;

   -- Custom exception
   no_data_found EXCEPTION;
BEGIN
   UPDATE employees
   SET salary = salary * 1.1
   WHERE department_id = 999  -- Non-existent department
   RETURNING employee_id, last_name
   BULK COLLECT INTO l_emp_ids, l_emp_names;

   IF l_emp_ids.COUNT = 0 THEN
      RAISE no_data_found;
   END IF;

EXCEPTION
   WHEN no_data_found THEN
      DBMS_OUTPUT.PUT_LINE('No employees were updated');
END;
/
```

Lab 18.3: Binding Collections in SQL Statements

Binding Collections When Using EXECUTE IMMEDIATE Statements

When working with dynamic SQL and collections, it's important to understand that we cannot bind entire collections directly in EXECUTE IMMEDIATE statements. Instead, we need to work with individual collection elements. Let's explore how to do this correctly using the HR schema.

Basic Example - Processing Collection Elements

```
DECLARE
   -- Define collection types for employee data
   TYPE emp_id_tab_type IS TABLE OF employees.employee_id%TYPE
      INDEX BY PLS_INTEGER;
   TYPE emp_name_tab_type IS TABLE OF employees.last_name%TYPE
      INDEX BY PLS_INTEGER;

   -- Declare collection variables
   emp_ids    emp_id_tab_type;
   emp_names  emp_name_tab_type;

   -- Variable for dynamic SQL
   v_sql  VARCHAR2(1000);
BEGIN
   -- Step 1: Populate collections with employee data
   FOR i IN 100..105 LOOP  -- Using first 6 employees
      emp_ids(i-99) := i;
      SELECT last_name
      INTO emp_names(i-99)
      FROM employees
      WHERE employee_id = i;
   END LOOP;

   -- Step 2: Process and display data using dynamic SQL
   FOR i IN emp_ids.FIRST..emp_ids.LAST LOOP
      v_sql := 'BEGIN
                  DBMS_OUTPUT.PUT_LINE(:1 || '': '' || :2);
                END;';

      EXECUTE IMMEDIATE v_sql
      USING emp_ids(i), emp_names(i);
   END LOOP;
END;
/
```

Expected output:

```
100: King
101: Kochhar
102: De Haan
103: Hunold
104: Ernst
105: Austin
```

Advanced Example with Multiple Operations

```
DECLARE
   -- Define collection types
   TYPE emp_id_tab_type IS TABLE OF employees.employee_id%TYPE
      INDEX BY PLS_INTEGER;
   TYPE emp_salary_tab_type IS TABLE OF employees.salary%TYPE
      INDEX BY PLS_INTEGER;
   TYPE emp_name_tab_type IS TABLE OF employees.last_name%TYPE
      INDEX BY PLS_INTEGER;

   -- Declare collection variables
   emp_ids      emp_id_tab_type;
   emp_salaries emp_salary_tab_type;
   emp_names    emp_name_tab_type;

   -- Variables for dynamic SQL and processing
   v_sql       VARCHAR2(2000);
   v_dept_id   employees.department_id%TYPE := 60;  -- IT department
   v_old_salary employees.salary%TYPE;
   v_new_salary employees.salary%TYPE;
   idx         NUMBER := 1;
BEGIN
   -- Step 1: Collect current employee data
   FOR emp_rec IN (SELECT employee_id, last_name, salary
                   FROM employees
                   WHERE department_id = v_dept_id) LOOP
      emp_ids(idx) := emp_rec.employee_id;
      emp_names(idx) := emp_rec.last_name;
      emp_salaries(idx) := emp_rec.salary;
      idx := idx + 1;
   END LOOP;

   -- Step 2: Display current salaries
   DBMS_OUTPUT.PUT_LINE('Current Salaries:');
   FOR i IN emp_ids.FIRST..emp_ids.LAST LOOP
      v_sql := 'BEGIN
                  DBMS_OUTPUT.PUT_LINE(:1 || '' - '' || :2 ||
                                       '': $'' || :3);
               END;';

      EXECUTE IMMEDIATE v_sql
      USING emp_ids(i), emp_names(i), emp_salaries(i);
   END LOOP;

   -- Step 3: Update salaries using dynamic SQL
   DBMS_OUTPUT.PUT_LINE(CHR(10) || 'Processing Salary Updates.');
   FOR i IN emp_ids.FIRST..emp_ids.
```

```
LAST LOOP
        -- Store old salary
        v_old_salary := emp_salaries(i);

        -- Update salary
        v_sql := 'UPDATE employees
                SET salary = salary * 1.01
                WHERE employee_id = :1
                RETURNING salary INTO :2';

        EXECUTE IMMEDIATE v_sql
        USING emp_ids(i)
        RETURNING INTO v_new_salary;

        -- Store new salary in collection
        emp_salaries(i) := v_new_salary;

        -- Display update information
        v_sql := 'BEGIN
                 DBMS_OUTPUT.PUT_LINE(:1 || '' - '' || :2 ||
                                '': $'' || :3 || '' -> $'' || :4);
                END;';

        EXECUTE IMMEDIATE v_sql
        USING emp_ids(i), emp_names(i), v_old_salary, v_new_salary;
    END LOOP;

    -- Step 4: Commit the transaction
    COMMIT;

    DBMS_OUTPUT.PUT_LINE(CHR(10) || 'Salary updates completed and committed.');
END;
```

Key Points to Remember:

1. Collection Binding Rules:
    - You cannot bind entire collections in EXECUTE IMMEDIATE
    - Bind individual collection elements instead
    - Each bound element must be a SQL type
2. Processing Collections:
    - Use PL/SQL loops to process collection elements
    - Access collection methods (FIRST, LAST) in PL/SQL, not in dynamic SQL
    - Bind scalar values from collections one at a time
3. Dynamic SQL Structure:
    - Keep dynamic SQL simple when working with collections
    - Move complex logic to PL/SQL where possible
    - Use proper bind variables for each scalar value

Common Mistakes to Avoid:

❌ Don't try this (will generate error):

```
-- Wrong: Cannot bind entire collections
EXECUTE IMMEDIATE 'BEGIN ... END;'
USING collection_name;

-- Wrong: Cannot use collection methods in dynamic SQL
v_sql := 'FOR i IN :1.FIRST..:1.LAST LOOP';

-- Wrong: Cannot index collections in dynamic SQL
v_sql := 'DBMS_OUTPUT.PUT_LINE(:1(i))';
```

✅ Do this instead:

```
-- Correct: Process collections in PL/SQL
FOR i IN collection_name.FIRST..collection_name.LAST LOOP
   EXECUTE IMMEDIATE v_sql
   USING collection_name(i);
END LOOP;
```

Practice Exercise:

Modify the example to display both employee ID and salary for employees in department 60 (IT department) using dynamic SQL and collections.

Using Collections with OPEN FOR Statements

When using REF CURSOR and collections together, we can process multiple rows efficiently. Here's how to use collections with OPEN FOR statements:

Basic Example Using OPEN FOR with Collections

```
DECLARE
   -- Cursor and record type definitions
   TYPE emp_cur_type IS REF CURSOR;
   TYPE emp_rec_type IS RECORD (
      emp_id    employees.employee_id%TYPE,
      emp_name  employees.last_name%TYPE,
      salary    employees.salary%TYPE
   );

   -- Collection type for storing employee records
   TYPE emp_tab_type IS TABLE OF emp_rec_type
      INDEX BY PLS_INTEGER;

   -- Variables
   emp_cur     emp_cur_type;
   emp_tab     emp_tab_type;
   v_sql       VARCHAR2(1000);
   v_dept_id   employees.department_id%TYPE := 60; -- IT Department
BEGIN
   -- Prepare dynamic query
   v_sql := 'SELECT employee_id, last_name, salary
             FROM employees
             WHERE department_id = :1';

   -- Open cursor using dynamic SQL
   OPEN emp_cur FOR v_sql USING v_dept_id;

   -- Fetch all rows into collection
   FETCH emp_cur
   BULK COLLECT INTO emp_tab;

   -- Close cursor
   CLOSE emp_cur;

   -- Process collected data
   FOR i IN emp_tab.FIRST..emp_tab.LAST LOOP
      DBMS_OUTPUT.PUT_LINE(
         'Employee: ' || emp_tab(i).emp_id ||
         ' - ' || emp_tab(i).emp_name ||
         ', Salary: $' || emp_tab(i).salary
      );
   END LOOP;
END;
```

Expected output:

```
Employee: 103 - Hunold, Salary: \$9000
Employee: 104 - Ernst, Salary: \$6000
Employee: 105 - Austin, Salary: \$4800
Employee: 106 - Pataballa, Salary: \$4800
Employee: 107 - Lorentz, Salary: $4200
```

Advanced Example with Multiple Operations and Error Handling

```
DECLARE
   -- Cursor types
   TYPE emp_cur_type IS REF CURSOR;

   -- Record types
   TYPE emp_rec_type IS RECORD (
      emp_id    employees.employee_id%TYPE,
      emp_name  employees.last_name%TYPE,
      dept_id   employees.department_id%TYPE,
      salary    employees.salary%TYPE
   );

   -- Collection types
   TYPE emp_tab_type IS TABLE OF emp_rec_type
      INDEX BY PLS_INTEGER;

   -- Variables
   emp_cur     emp_cur_type;
   emp_tab     emp_tab_type;
   v_sql       VARCHAR2(1000);
   v_min_salary NUMBER := 5000;
   v_dept_list  VARCHAR2(100) := '60, 90, 100'; -- IT, Executive, Finance

   -- For error handling
   v_err_count NUMBER := 0;
BEGIN
   -- Dynamic query with multiple conditions
   v_sql := 'SELECT e.employee_id, e.last_name,
                    e.department_id, e.salary
             FROM employees e
             WHERE e.department_id IN (' || v_dept_list || ')
             AND e.salary >= :1
             ORDER BY e.salary DESC';

   -- Open cursor with parameter
   OPEN emp_cur FOR v_sql USING v_min_salary;

   -- Fetch data in batches of 10 rows
   LOOP
      FETCH emp_cur
      BULK COLLECT INTO emp_tab LIMIT 10;

      EXIT WHEN emp_tab.COUNT = 0;

      -- Process each batch
      DBMS_OUTPUT.PUT_LINE('Processing batch of ' || emp_tab.COUNT || ' records:');
      DBMS_OUTPUT.
```

```
PUT_LINE('---------------------------------------');

    FOR i IN emp_tab.FIRST..emp_tab.LAST LOOP
      BEGIN
        -- Display employee information
        DBMS_OUTPUT.PUT_LINE(
          'Dept ' || LPAD(emp_tab(i).dept_id, 3) ||
          ': Employee ' || LPAD(emp_tab(i).emp_id, 3) ||
          ' - ' || RPAD(emp_tab(i).emp_name, 20) ||
          ' Salary: $' || TO_CHAR(emp_tab(i).salary, '999,999')
        );

        -- Simulate some processing
        IF emp_tab(i).salary > 15000 THEN
          DBMS_OUTPUT.PUT_LINE('  ** High salary flagged for review **');
        END IF;

      EXCEPTION
        WHEN OTHERS THEN
          v_err_count := v_err_count + 1;
          DBMS_OUTPUT.PUT_LINE('Error processing employee ' ||
                               emp_tab(i).emp_id || ': ' ||
                               SQLERRM);
      END;
    END LOOP;

    DBMS_OUTPUT.PUT_LINE('---------------------------------------' || CHR(10));
  END LOOP;

  -- Close cursor
  CLOSE emp_cur;

  -- Final status report
  DBMS_OUTPUT.PUT_LINE('Processing complete.');
  IF v_err_count > 0 THEN
    DBMS_OUTPUT.PUT_LINE('Errors encountered: ' || v_err_count);
  END IF;

EXCEPTION
  WHEN OTHERS THEN
    IF emp_cur%ISOPEN THEN
      CLOSE emp_cur;
    END IF;
    RAISE;
END;
```

Key Points about OPEN FOR with Collections:

1. Benefits:
    - Efficient bulk processing of data
    - Better performance with BULK COLLECT
    - Flexible dynamic SQL usage
2. Best Practices:
    - Always close cursors
    - Use LIMIT clause with BULK COLLECT for large datasets
    - Implement proper error handling
    - Process data in batches when dealing with large results
3. Important Considerations:
    - Memory usage with large collections
    - Proper cursor management
    - Error handling at multiple levels
    - Transaction management when needed

FETCH and CLOSE Operations with Collections

This section demonstrates how to properly handle FETCH and CLOSE operations when working with collections and cursors.

Basic Example with FETCH and CLOSE

```
DECLARE
  -- Cursor and collection types
  TYPE emp_cur_type IS REF CURSOR;
  TYPE emp_rec_type IS RECORD (
    emp_id    employees.employee_id%TYPE,
    emp_name  employees.last_name%TYPE,
    dept_id   employees.department_id%TYPE,
    salary    employees.salary%TYPE
  );
  TYPE emp_tab_type IS TABLE OF emp_rec_type
    INDEX BY PLS_INTEGER;

  -- Variables
  emp_cur     emp_cur_type;
  emp_tab     emp_tab_type;
  v_sql       VARCHAR2(1000);
  v_fetch_count PLS_INTEGER := 0;
  v_total_rows  PLS_INTEGER := 0;
BEGIN
  -- Prepare dynamic query
  v_sql := 'SELECT e.employee_id, e.last_name,
              e.department_id, e.salary
        FROM employees e
        WHERE e.department_id = :1
        ORDER BY e.salary DESC';

  -- Open cursor
  OPEN emp_cur FOR v_sql USING 60; -- IT Department

  -- Fetch in batches of 5 rows
  LOOP
    -- Clear collection before each fetch
    emp_tab.DELETE;

    -- Fetch batch of records
    FETCH emp_cur
    BULK COLLECT INTO emp_tab LIMIT 5;

    -- Process fetched batch
    v_fetch_count := emp_tab.COUNT;
    v_total_rows := v_total_rows + v_fetch_count;

    -- Display batch information
    DBMS_OUTPUT.PUT_LINE('Processing batch of ' ||
                  v_fetch_count || ' records:');
    DBMS_OUTPUT.
```

```
PUT_LINE('--------------------');

        -- Process records in current batch
        FOR i IN 1.v_fetch_count LOOP
            DBMS_OUTPUT.PUT_LINE(
                'Employee ' || emp_tab(i).emp_id ||
                ' - ' || emp_tab(i).emp_name ||
                ' (Dept ' || emp_tab(i).dept_id ||
                ') Salary: $' || emp_tab(i).salary
            );
        END LOOP;

        DBMS_OUTPUT.PUT_LINE('--------------------' || CHR(10));

        -- Exit when no more rows
        EXIT WHEN v_fetch_count = 0;
    END LOOP;

    -- Always close cursor
    CLOSE emp_cur;

    -- Display final count
    DBMS_OUTPUT.PUT_LINE('Total rows processed: ' || v_total_rows);

EXCEPTION
    WHEN OTHERS THEN
        -- Ensure cursor is closed in case of errors
        IF emp_cur%ISOPEN THEN
            CLOSE emp_cur;
        END IF;
        RAISE;
END;
```

Advanced Example with Multiple Cursors and Error Handling

```
DECLARE
   -- Cursor types
   TYPE emp_cur_type IS REF CURSOR;

   -- Record types
   TYPE emp_rec_type IS RECORD (
      emp_id    employees.employee_id%TYPE,
      emp_name  employees.last_name%TYPE,
      salary    employees.
```

```
salary%TYPE
    );

    -- Collection types
    TYPE emp_tab_type IS TABLE OF emp_rec_type
        INDEX BY PLS_INTEGER;

    -- Cursor variables
    emp_cur_it      emp_cur_type;  -- IT Department
    emp_cur_sales   emp_cur_type;  -- Sales Department

    -- Collection variables
    emp_tab_it      emp_tab_type;
    emp_tab_sales   emp_tab_type;

    -- Other variables
    v_sql           VARCHAR2(1000);
    v_batch_size    PLS_INTEGER := 5;
    v_processed_it  PLS_INTEGER := 0;
    v_processed_sales PLS_INTEGER := 0;

    -- Custom exception
    cursor_error EXCEPTION;
BEGIN
    -- Prepare SQL statements
    v_sql := 'SELECT employee_id, last_name, salary
              FROM employees
              WHERE department_id = :1
              ORDER BY salary DESC';

    -- Open cursors for different departments
    OPEN emp_cur_it FOR v_sql USING 60;     -- IT
    OPEN emp_cur_sales FOR v_sql USING 80;  -- Sales

    -- Process both departments simultaneously
    LOOP
        -- Fetch IT department employees
        FETCH emp_cur_it
        BULK COLLECT INTO emp_tab_it LIMIT v_batch_size;

        -- Fetch Sales department employees
        FETCH emp_cur_sales
        BULK COLLECT INTO emp_tab_sales LIMIT v_batch_size;

        -- Exit if both fetches return no rows
        EXIT WHEN emp_tab_it.COUNT = 0 AND emp_tab_sales.
```

```
COUNT = 0;

    -- Process IT department records
    IF emp_tab_it.COUNT > 0 THEN
      DBMS_OUTPUT.PUT_LINE('IT Department Batch:');
      DBMS_OUTPUT.PUT_LINE('-------------------');
      FOR i IN 1.emp_tab_it.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(
          'IT: ' || emp_tab_it(i).emp_name ||
          ' (ID: ' || emp_tab_it(i).emp_id ||
          ') Salary: $' || emp_tab_it(i).salary
        );
        v_processed_it := v_processed_it + 1;
      END LOOP;
      DBMS_OUTPUT.PUT_LINE(CHR(10));
    END IF;

    -- Process Sales department records
    IF emp_tab_sales.COUNT > 0 THEN
      DBMS_OUTPUT.PUT_LINE('Sales Department Batch:');
      DBMS_OUTPUT.PUT_LINE('----------------------');
      FOR i IN 1.emp_tab_sales.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(
          'Sales: ' || emp_tab_sales(i).emp_name ||
          ' (ID: ' || emp_tab_sales(i).emp_id ||
          ') Salary: $' || emp_tab_sales(i).salary
        );
        v_processed_sales := v_processed_sales + 1;
      END LOOP;
      DBMS_OUTPUT.PUT_LINE(CHR(10));
    END IF;
  END LOOP;

  -- Close cursors
  IF emp_cur_it%ISOPEN THEN
    CLOSE emp_cur_it;
  END IF;

  IF emp_cur_sales%ISOPEN THEN
    CLOSE emp_cur_sales;
  END IF;

  -- Display summary
  DBMS_OUTPUT.PUT_LINE('Processing Complete:');
  DBMS_OUTPUT.PUT_LINE('IT Department: ' || v_processed_it || ' records');
  DBMS_OUTPUT.PUT_LINE('Sales Department: ' || v_processed_sales || ' records');
  DBMS_OUTPUT.
```

```
PUT_LINE('Total: ' || (v_processed_it + v_processed_sales) || ' records');

EXCEPTION
   WHEN cursor_error THEN
      DBMS_OUTPUT.PUT_LINE('Error processing cursors');
      -- Ensure all cursors are closed
      IF emp_cur_it%ISOPEN THEN
         CLOSE emp_cur_it;
      END IF;
      IF emp_cur_sales%ISOPEN THEN
         CLOSE emp_cur_sales;
      END IF;
      RAISE;

   WHEN OTHERS THEN
      -- Close any open cursors
      IF emp_cur_it%ISOPEN THEN
         CLOSE emp_cur_it;
      END IF;
      IF emp_cur_sales%ISOPEN THEN
         CLOSE emp_cur_sales;
      END IF;
      RAISE;
END;
```

Key Points about FETCH and CLOSE Operations:

1. Best Practices:
    - Always close cursors in both normal and exception paths
    - Use BULK COLLECT with LIMIT for large datasets
    - Clear collections before reuse
    - Maintain proper error handling
2. Important Considerations:
    - Cursor state management
    - Memory usage with collections
    - Batch processing for large datasets
    - Exception handling at multiple levels

Dynamic SQL with Collection Parameters

This section demonstrates how to effectively use collections as parameters in dynamic SQL statements, with focus on different binding techniques and best practices.

Basic Example - Using Collections as IN-List Parameters

```
DECLARE
   -- Define collection types
   TYPE number_table_type IS TABLE OF NUMBER
      INDEX BY PLS_INTEGER;
   TYPE varchar_table_type IS TABLE OF VARCHAR2(100)
      INDEX BY PLS_INTEGER;

   -- Declare collections
   v_dept_ids    number_table_type;
   v_emp_names   varchar_table_type;

   -- Other variables
   v_sql         VARCHAR2(4000);
   v_in_list     VARCHAR2(1000);
   v_count       NUMBER;
BEGIN
   -- Populate department IDs collection
   v_dept_ids(1) := 60;  -- IT
   v_dept_ids(2) := 90;  -- Executive
   v_dept_ids(3) := 100; -- Finance

   -- Create IN-list from collection
   v_in_list := '';
   FOR i IN v_dept_ids.FIRST.v_dept_ids.LAST LOOP
      IF i > v_dept_ids.FIRST THEN
         v_in_list := v_in_list || ',';
      END IF;
      v_in_list := v_in_list || v_dept_ids(i);
   END LOOP;

   -- Build and execute dynamic query
   v_sql := 'SELECT last_name
            FROM employees
            WHERE department_id IN (' || v_in_list || ')
            ORDER BY last_name';

   -- Store results in collection
   EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM (' || v_sql || ')'
   INTO v_count;

   DBMS_OUTPUT.PUT_LINE('Found ' || v_count || ' employees');
   DBMS_OUTPUT.
```

```
PUT_LINE('In departments: ' || v_in_list);

   -- Fetch employee names using BULK COLLECT
   EXECUTE IMMEDIATE v_sql
   BULK COLLECT INTO v_emp_names;

   -- Display results
   FOR i IN v_emp_names.FIRST.v_emp_names.LAST LOOP
      DBMS_OUTPUT.PUT_LINE(i || '. ' || v_emp_names(i));
   END LOOP;
END;
```

Advanced Example - Multiple Collection Parameters and Complex Operations

```
DECLARE
   -- Collection types
   TYPE number_table_type IS TABLE OF NUMBER
      INDEX BY PLS_INTEGER;
   TYPE varchar_table_type IS TABLE OF VARCHAR2(100)
      INDEX BY PLS_INTEGER;
   TYPE salary_table_type IS TABLE OF employees.salary%TYPE
      INDEX BY PLS_INTEGER;

   -- Declare collections
   v_emp_ids     number_table_type;
   v_dept_ids    number_table_type;
   v_emp_names   varchar_table_type;
   v_salaries    salary_table_type;

   -- Cursor type
   TYPE ref_cursor_type IS REF CURSOR;
   v_result_set  ref_cursor_type;

   -- Other variables
   v_sql         VARCHAR2(4000);
   v_where_clause VARCHAR2(1000);
   v_params_clause VARCHAR2(1000);
   v_dept_list   VARCHAR2(500);
   v_min_salary  NUMBER := 5000;
   v_count       NUMBER := 0;

   -- For error handling
   v_error_occurred BOOLEAN := FALSE;
BEGIN
   -- Initialize department list
   v_dept_ids(1) := 60;  -- IT
   v_dept_ids(2) := 90;  -- Executive
   v_dept_ids(3) := 100; -- Finance

   -- Create department IN-list
   FOR i IN v_dept_ids.FIRST.v_dept_ids.LAST LOOP
      IF i > v_dept_ids.
```

```
FIRST THEN
            v_dept_list := v_dept_list || ',';
        END IF;
        v_dept_list := v_dept_list || v_dept_ids(i);
    END LOOP;

    -- Build dynamic query with multiple conditions
    v_sql := 'SELECT employee_id, last_name, salary, department_id
              FROM employees
              WHERE department_id IN (' || v_dept_list || ')
              AND salary >= :1
              ORDER BY department_id, salary DESC';

    -- Open cursor with parameter
    OPEN v_result_set FOR v_sql USING v_min_salary;

    -- Fetch and process results
    LOOP
        -- Fetch employee data into collections
        FETCH v_result_set
        BULK COLLECT INTO v_emp_ids, v_emp_names, v_salaries, v_dept_ids
        LIMIT 10;

        EXIT WHEN v_emp_ids.COUNT = 0;

        -- Process each batch
        FOR i IN v_emp_ids.FIRST.v_emp_ids.LAST LOOP
            v_count := v_count + 1;
            BEGIN
                -- Build dynamic update statement
                v_sql := 'UPDATE employees
                          SET salary = :1
                          WHERE employee_id = :2
                          RETURNING last_name INTO :3';

                -- Perform update with salary increase
                EXECUTE IMMEDIATE v_sql
                USING (v_salaries(i) * 1.05),
                      v_emp_ids(i)
                RETURNING INTO v_emp_names(i);

                -- Display update information
                DBMS_OUTPUT.PUT_LINE(
                    'Updated: ' || v_emp_names(i) ||
                    ' (ID: ' || v_emp_ids(i) ||
                    ', Dept: ' || v_dept_ids(i) ||
                    ') New Salary: $' || (v_salaries(i) * 1.
```

```
05)
            );

        EXCEPTION
          WHEN OTHERS THEN
            v_error_occurred := TRUE;
            DBMS_OUTPUT.PUT_LINE(
              'Error updating employee ' || v_emp_ids(i) ||
              ': ' || SQLERRM
            );
        END;
    END LOOP;

    -- Commit after each batch
    IF NOT v_error_occurred THEN
      COMMIT;
      DBMS_OUTPUT.PUT_LINE('Batch committed successfully.');
    ELSE
      ROLLBACK;
      DBMS_OUTPUT.PUT_LINE('Batch rolled back due to errors.');
    END IF;
  END LOOP;

  -- Close cursor
  CLOSE v_result_set;

  -- Final summary
  DBMS_OUTPUT.PUT_LINE(CHR(10) || 'Processing Complete:');
  DBMS_OUTPUT.PUT_LINE('Total records processed: ' || v_count);

EXCEPTION
  WHEN OTHERS THEN
    IF v_result_set%ISOPEN THEN
      CLOSE v_result_set;
    END IF;
    ROLLBACK;
    RAISE;
END;
```

Key Points for Dynamic SQL with Collection Parameters:

1. Best Practices:
    - Always validate collection contents before use
    - Use bind variables when possible
    - Handle large collections in batches
    - Implement proper error handling
    - Manage transactions appropriately
2. Performance Considerations:
    - Use BULK COLLECT for efficient data retrieval
    - Limit batch sizes for large datasets
    - Consider memory usage with large collections
    - Use proper indexing strategies
3. Security Considerations:
    - Avoid SQL injection by properly handling user input
    - Validate collection contents before using in dynamic SQL
    - Use bind variables instead of direct string concatenation
    - Implement proper access controls
4. Error Handling:
    - Handle exceptions at appropriate levels
    - Implement proper rollback mechanisms
    - Log errors appropriately
    - Maintain data consistency

Let me clarify the key differences between regular collection usage and specifically using collections as parameters in dynamic SQL.

In previous examples, we were mainly:

1. Storing data in collections
2. Processing collections element by element
3. Using individual elements as bind variables

Let me show you the difference with simplified examples:

Previous Approach (Individual Elements)

```
DECLARE
   TYPE emp_id_tab_type IS TABLE OF employees.employee_id%TYPE
      INDEX BY PLS_INTEGER;
   emp_ids    emp_id_tab_type;
   v_sql      VARCHAR2(1000);
BEGIN
   -- Populate collection
   emp_ids(1) := 100;
   emp_ids(2) := 101;

   -- Process ONE element at a time
   FOR i IN emp_ids.FIRST..emp_ids.LAST LOOP
      v_sql := 'BEGIN
                  DBMS_OUTPUT.PUT_LINE(:1);
                END;';

      EXECUTE IMMEDIATE v_sql
      USING emp_ids(i);  -- Binding single element
   END LOOP;
END;
/
```

Collection Parameters Approach

```
DECLARE
   TYPE emp_id_tab_type IS TABLE OF employees.employee_id%TYPE
      INDEX BY PLS_INTEGER;
   TYPE name_tab_type IS TABLE OF employees.last_name%TYPE
      INDEX BY PLS_INTEGER;

   emp_ids    emp_id_tab_type;
   emp_names  name_tab_type;
   v_sql      VARCHAR2(1000);
   v_in_list  VARCHAR2(1000);
BEGIN
   -- Populate collection
   emp_ids(1) := 100;
   emp_ids(2) := 101;

   -- Create parameter list from ENTIRE collection
   v_in_list := '';
   FOR i IN emp_ids.FIRST..emp_ids.LAST LOOP
      IF i > emp_ids.FIRST THEN
         v_in_list := v_in_list || ',';
      END IF;
      v_in_list := v_in_list || emp_ids(i);
   END LOOP;

   -- Use ENTIRE collection as a parameter
   v_sql := 'SELECT last_name
            FROM employees
            WHERE employee_id IN (' || v_in_list || ')';

   -- Execute and store results
   EXECUTE IMMEDIATE v_sql
   BULK COLLECT INTO emp_names;

   -- Display results
   FOR i IN emp_names.FIRST..emp_names.LAST LOOP
      DBMS_OUTPUT.PUT_LINE('Employee ' || emp_ids(i) || ': ' || emp_names(i));
   END LOOP;
END;
```

The key differences:

1. Parameter Usage:
    - Previous: Uses collections as data storage and processes one element at a time
    - Collection Parameters: Uses entire collection as a parameter in the SQL statement
2. Execution Efficiency:
    - Previous: Multiple EXECUTE IMMEDIATE calls (one per element)
    - Collection Parameters: Single EXECUTE IMMEDIATE for all elements
3. SQL Construction:
    - Previous: Simple bind variables
    - Collection Parameters: Constructs IN-lists or complex conditions using collection data

Here's a more practical example showing the difference:

```
DECLARE
   -- Collection type and variable
   TYPE dept_id_tab_type IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
   dept_ids    dept_id_tab_type;

   -- Variables for processing
   v_sql       VARCHAR2(1000);
   v_count     NUMBER;
   v_in_list   VARCHAR2(100);
   TYPE refcur_type IS REF CURSOR;
   v_refcur    refcur_type;
   v_dept_id   NUMBER;
   v_emp_count NUMBER;
BEGIN
   -- Populate collection
   dept_ids(1) := 60;  -- IT
   dept_ids(2) := 90;  -- Executive
   dept_ids(3) := 100; -- Finance

   -- Previous Approach (Multiple Executions)
   DBMS_OUTPUT.PUT_LINE('Previous Approach:');
   FOR i IN dept_ids.FIRST.dept_ids.LAST LOOP
      v_sql := 'SELECT COUNT(*) FROM employees WHERE department_id = :1';
      EXECUTE IMMEDIATE v_sql
      INTO v_count
      USING dept_ids(i);

      DBMS_OUTPUT.PUT_LINE('Dept ' || dept_ids(i) || ': ' || v_count);
   END LOOP;

   -- Collection Parameters Approach (Single Execution)
   DBMS_OUTPUT.PUT_LINE(CHR(10) || 'Collection Parameters Approach:');

   -- Build IN list from collection
   v_in_list := '';
   FOR i IN dept_ids.FIRST.dept_ids.LAST LOOP
      IF i > dept_ids.
```

```
FIRST THEN
            v_in_list := v_in_list || ',';
        END IF;
        v_in_list := v_in_list || dept_ids(i);
    END LOOP;

    -- Create dynamic query
    v_sql := 'SELECT department_id, COUNT(*) as emp_count
              FROM employees
              WHERE department_id IN (' || v_in_list || ')
              GROUP BY department_id
              ORDER BY department_id';

    -- Execute and fetch results
    OPEN v_refcur FOR v_sql;
    LOOP
        FETCH v_refcur INTO v_dept_id, v_emp_count;
        EXIT WHEN v_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Dept ' || v_dept_id || ': ' || v_emp_count);
    END LOOP;
    CLOSE v_refcur;
END;
```

The output should look something like:

```
Previous Approach:
Dept 60: 5
Dept 90: 3
Dept 100: 6

Collection Parameters Approach:
Dept 60: 5
Dept 90: 3
Dept 100: 6
```

The Collection Parameters approach is particularly useful when:

1. You need to process multiple values in a single SQL statement
2. You want to reduce the number of context switches between PL/SQL and SQL
3. You're working with IN clauses or bulk operations
4. You need to perform set-based operations rather than row-by-row processing

Bulk Operations in PL/SQL