13 PL/SQL Optimization and Tuning

This chapter explains how the PL/SQL compiler optimizes your code and how to write efficient PL/SQL code and improve existing PL/SQL code.

Topics

> **See Also:**
>
> *Oracle Database Development Guide* for disadvantages of cursor variables

PL/SQL Optimizer

Prior to Oracle Database 10g release 1, the PL/SQL compiler translated your source text to system code without applying many changes to improve performance. Now, PL/SQL uses an optimizer that can rearrange code for better performance.

The optimizer is enabled by default. In rare cases, if the overhead of the optimizer makes compilation of very large applications too slow, you can lower the optimization by setting the compilation parameter `PLSQL_OPTIMIZE_LEVEL=1` instead of its default value 2. In even rarer cases, PL/SQL might raise an exception earlier than expected or not at all. Setting `PLSQL_OPTIMIZE_LEVEL=1` prevents the code from being rearranged.

See Also:

- *Oracle Database Reference* for information about the `PLSQL_OPTIMIZE_LEVEL` compilation parameter
- *Oracle Database Development Guide* for examples of changing the `PLSQL_OPTIMIZE_LEVEL` compilation parameter
- *Oracle Database Reference* for information about the static dictionary view `ALL_PLSQL_OBJECT_SETTINGS`

Subprogram Inlining

One optimization that the compiler can perform is **subprogram inlining**.

Subprogram inlining replaces a subprogram invocation with a copy of the invoked subprogram (if the invoked and invoking subprograms are in the same program unit). To allow subprogram inlining, either accept the default value of the `PLSQL_OPTIMIZE_LEVEL` compilation parameter (which is 2) or set it to 3.

With `PLSQL_OPTIMIZE_LEVEL=2`, you must specify each subprogram to be inlined with the `INLINE` pragma:

```
PRAGMA INLINE (subprogram, 'YES')
```

If *`subprogram`* is overloaded, then the preceding pragma applies to every subprogram with that name.

With `PLSQL_OPTIMIZE_LEVEL=3`, the PL/SQL compiler seeks opportunities to inline subprograms. You need not specify subprograms to be inlined. However, you can use the `INLINE` pragma (with the preceding syntax) to give a subprogram a high priority for inlining, and then the compiler inlines it unless other considerations or limits make the inlining undesirable.

If a particular subprogram is inlined, performance almost always improves. However, because the compiler inlines subprograms early in the optimization process, it is possible for subprogram inlining to preclude later, more powerful optimizations.

If subprogram inlining slows the performance of a particular PL/SQL program, then use the PL/SQL hierarchical profiler to identify subprograms for which you want to turn off inlining. To turn off inlining for a subprogram, use the `INLINE` pragma:

```
PRAGMA INLINE (subprogram, 'NO')
```

The `INLINE` pragma affects only the immediately following declaration or statement, and only some kinds of statements.

When the `INLINE` pragma immediately precedes a declaration, it affects:

- Every invocation of the specified subprogram in that declaration
- Every initialization value in that declaration except the default initialization values of records

When the `INLINE` pragma immediately precedes one of these statements, the pragma affects every invocation of the specified subprogram in that statement:

- Assignment

- `CALL`
- Conditional
- `CASE`
- `CONTINUE WHEN`
- `EXECUTE IMMEDIATE`
- `EXIT WHEN`
- `LOOP`
- `RETURN`

The `INLINE` pragma does not affect statements that are not in the preceding list.

Multiple pragmas can affect the same declaration or statement. Each pragma applies its own effect to the statement. If `PRAGMA INLINE(`*`subprogram`*`,'YES')` and `PRAGMA INLINE(`*`identifier`*`,'NO')` have the same *`subprogram`*, then `'NO'` overrides `'YES'`. One `PRAGMA INLINE(`*`subprogram`*`,'NO')` overrides any number of occurrences of `PRAGMA INLINE(`*`subprogram`*`,'YES')`, and the order of these pragmas is not important.

See Also:

- *Oracle Database Development Guide* for more information about PL/SQL hierarchical profiler
- *Oracle Database Reference* for information about the `PLSQL_OPTIMIZE_LEVEL` compilation parameter
- *Oracle Database Reference* for information about the static dictionary view `ALL_PLSQL_OBJECT_SETTINGS`

Example 13-1 Specifying that Subprogram Is To Be Inlined

In this example, if `PLSQL_OPTIMIZE_LEVEL=2`, the `INLINE` pragma affects the procedure invocations `p1(1)` and `p1(2)`, but not the procedure invocations `p1(3)` and `p1(4)`.

```
PROCEDURE p1 (x PLS_INTEGER) IS ...
...
PRAGMA INLINE (p1, 'YES');
x:= p1(1) + p1(2) + 17;    -- These 2 invocations to p1 are inlined
...
x:= p1(3) + p1(4) + 17;    -- These 2 invocations to p1 are not inlined
...
```

Example 13-2 Specifying that Overloaded Subprogram Is To Be Inlined

In this example, if `PLSQL_OPTIMIZE_LEVEL=2`, the `INLINE` pragma affects both functions named `p2`.

```
FUNCTION p2 (p boolean) return PLS_INTEGER IS ...
FUNCTION p2 (x PLS_INTEGER) return PLS_INTEGER IS ...
...
PRAGMA INLINE(p2, 'YES');
x := p2(true) + p2(3);

...
```

Example 13-3 Specifying that Subprogram Is Not To Be Inlined

In this example, the `INLINE` pragma affects the procedure invocations `p1(1)` and `p1(2)`, but not the procedure invocations `p1(3)` and `p1(4)`.

```
PROCEDURE p1 (x PLS_INTEGER) IS ...
...
PRAGMA INLINE (p1, 'NO');
x:= p1(1) + p1(2) + 17;    -- These 2 invocations to p1 are not inlined
...
x:= p1(3) + p1(4) + 17;    -- These 2 invocations to p1 might be inlined
...
```

Example 13-4 PRAGMA INLINE ... 'NO' Overrides PRAGMA INLINE ... 'YES'

In this example, the second `INLINE` pragma overrides both the first and third `INLINE` pragmas.

```
PROCEDURE p1 (x PLS_INTEGER) IS ...
...
PRAGMA INLINE (p1, 'YES');
PRAGMA INLINE (p1, 'NO');
PRAGMA INLINE (p1, 'YES');
x:= p1(1) + p1(2) + 17;    -- These 2 invocations to p1 are not inlined
...
```

Candidates for Tuning

The following kinds of PL/SQL code are very likely to benefit from tuning:

- Older code that does not take advantage of new PL/SQL language features.

  > **Tip:**
  >
  > Before tuning older code, benchmark the current system and profile the older subprograms that your program invokes (see "Profiling and Tracing PL/SQL Programs"). With the many automatic optimizations of the PL/SQL optimizer (described in "PL/SQL Optimizer"), you might see performance improvements before doing any tuning.

- Older dynamic SQL statements written with the `DBMS_SQL` package.

  If you know at compile time the number and data types of the input and output variables of a dynamic SQL statement, then you can rewrite the statement in native dynamic SQL, which runs noticeably faster than equivalent code that uses the `DBMS_SQL` package (especially when it can be optimized by the compiler). For more information, see PL/SQL Dynamic SQL.

- Code that spends much time processing SQL statements.

  See "Tune SQL Statements".

- Functions invoked in queries, which might run millions of times.

  See "Tune Function Invocations in Queries".

- Code that spends much time looping through query results.

  See "Tune Loops".

- Code that does many numeric computations.

See "Tune Computation-Intensive PL/SQL Code".

- Code that spends much time processing PL/SQL statements (as opposed to issuing database definition language (DDL) statements that PL/SQL passes directly to SQL).

  See "Compiling PL/SQL Units for Native Execution".

Minimizing CPU Overhead

Topics

Tune SQL Statements

The most common cause of slowness in PL/SQL programs is slow SQL statements. To make SQL statements in a PL/SQL program as efficient as possible:

- Use appropriate indexes.

  For details, see Oracle Database Performance Tuning Guide.
- Use query hints to avoid unnecessary full-table scans.

  For details, see *Oracle Database SQL Language Reference*.
- Collect current statistics on all tables, using the subprograms in the `DBMS_STATS` package.

  For details, see *Oracle Database Performance Tuning Guide*.
- Analyze the execution plans and performance of the SQL statements, using:
  - `EXPLAIN PLAN` statement

    For details, see *Oracle Database Performance Tuning Guide*.
  - SQL Trace facility with `TKPROF` utility

    For details, see *Oracle Database Performance Tuning Guide*.
- Use bulk SQL, a set of PL/SQL features that minimizes the performance overhead of the communication between PL/SQL and SQL.

  For details, see "Bulk SQL and Bulk Binding".

Tune Function Invocations in Queries

Functions invoked in queries might run millions of times. Do not invoke a function in a query unnecessarily, and make the invocation as efficient as possible.

Create a function-based index on the table in the query. The `CREATE INDEX` statement might take a while, but the query can run much faster because the function value for each row is cached.

If the query passes a column to a function, then the query cannot use user-created indexes on that column, so the query might invoke the function for every row of the table (which might be very large). To minimize the number of function invocations, use a nested query. Have the inner query filter the result set to a small number of rows, and have the outer query invoke the function for only those rows.

See Also:

- *Oracle Database SQL Language Reference* for more information about `CREATE INDEX` statement syntax
- "PL/SQL Function Result Cache" for information about caching the results of PL/SQL functions

Example 13-5 Nested Query Improves Performance

In this example, the two queries produce the same result set, but the second query is more efficient than the first. (In the example, the times and time difference are very small, because the `EMPLOYEES` table is very small. For a very large table, they would be significant.)

```
DECLARE
  starting_time  TIMESTAMP WITH TIME ZONE;
  ending_time    TIMESTAMP WITH TIME ZONE;
BEGIN
  -- Invokes SQRT for every row of employees table:

  SELECT SYSTIMESTAMP INTO starting_time FROM DUAL;

  FOR item IN (
    SELECT DISTINCT(SQRT(department_id)) col_alias
    FROM employees
    ORDER BY col_alias
  )
  LOOP
    DBMS_OUTPUT.PUT_LINE('Square root of dept. ID = ' || item.col_alias);
  END LOOP;

  SELECT SYSTIMESTAMP INTO ending_time FROM DUAL;

  DBMS_OUTPUT.PUT_LINE('Time = ' || TO_CHAR(ending_time - starting_time));

  -- Invokes SQRT for every distinct department_id of employees table:

  SELECT SYSTIMESTAMP INTO starting_time FROM DUAL;

  FOR item IN (
    SELECT SQRT(department_id) col_alias
    FROM (SELECT DISTINCT department_id FROM employees)
    ORDER BY col_alias
  )
  LOOP
    IF item.col_alias IS NOT NULL THEN
      DBMS_OUTPUT.PUT_LINE('Square root of dept. ID = ' || item.col_alias);
    END IF;
  END LOOP;

  SELECT SYSTIMESTAMP INTO ending_time FROM DUAL;
```

```
  DBMS_OUTPUT.PUT_LINE('Time = ' || TO_CHAR(ending_time - starting_time));
END;
/
```

Result is similar to:

```
Square root of dept. ID = 3.16227766016837933199889354443271853372
Square root of dept. ID = 4.47213595499957939281834733746255247088
Square root of dept. ID = 5.47722557505166113456969782800802133953
Square root of dept. ID = 6.32455532033675866399778708886543706744
Square root of dept. ID = 7.07106781186547524400844362104849039285
Square root of dept. ID = 7.74596669241483377035853079956479922167
Square root of dept. ID = 8.36660026534075547978172025785187489393
Square root of dept. ID = 8.94427190999915878563669467492510494176
Square root of dept. ID = 9.48683298050513799599668063329815560116
Square root of dept. ID = 10
Square root of dept. ID = 10.48808848170151546991453513679937598475
Time = +000000000 00:00:00.046000000
Square root of dept. ID = 3.16227766016837933199889354443271853372
Square root of dept. ID = 4.47213595499957939281834733746255247088
Square root of dept. ID = 5.47722557505166113456969782800802133953
Square root of dept. ID = 6.32455532033675866399778708886543706744
Square root of dept. ID = 7.07106781186547524400844362104849039285
Square root of dept. ID = 7.74596669241483377035853079956479922167
Square root of dept. ID = 8.36660026534075547978172025785187489393
Square root of dept. ID = 8.94427190999915878563669467492510494176
Square root of dept. ID = 9.48683298050513799599668063329815560116
Square root of dept. ID = 10
Square root of dept. ID = 10.48808848170151546991453513679937598475
Time = +000000000 00:00:00.000000000
```

Tune Subprogram Invocations

If a subprogram has `OUT` or `IN OUT` parameters, you can sometimes decrease its invocation overhead by declaring those parameters with the `NOCOPY` hint.

When `OUT` or `IN OUT` parameters represent large data structures such as collections, records, and instances of ADTs, copying them slows execution and increases memory use—especially for an instance of an ADT.

For each invocation of an ADT method, PL/SQL copies every attribute of the ADT. If the method is exited normally, then PL/SQL applies any changes that the method made to the attributes. If the method is exited with an unhandled exception, then PL/SQL does not change the attributes.

If your program does not require that an `OUT` or `IN OUT` parameter retain its pre-invocation value if the subprogram ends with an unhandled exception, then include the `NOCOPY` hint in the parameter declaration. The `NOCOPY` hint requests (but does not ensure) that the compiler pass the corresponding actual parameter by reference instead of value.

> **Caution:**
>
> Do not rely on `NOCOPY` (which the compiler might or might not obey for a particular invocation) to ensure that an actual parameter or ADT attribute retains its pre-invocation value if the subprogram is exited with an unhandled exception. Instead, ensure that the subprogram handle all exceptions.

See Also:

- "NOCOPY" for more information about `NOCOPY` hint
- *Oracle Database Object-Relational Developer's Guide* for information about using `NOCOPY` with member methods of ADTs

Example 13-6 NOCOPY Subprogram Parameters

In this example, if the compiler obeys the `NOCOPY` hint for the invocation of `do_nothing2`, then the invocation of `do_nothing2` is faster than the invocation of `do_nothing1`.

```
DECLARE
  TYPE EmpTabTyp IS TABLE OF employees%ROWTYPE;
  emp_tab EmpTabTyp := EmpTabTyp(NULL);  -- initialize
  t1 NUMBER;
  t2 NUMBER;
  t3 NUMBER;

  PROCEDURE get_time (t OUT NUMBER) IS
  BEGIN
    t := DBMS_UTILITY.get_time;
  END;

  PROCEDURE do_nothing1 (tab IN OUT EmpTabTyp) IS
  BEGIN
    NULL;
  END;

  PROCEDURE do_nothing2 (tab IN OUT NOCOPY EmpTabTyp) IS
  BEGIN
    NULL;
  END;

BEGIN
  SELECT * INTO emp_tab(1)
  FROM employees
  WHERE employee_id = 100;

  emp_tab.EXTEND(49999, 1);  -- Copy element 1 into 2..50000
  get_time(t1);
  do_nothing1(emp_tab);  -- Pass IN OUT parameter
  get_time(t2);
  do_nothing2(emp_tab);  -- Pass IN OUT NOCOPY parameter
  get_time(t3);
  DBMS_OUTPUT.PUT_LINE ('Call Duration (secs)');
  DBMS_OUTPUT.PUT_LINE ('--------------------');
  DBMS_OUTPUT.PUT_LINE ('Just IN OUT: ' || TO_CHAR((t2 - t1)/100.0));
  DBMS_OUTPUT.PUT_LINE ('With NOCOPY: ' || TO_CHAR((t3 - t2))/100.0);
END;
/
```

Tune Loops

Because PL/SQL applications are often built around loops, it is important to optimize both the loops themselves and the code inside them.

If you must loop through a result set more than once, or issue other queries as you loop through a result set, you might be able to change the original query to give you exactly the results you want. Explore the SQL set operators that let you combine multiple queries, described in *Oracle Database SQL Language Reference*.

You can also use subqueries to do the filtering and sorting in multiple stages—see "Processing Query Result Sets with Subqueries".

See Also:

"Bulk SQL and Bulk Binding"

Tune Computation-Intensive PL/SQL Code

These recommendations apply especially (but not only) to computation-intensive PL/SQL code.

Topics

- Use Data Types that Use Hardware Arithmetic
- Avoid Constrained Subtypes in Performance-Critical Code
- Minimize Implicit Data Type Conversion

Use Data Types that Use Hardware Arithmetic

Avoid using data types in the `NUMBER` data type family (described in "NUMBER Data Type Family"). These data types are represented internally in a format designed for portability and arbitrary scale and precision, not for performance. Operations on data of these types use library arithmetic, while operations on data of the types `PLS_INTEGER`, `BINARY_FLOAT` and `BINARY_DOUBLE` use hardware arithmetic.

For local integer variables, use `PLS_INTEGER`, described in "PLS_INTEGER and BINARY_INTEGER Data Types". For variables used in performance-critical code, that can never have the value `NULL`, and do not need overflow checking, use `SIMPLE_INTEGER`, described in "SIMPLE_INTEGER Subtype of PLS_INTEGER".

For floating-point variables, use `BINARY_FLOAT` or `BINARY_DOUBLE`, described in *Oracle Database SQL Language Reference*. For variables used in performance-critical code, that can never have the value `NULL`, and that do not need overflow checking, use `SIMPLE_FLOAT` or `SIMPLE_DOUBLE`, explained in "Additional PL/SQL Subtypes of BINARY_FLOAT and BINARY_DOUBLE".

Note:

`BINARY_FLOAT` and `BINARY_DOUBLE` and their subtypes are less suitable for financial code where accuracy is critical, because they do not always represent fractional values precisely, and handle rounding differently than the `NUMBER` types.

Many SQL numeric functions (described in *Oracle Database SQL Language Reference*) are overloaded with versions that accept `BINARY_FLOAT` and `BINARY_DOUBLE` parameters. You can speed up computation-intensive code by passing variables of these data types to such functions, and by invoking the conversion functions `TO_BINARY_FLOAT` (described in *Oracle Database SQL Language Reference*) and `TO_BINARY_DOUBLE` (described in *Oracle Database SQL Language Reference*) when passing expressions to such functions.

Avoid Constrained Subtypes in Performance-Critical Code

In performance-critical code, avoid constrained subtypes (described in "Constrained Subtypes"). Each assignment to a variable or parameter of a constrained subtype requires extra checking at run time to ensure that the value to be assigned does not violate the constraint.

> **See Also:**
>
> PL/SQL Predefined Data Types includes predefined constrained subtypes

Minimize Implicit Data Type Conversion

At run time, PL/SQL converts between different data types implicitly (automatically) if necessary. For example, if you assign a `PLS_INTEGER` variable to a `NUMBER` variable, then PL/SQL converts the `PLS_INTEGER` value to a `NUMBER` value (because the internal representations of the values differ).

Whenever possible, minimize implicit conversions. For example:

- If a variable is to be either inserted into a table column or assigned a value from a table column, then give the variable the same data type as the table column.

  > **Tip:**
  >
  > Declare the variable with the `%TYPE` attribute, described in "%TYPE Attribute".

- Make each literal the same data type as the variable to which it is assigned or the expression in which it appears.
- Convert values from SQL data types to PL/SQL data types and then use the converted values in expressions.

  For example, convert `NUMBER` values to `PLS_INTEGER` values and then use the `PLS_INTEGER` values in expressions. `PLS_INTEGER` operations use hardware arithmetic, so they are faster than `NUMBER` operations, which use library arithmetic. For more information about the `PLS_INTEGER` data type, see "PLS_INTEGER and BINARY_INTEGER Data Types".
- Before assigning a value of one SQL data type to a variable of another SQL data type, explicitly convert the source value to the target data type, using a SQL conversion function (for information about SQL conversion functions, see *Oracle Database SQL Language Reference*).
- Overload your subprograms with versions that accept parameters of different data types and optimize each version for its parameter types. For information about overloaded subprograms, see "Overloaded Subprograms".

See Also:

- *Oracle Database SQL Language Reference* for information about implicit conversion of SQL data types (which are also PL/SQL data types)
- "Subtypes with Base Types in Same Data Type Family"

Use SQL Character Functions

SQL has many highly optimized character functions, which use low-level code that is more efficient than PL/SQL code. Use these functions instead of writing PL/SQL code to do the same things.

See:

- *Oracle Database SQL Language Reference* for information about SQL character functions that return character values
- *Oracle Database SQL Language Reference* for information about SQL character functions that return NLS character values
- *Oracle Database SQL Language Reference* for information about SQL character functions that return number values
- Example 7-6 for an example of PL/SQL code that uses SQL character function `REGEXP_LIKE`

Put Least Expensive Conditional Tests First

PL/SQL stops evaluating a logical expression as soon as it can determine the result. Take advantage of this short-circuit evaluation by putting the conditions that are least expensive to evaluate first in logical expressions whenever possible. For example, test the values of PL/SQL variables before testing function return values, so that if the variable tests fail, PL/SQL need not invoke the functions:

```
IF boolean_variable OR (number > 10) OR boolean_function(parameter) THEN ...
```

See Also:

"Short-Circuit Evaluation"

Bulk SQL and Bulk Binding

Bulk SQL minimizes the performance overhead of the communication between PL/SQL and SQL. The PL/SQL features that comprise bulk SQL are the `FORALL` statement and the `BULK COLLECT` clause. Assigning values to PL/SQL variables that appear in SQL statements is called **binding**.

PL/SQL and SQL communicate as follows: To run a `SELECT INTO` or DML statement, the PL/SQL engine sends the query or DML statement to the SQL engine. The SQL engine runs the query or DML statement and returns the result to the PL/SQL engine.

The `FORALL` statement sends DML statements from PL/SQL to SQL in batches rather than one at a time. The `BULK COLLECT` clause returns results from SQL to PL/SQL in batches rather than one at a time. If a query or DML statement affects four or more database rows, then bulk SQL can significantly improve performance.

> **Note:**
>
> You cannot perform bulk SQL on remote tables.

PL/SQL binding operations fall into these categories:

Binding Category	When This Binding Occurs
In-bind	When an `INSERT`, `UPDATE`, or `MERGE` statement stores a PL/SQL or host variable in the database
Out-bind	When the `RETURNING INTO` clause of an `INSERT`, `UPDATE`, `MERGE`, or `DELETE` statement assigns a database value to a PL/SQL or host variable
`DEFINE`	When a `SELECT` or `FETCH` statement assigns a database value to a PL/SQL or host variable

For in-binds and out-binds, bulk SQL uses **bulk binding**; that is, it binds an entire collection of values at once. For a collection of *n* elements, bulk SQL uses a single operation to perform the equivalent of *n* `SELECT INTO` or DML statements. A query that uses bulk SQL can return any number of rows, without using a `FETCH` statement for each one.

> **Note:**
>
> Parallel DML is disabled with bulk SQL.

Topics

- FORALL Statement
- BULK COLLECT Clause
- Using FORALL Statement and BULK COLLECT Clause Together
- Client Bulk-Binding of Host Arrays

FORALL Statement

The `FORALL` statement, a feature of bulk SQL, sends DML statements from PL/SQL to SQL in batches rather than one at a time.

To understand the `FORALL` statement, first consider the `FOR LOOP` statement in Example 13-7. It sends these DML statements from PL/SQL to SQL one at a time:

```
DELETE FROM employees_temp WHERE department_id = 10;
DELETE FROM employees_temp WHERE department_id = 30;
DELETE FROM employees_temp WHERE department_id = 70;
```

Now consider the `FORALL` statement in Example 13-8. It sends the same three DML statements from PL/SQL to SQL as a batch.

A `FORALL` statement is usually much faster than an equivalent `FOR LOOP` statement. However, a `FOR LOOP` statement can contain multiple DML statements, while a `FORALL` statement can contain only one. The batch of DML statements that a `FORALL` statement sends to SQL differ only in their `VALUES` and `WHERE` clauses. The values in those clauses must come from existing, populated collections.

> **Note:**
>
> The DML statement in a `FORALL` statement can reference multiple collections, but performance benefits apply only to collection references that use the `FORALL` index variable as an index.

Example 13-9 inserts the same collection elements into two database tables, using a `FOR LOOP` statement for the first table and a `FORALL` statement for the second table and showing how long each statement takes. (Times vary from run to run.)

In Example 13-10, the `FORALL` statement applies to a subset of a collection.

Topics

- Using FORALL Statements for Sparse Collections
- Unhandled Exceptions in FORALL Statements
- Handling FORALL Exceptions Immediately
- Handling FORALL Exceptions After FORALL Statement Completes
- Getting Number of Rows Affected by FORALL Statement

> **See Also:**
>
> - "FORALL Statement" for its complete syntax and semantics, including restrictions
> - "Implicit Cursors" for information about implicit cursor attributes in general and other implicit cursor attributes that you can use with the `FORALL` statement

Example 13-7 DELETE Statement in FOR LOOP Statement

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS SELECT * FROM employees;

DECLARE
  TYPE NumList IS VARRAY(20) OF NUMBER;
  depts NumList := NumList(10, 30, 70);  -- department numbers
BEGIN
  FOR i IN depts.FIRST..depts.LAST LOOP
```

```
    DELETE FROM employees_temp
    WHERE department_id = depts(i);
  END LOOP;
END;
/
```

Example 13-8 DELETE Statement in FORALL Statement

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS SELECT * FROM employees;

DECLARE
  TYPE NumList IS VARRAY(20) OF NUMBER;
  depts NumList := NumList(10, 30, 70);  -- department numbers
BEGIN
  FORALL i IN depts.FIRST..depts.LAST
    DELETE FROM employees_temp
    WHERE department_id = depts(i);
END;
/
```

Example 13-9 Time Difference for INSERT Statement in FOR LOOP and FORALL Statements

```
DROP TABLE parts1;
CREATE TABLE parts1 (
  pnum INTEGER,
  pname VARCHAR2(15)
);

DROP TABLE parts2;
CREATE TABLE parts2 (
  pnum INTEGER,
  pname VARCHAR2(15)
);

DECLARE
  TYPE NumTab IS TABLE OF parts1.pnum%TYPE INDEX BY PLS_INTEGER;
  TYPE NameTab IS TABLE OF parts1.pname%TYPE INDEX BY PLS_INTEGER;
  pnums   NumTab;
  pnames  NameTab;
  iterations  CONSTANT PLS_INTEGER := 50000;
  t1  INTEGER;
  t2  INTEGER;
  t3  INTEGER;
BEGIN
  FOR j IN 1..iterations LOOP  -- populate collections
    pnums(j) := j;
    pnames(j) := 'Part No. ' || TO_CHAR(j);
  END LOOP;

  t1 := DBMS_UTILITY.get_time;

  FOR i IN 1..iterations LOOP
    INSERT INTO parts1 (pnum, pname)
    VALUES (pnums(i), pnames(i));
  END LOOP;

  t2 := DBMS_UTILITY.get_time;

  FORALL i IN 1..iterations
    INSERT INTO parts2 (pnum, pname)
    VALUES (pnums(i), pnames(i));
```

```
  t3 := DBMS_UTILITY.get_time;

  DBMS_OUTPUT.PUT_LINE('Execution Time (secs)');
  DBMS_OUTPUT.PUT_LINE('---------------------');
  DBMS_OUTPUT.PUT_LINE('FOR LOOP: ' || TO_CHAR((t2 - t1)/100));
  DBMS_OUTPUT.PUT_LINE('FORALL:   ' || TO_CHAR((t3 - t2)/100));
  COMMIT;
END;
/
```

Result is similar to:

```
Execution Time (secs)
---------------------
FOR LOOP: 5.97
FORALL:   .07

PL/SQL procedure successfully completed.
```

Example 13-10 FORALL Statement for Subset of Collection

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS SELECT * FROM employees;

DECLARE
  TYPE NumList IS VARRAY(10) OF NUMBER;
  depts NumList := NumList(5,10,20,30,50,55,57,60,70,75);
BEGIN
  FORALL j IN 4..7
    DELETE FROM employees_temp WHERE department_id = depts(j);
END;
/
```

Using FORALL Statements for Sparse Collections

If the `FORALL` statement bounds clause references a sparse collection, then specify only existing index values, using either the `INDICES OF` or `VALUES OF` clause.

You can use `INDICES OF` for any collection except an associative array indexed by string. You can use `VALUES OF` only for a collection of `PLS_INTEGER` elements indexed by `PLS_INTEGER`.

A collection of `PLS_INTEGER` elements indexed by `PLS_INTEGER` can be an **index collection**; that is, a collection of pointers to elements of another collection (the **indexed collection**).

Index collections are useful for processing different subsets of the same collection with different `FORALL` statements. Instead of copying elements of the original collection into new collections that represent the subsets (which can use significant time and memory), represent each subset with an index collection and then use each index collection in the `VALUES OF` clause of a different `FORALL` statement.

> **See Also:**
>
> "Sparse Collections and SQL%BULK_EXCEPTIONS"

Example 13-11 FORALL Statements for Sparse Collection and Its Subsets

This example uses a `FORALL` statement with the `INDICES OF` clause to populate a table with the elements of a sparse collection. Then it uses two `FORALL` statements with `VALUES OF` clauses to populate two tables with subsets of a collection.

```
DROP TABLE valid_orders;
CREATE TABLE valid_orders (
  cust_name  VARCHAR2(32),
  amount     NUMBER(10,2)
);

DROP TABLE big_orders;
CREATE TABLE big_orders AS
  SELECT * FROM valid_orders
  WHERE 1 = 0;

DROP TABLE rejected_orders;
CREATE TABLE rejected_orders AS
  SELECT * FROM valid_orders
  WHERE 1 = 0;

DECLARE
  SUBTYPE cust_name IS valid_orders.cust_name%TYPE;
  TYPE cust_typ IS TABLE OF cust_name;
  cust_tab  cust_typ;  -- Collection of customer names

  SUBTYPE order_amount IS valid_orders.amount%TYPE;
  TYPE amount_typ IS TABLE OF NUMBER;
  amount_tab  amount_typ;  -- Collection of order amounts

  TYPE index_pointer_t IS TABLE OF PLS_INTEGER;

  /* Collections for pointers to elements of cust_tab collection
     (to represent two subsets of cust_tab): */

  big_order_tab       index_pointer_t := index_pointer_t();
  rejected_order_tab  index_pointer_t := index_pointer_t();

  PROCEDURE populate_data_collections IS
  BEGIN
    cust_tab := cust_typ(
      'Company1','Company2','Company3','Company4','Company5'
    );

    amount_tab := amount_typ(5000.01, 0, 150.25, 4000.00, NULL);
  END;

BEGIN
  populate_data_collections;

  DBMS_OUTPUT.PUT_LINE ('--- Original order data ---');

  FOR i IN 1..cust_tab.LAST LOOP
    DBMS_OUTPUT.PUT_LINE (
      'Customer #' || i || ', ' || cust_tab(i) || ': $' || amount_tab(i)
    );
  END LOOP;

  -- Delete invalid orders:

  FOR i IN 1..cust_tab.LAST LOOP
```

```
    IF amount_tab(i) IS NULL OR amount_tab(i) = 0 THEN
      cust_tab.delete(i);
      amount_tab.delete(i);
    END IF;
  END LOOP;

  -- cust_tab is now a sparse collection.

  DBMS_OUTPUT.PUT_LINE ('--- Order data with invalid orders deleted ---');

  FOR i IN 1..cust_tab.LAST LOOP
    IF cust_tab.EXISTS(i) THEN
      DBMS_OUTPUT.PUT_LINE (
        'Customer #' || i || ', ' || cust_tab(i) || ': $' || amount_tab(i)
      );
    END IF;
  END LOOP;

  -- Using sparse collection, populate valid_orders table:

  FORALL i IN INDICES OF cust_tab
    INSERT INTO valid_orders (cust_name, amount)
    VALUES (cust_tab(i), amount_tab(i));

  populate_data_collections;  -- Restore original order data

  -- cust_tab is a dense collection again.

  /* Populate collections of pointers to elements of cust_tab collection
     (which represent two subsets of cust_tab): */

  FOR i IN cust_tab.FIRST .. cust_tab.LAST LOOP
    IF amount_tab(i) IS NULL OR amount_tab(i) = 0 THEN
      rejected_order_tab.EXTEND;
      rejected_order_tab(rejected_order_tab.LAST) := i;
    END IF;

    IF amount_tab(i) > 2000 THEN
      big_order_tab.EXTEND;
      big_order_tab(big_order_tab.LAST) := i;
    END IF;
  END LOOP;

  /* Using each subset in a different FORALL statement,
     populate rejected_orders and big_orders tables: */

  FORALL i IN VALUES OF rejected_order_tab
    INSERT INTO rejected_orders (cust_name, amount)
    VALUES (cust_tab(i), amount_tab(i));

  FORALL i IN VALUES OF big_order_tab
    INSERT INTO big_orders (cust_name, amount)
    VALUES (cust_tab(i), amount_tab(i));
END;
/
```

Result:

```
--- Original order data ---
Customer #1, Company1: $5000.01
Customer #2, Company2: $0
Customer #3, Company3: $150.25
```

```
Customer #4, Company4: $4000
Customer #5, Company5: $
--- Data with invalid orders deleted ---
Customer #1, Company1: $5000.01
Customer #3, Company3: $150.25
Customer #4, Company4: $4000
```

Verify that correct order details were stored:

```
SELECT cust_name "Customer", amount "Valid order amount"
FROM valid_orders
ORDER BY cust_name;
```

Result:

```
Customer                         Valid order amount
-------------------------------- ------------------
Company1                                    5000.01
Company3                                     150.25
Company4                                       4000

3 rows selected.
```

Query:

```
SELECT cust_name "Customer", amount "Big order amount"
FROM big_orders
ORDER BY cust_name;
```

Result:

```
Customer                         Big order amount
-------------------------------- ----------------
Company1                                  5000.01
Company4                                     4000

2 rows selected.
```

Query:

```
SELECT cust_name "Customer", amount "Rejected order amount"
FROM rejected_orders
ORDER BY cust_name;
```

Result:

```
Customer                         Rejected order amount
-------------------------------- ---------------------
Company2                                             0
Company5

2 rows selected.
```

Unhandled Exceptions in FORALL Statements

In a `FORALL` statement without the `SAVE EXCEPTIONS` clause, if one DML statement raises an unhandled exception, then PL/SQL stops the `FORALL` statement and rolls back all changes made by previous DML statements.

For example, the `FORALL` statement in Example 13-8 processes these DML statements in this order, unless one of them raises an unhandled exception:

```
DELETE FROM employees_temp WHERE department_id = depts(10);
DELETE FROM employees_temp WHERE department_id = depts(30);
DELETE FROM employees_temp WHERE department_id = depts(70);
```

If the third statement raises an unhandled exception, then PL/SQL rolls back the changes that the first and second statements made. If the second statement raises an unhandled exception, then PL/SQL rolls back the changes that the first statement made and never runs the third statement.

You can handle exceptions raised in a `FORALL` statement in either of these ways:

- As each exception is raised (see "Handling FORALL Exceptions Immediately")
- After the `FORALL` statement completes execution, by including the `SAVE EXCEPTIONS` clause (see "Handling FORALL Exceptions After FORALL Statement Completes")

Handling FORALL Exceptions Immediately

To handle exceptions raised in a `FORALL` statement immediately, omit the `SAVE EXCEPTIONS` clause and write the appropriate exception handlers.

If one DML statement raises a handled exception, then PL/SQL rolls back the changes made by that statement, but does not roll back changes made by previous DML statements.

In Example 13-12, the `FORALL` statement is designed to run three `UPDATE` statements. However, the second one raises an exception. An exception handler handles the exception, displaying the error message and committing the change made by the first `UPDATE` statement. The third `UPDATE` statement never runs.

For information about exception handlers, see PL/SQL Error Handling.

Example 13-12   Handling FORALL Exceptions Immediately

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp (
  deptno NUMBER(2),
  job VARCHAR2(18)
);

CREATE OR REPLACE PROCEDURE p AUTHID DEFINER AS
  TYPE NumList IS TABLE OF NUMBER;

  depts          NumList := NumList(10, 20, 30);
  error_message  VARCHAR2(100);

BEGIN
  -- Populate table:

  INSERT INTO emp_temp (deptno, job) VALUES (10, 'Clerk');
  INSERT INTO emp_temp (deptno, job) VALUES (20, 'Bookkeeper');
  INSERT INTO emp_temp (deptno, job) VALUES (30, 'Analyst');
  COMMIT;

  -- Append 9-character string to each job:

  FORALL j IN depts.FIRST..depts.LAST
    UPDATE emp_temp SET job = job || ' (Senior)'
    WHERE deptno = depts(j);

EXCEPTION
  WHEN OTHERS THEN
    error_message := SQLERRM;
```

```
    DBMS_OUTPUT.PUT_LINE (error_message);

    COMMIT;  -- Commit results of successful updates
    RAISE;
END;
/
```

Result:

```
Procedure created.
```

Invoke procedure:

```
BEGIN
  p;
END;
/
```

Result:

```
ORA-12899: value too large for column "HR"."EMP_TEMP"."JOB" (actual: 19,
maximum: 18)
BEGIN
*
ERROR at line 1:
ORA-12899: value too large for column "HR"."EMP_TEMP"."JOB" (actual: 19,
maximum: 18)
ORA-06512: at "HR.P", line 27
ORA-06512: at line 2
```

Query:

```
SELECT * FROM emp_temp;
```

Result:

```
    DEPTNO JOB
---------- ------------------
        10 Clerk (Senior)
        20 Bookkeeper
        30 Analyst

3 rows selected.
```

Handling FORALL Exceptions After FORALL Statement Completes

To allow a `FORALL` statement to continue even if some of its DML statements fail, include the `SAVE EXCEPTIONS` clause. When a DML statement fails, PL/SQL does not raise an exception; instead, it saves information about the failure. After the `FORALL` statement completes, PL/SQL raises a single exception for the `FORALL` statement (ORA-24381).

In the exception handler for ORA-24381, you can get information about each individual DML statement failure from the implicit cursor attribute `SQL%BULK_EXCEPTIONS`.

`SQL%BULK_EXCEPTIONS` is like an associative array of information about the DML statements that failed during the most recently run `FORALL` statement.

`SQL%BULK_EXCEPTIONS.COUNT` is the number of DML statements that failed. If `SQL%BULK_EXCEPTIONS.COUNT` is not zero, then for each index value *i* from 1 through `SQL%BULK_EXCEPTIONS.COUNT`:

- `SQL%BULK_EXCEPTIONS(i).ERROR_INDEX` is the number of the DML statement that failed.
- `SQL%BULK_EXCEPTIONS(i).ERROR_CODE` is the Oracle Database error code for the failure.

For example, if a `FORALL SAVE EXCEPTIONS` statement runs 100 DML statements, and the tenth and sixty-fourth ones fail with error codes ORA-12899 and ORA-19278, respectively, then:

- `SQL%BULK_EXCEPTIONS.COUNT` = 2
- `SQL%BULK_EXCEPTIONS(1).ERROR_INDEX` = 10
- `SQL%BULK_EXCEPTIONS(1).ERROR_CODE` = 12899
- `SQL%BULK_EXCEPTIONS(2).ERROR_INDEX` = 64
- `SQL%BULK_EXCEPTIONS(2).ERROR_CODE` = 19278

> **Note:**
>
> After a `FORALL` statement *without* the `SAVE EXCEPTIONS` clause raises an exception, `SQL%BULK_EXCEPTIONS.COUNT` = 1.

With the error code, you can get the associated error message with the `SQLERRM` function (described in "SQLERRM Function"):

```
SQLERRM(-(SQL%BULK_EXCEPTIONS(i).ERROR_CODE))
```

However, the error message that `SQLERRM` returns excludes any substitution arguments (compare the error messages in Example 13-12 and Example 13-13).

Example 13-13 is like Example 13-12 except:

- The `FORALL` statement includes the `SAVE EXCEPTIONS` clause.
- The exception-handling part has an exception handler for ORA-24381, the internally defined exception that PL/SQL raises implicitly when a bulk operation raises and saves exceptions. The example gives ORA-24381 the user-defined name `dml_errors`.
- The exception handler for `dml_errors` uses `SQL%BULK_EXCEPTIONS` and `SQLERRM` (and some local variables) to show the error message and which statement, collection item, and string caused the error.

Example 13-13 Handling FORALL Exceptions After FORALL Statement Completes

```
CREATE OR REPLACE PROCEDURE p AUTHID DEFINER AS
  TYPE NumList IS TABLE OF NUMBER;
  depts        NumList := NumList(10, 20, 30);

  error_message  VARCHAR2(100);
  bad_stmt_no    PLS_INTEGER;
  bad_deptno     emp_temp.deptno%TYPE;
  bad_job        emp_temp.job%TYPE;

  dml_errors  EXCEPTION;
  PRAGMA EXCEPTION_INIT(dml_errors, -24381);
BEGIN
  -- Populate table:

  INSERT INTO emp_temp (deptno, job) VALUES (10, 'Clerk');
  INSERT INTO emp_temp (deptno, job) VALUES (20, 'Bookkeeper');
  INSERT INTO emp_temp (deptno, job) VALUES (30, 'Analyst');
```

```
  COMMIT;

  -- Append 9-character string to each job:

  FORALL j IN depts.FIRST..depts.LAST SAVE EXCEPTIONS
    UPDATE emp_temp SET job = job || ' (Senior)'
    WHERE deptno = depts(j);

EXCEPTION
  WHEN dml_errors THEN
    FOR i IN 1..SQL%BULK_EXCEPTIONS.COUNT LOOP
      error_message := SQLERRM(-(SQL%BULK_EXCEPTIONS(i).ERROR_CODE));
      DBMS_OUTPUT.PUT_LINE (error_message);

      bad_stmt_no := SQL%BULK_EXCEPTIONS(i).ERROR_INDEX;
      DBMS_OUTPUT.PUT_LINE('Bad statement #: ' || bad_stmt_no);

      bad_deptno := depts(bad_stmt_no);
      DBMS_OUTPUT.PUT_LINE('Bad department #: ' || bad_deptno);

      SELECT job INTO bad_job FROM emp_temp WHERE deptno = bad_deptno;

      DBMS_OUTPUT.PUT_LINE('Bad job: ' || bad_job);
    END LOOP;

    COMMIT;  -- Commit results of successful updates

    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Unrecognized error.');
      RAISE;
END;
/
```

Result:

```
Procedure created.
```

Invoke procedure:

```
BEGIN
  p;
END;
/
```

Result:

```
ORA-12899: value too large for column  (actual: , maximum: )
Bad statement #: 2
Bad department #: 20
Bad job: Bookkeeper

PL/SQL procedure successfully completed.
```

Query:

```
SELECT * FROM emp_temp;
```

Result:

```
    DEPTNO JOB
---------- ------------------
        10 Clerk (Senior)
```

```
        20 Bookkeeper
        30 Analyst (Senior)

3 rows selected.
```

Sparse Collections and SQL%BULK_EXCEPTIONS

If the `FORALL` statement bounds clause references a sparse collection, then to find the collection element that caused a DML statement to fail, you must step through the elements one by one until you find the element whose index is `SQL%BULK_EXCEPTIONS(i).ERROR_INDEX`. Then, if the `FORALL` statement uses the `VALUES OF` clause to reference a collection of pointers into another collection, you must find the element of the other collection whose index is `SQL%BULK_EXCEPTIONS(i).ERROR_INDEX`.

Getting Number of Rows Affected by FORALL Statement

After a `FORALL` statement completes, you can get the number of rows that each DML statement affected from the implicit cursor attribute `SQL%BULK_ROWCOUNT`.

To get the total number of rows affected by the `FORALL` statement, use the implicit cursor attribute `SQL%ROWCOUNT`, described in "SQL%ROWCOUNT Attribute: How Many Rows Were Affected?".

`SQL%BULK_ROWCOUNT` is like an associative array whose *i*th element is the number of rows affected by the *i*th DML statement in the most recently completed `FORALL` statement. The data type of the element is `INTEGER`.

> **Note:**
>
> If a server is Oracle Database 12c or later and its client is Oracle Database 11g release 2 or earlier (or the reverse), then the maximum number that `SQL%BULK_ROWCOUNT` returns is 4,294,967,295.

Example 13-14 uses `SQL%BULK_ROWCOUNT` to show how many rows each `DELETE` statement in the `FORALL` statement deleted and `SQL%ROWCOUNT` to show the total number of rows deleted.

Example 13-15 uses `SQL%BULK_ROWCOUNT` to show how many rows each `INSERT SELECT` construct in the `FORALL` statement inserted and `SQL%ROWCOUNT` to show the total number of rows inserted.

Example 13-14 Showing Number of Rows Affected by Each DELETE in FORALL

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp AS SELECT * FROM employees;

DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  depts NumList := NumList(30, 50, 60);
BEGIN
  FORALL j IN depts.FIRST..depts.LAST
    DELETE FROM emp_temp WHERE department_id = depts(j);

  FOR i IN depts.FIRST..depts.LAST LOOP
    DBMS_OUTPUT.PUT_LINE (
      'Statement #' || i || ' deleted ' ||
      SQL%BULK_ROWCOUNT(i) || ' rows.'
```

```
    );
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('Total rows deleted: ' || SQL%ROWCOUNT);
END;
/
```

Result:

```
Statement #1 deleted 6 rows.
Statement #2 deleted 45 rows.
Statement #3 deleted 5 rows.
Total rows deleted: 56
```

Example 13-15 Showing Number of Rows Affected by Each INSERT SELECT in FORALL

```
DROP TABLE emp_by_dept;
CREATE TABLE emp_by_dept AS
  SELECT employee_id, department_id
  FROM employees
  WHERE 1 = 0;

DECLARE
  TYPE dept_tab IS TABLE OF departments.department_id%TYPE;
  deptnums  dept_tab;
BEGIN
  SELECT department_id BULK COLLECT INTO deptnums FROM departments;

  FORALL i IN 1..deptnums.COUNT
    INSERT INTO emp_by_dept (employee_id, department_id)
      SELECT employee_id, department_id
      FROM employees
      WHERE department_id = deptnums(i)
      ORDER BY department_id, employee_id;

  FOR i IN 1..deptnums.COUNT LOOP
    -- Count how many rows were inserted for each department; that is,
    -- how many employees are in each department.
    DBMS_OUTPUT.PUT_LINE (
      'Dept '||deptnums(i)||': inserted '||
      SQL%BULK_ROWCOUNT(i)||' records'
    );
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Total records inserted: ' || SQL%ROWCOUNT);
END;
/
```

Result:

```
Dept 10: inserted 1 records
Dept 20: inserted 2 records
Dept 30: inserted 6 records
Dept 40: inserted 1 records
Dept 50: inserted 45 records
Dept 60: inserted 5 records
Dept 70: inserted 1 records
Dept 80: inserted 34 records
Dept 90: inserted 3 records
Dept 100: inserted 6 records
Dept 110: inserted 2 records
Dept 120: inserted 0 records
Dept 130: inserted 0 records
```

```
Dept 140: inserted 0 records
Dept 150: inserted 0 records
Dept 160: inserted 0 records
Dept 170: inserted 0 records
Dept 180: inserted 0 records
Dept 190: inserted 0 records
Dept 200: inserted 0 records
Dept 210: inserted 0 records
Dept 220: inserted 0 records
Dept 230: inserted 0 records
Dept 240: inserted 0 records
Dept 250: inserted 0 records
Dept 260: inserted 0 records
Dept 270: inserted 0 records
Dept 280: inserted 0 records
Total records inserted: 106
```

BULK COLLECT Clause

The `BULK COLLECT` clause, a feature of bulk SQL, returns results from SQL to PL/SQL in batches rather than one at a time.

The `BULK COLLECT` clause can appear in:

- `SELECT INTO` statement
- `FETCH` statement
- `RETURNING INTO` clause of:
  - `DELETE` statement
  - `INSERT` statement
  - `UPDATE` statement
  - `MERGE` statement
  - `EXECUTE IMMEDIATE` statement

With the `BULK COLLECT` clause, each of the preceding statements retrieves an entire result set and stores it in one or more collection variables in a single operation (which is more efficient than using a loop statement to retrieve one result row at a time).

> **Note:**
>
> PL/SQL processes the `BULK COLLECT` clause similar to the way it processes a `FETCH` statement inside a `LOOP` statement. PL/SQL does not raise an exception when a statement with a `BULK COLLECT` clause returns no rows. You must check the target collections for emptiness, as in Example 13-22.

Topics

- SELECT INTO Statement with BULK COLLECT Clause
- FETCH Statement with BULK COLLECT Clause
- RETURNING INTO Clause with BULK COLLECT Clause

SELECT INTO Statement with BULK COLLECT Clause

The `SELECT INTO` statement with the `BULK COLLECT` clause (also called the `SELECT BULK COLLECT INTO` statement) selects an entire result set into one or more collection variables.

For more information, see "SELECT INTO Statement".

> **Caution:**
>
> The `SELECT BULK COLLECT INTO` statement is vulnerable to aliasing, which can cause unexpected results. For details, see "SELECT BULK COLLECT INTO Statements and Aliasing".

Example 13-16 uses a `SELECT BULK COLLECT INTO` statement to select two database columns into two collections (nested tables).

Example 13-17 uses a `SELECT BULK COLLECT INTO` statement to select a result set into a nested table of records.

Topics

- SELECT BULK COLLECT INTO Statements and Aliasing
- Row Limits for SELECT BULK COLLECT INTO Statements
- Guidelines for Looping Through Collections

Example 13-16 Bulk-Selecting Two Database Columns into Two Nested Tables

```
DECLARE
  TYPE NumTab IS TABLE OF employees.employee_id%TYPE;
  TYPE NameTab IS TABLE OF employees.last_name%TYPE;

  enums NumTab;
  names NameTab;

  PROCEDURE print_first_n (n POSITIVE) IS
  BEGIN
    IF enums.COUNT = 0 THEN
      DBMS_OUTPUT.PUT_LINE ('Collections are empty.');
    ELSE
      DBMS_OUTPUT.PUT_LINE ('First ' || n || ' employees:');

      FOR i IN 1 .. n LOOP
        DBMS_OUTPUT.PUT_LINE (
          '  Employee #' || enums(i) || ': ' || names(i));
      END LOOP;
    END IF;
  END;

BEGIN
  SELECT employee_id, last_name
  BULK COLLECT INTO enums, names
  FROM employees
```

```
  ORDER BY employee_id;

  print_first_n(3);
  print_first_n(6);
END;
/
```

Result:

```
First 3 employees:
Employee #100: King
Employee #101: Yang
Employee #102: Garcia
First 6 employees:
Employee #100: King
Employee #101: Yang
Employee #102: Garcia
Employee #103: James
Employee #104: Miller
Employee #105: Williams
```

Example 13-17 Bulk-Selecting into Nested Table of Records

```
DECLARE
  CURSOR c1 IS
    SELECT first_name, last_name, hire_date
    FROM employees;

  TYPE NameSet IS TABLE OF c1%ROWTYPE;

  stock_managers  NameSet;  -- nested table of records

BEGIN
  -- Assign values to nested table of records:

  SELECT first_name, last_name, hire_date
    BULK COLLECT INTO stock_managers
    FROM employees
    WHERE job_id = 'ST_MAN'
    ORDER BY hire_date;

  -- Print nested table of records:

    FOR i IN stock_managers.FIRST .. stock_managers.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (
        stock_managers(i).hire_date || ' ' ||
        stock_managers(i).last_name  || ', ' ||
        stock_managers(i).first_name
      );
    END LOOP;END;
/
```

Result:

```
01-MAY-13 Kaufling, Payam
18-JUL-14 Weiss, Matthew
10-APR-15 Fripp, Adam
10-OCT-15 Vollman, Shanta
16-NOV-17 Mourgos, Kevin
```

SELECT BULK COLLECT INTO Statements and Aliasing

In a statement of the form

```
SELECT column BULK COLLECT INTO collection FROM table ...
```

column and *collection* are analogous to IN NOCOPY and OUT NOCOPY subprogram parameters, respectively, and PL/SQL passes them by reference. As with subprogram parameters that are passed by reference, aliasing can cause unexpected results.

> **See Also:**
>
> "Subprogram Parameter Aliasing with Parameters Passed by Reference"

In Example 13-18, the intention is to select specific values from a collection, numbers1, and then store them in the same collection. The unexpected result is that all elements of numbers1 are deleted. For workarounds, see Example 13-19 and Example 13-20.

Example 13-19 uses a cursor to achieve the result intended by Example 13-18.

Example 13-20 selects specific values from a collection, numbers1, and then stores them in a different collection, numbers2. Example 13-20 runs faster than Example 13-19.

Example 13-18 SELECT BULK COLLECT INTO Statement with Unexpected Results

```
CREATE OR REPLACE TYPE numbers_type IS
  TABLE OF INTEGER
/
CREATE OR REPLACE PROCEDURE p (i IN INTEGER) AUTHID DEFINER IS
  numbers1  numbers_type := numbers_type(1,2,3,4,5);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before SELECT statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());

  FOR j IN 1..numbers1.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
  END LOOP;

  --Self-selecting BULK COLLECT INTO clause:

  SELECT a.COLUMN_VALUE
  BULK COLLECT INTO numbers1
  FROM TABLE(numbers1) a
  WHERE a.COLUMN_VALUE > p.i
  ORDER BY a.COLUMN_VALUE;

  DBMS_OUTPUT.PUT_LINE('After SELECT statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());
```

```
END p;
/
```

Invoke p:

```
BEGIN
  p(2);
END;
/
```

Result:

```
Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
After SELECT statement
numbers1.COUNT() = 0

PL/SQL procedure successfully completed.
```

Invoke p:

```
BEGIN
  p(10);
END;
/
```

Result:

```
Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
After SELECT statement
numbers1.COUNT() = 0
```

Example 13-19 Cursor Workaround for Example 13-18

```
CREATE OR REPLACE TYPE numbers_type IS
  TABLE OF INTEGER
/
CREATE OR REPLACE PROCEDURE p (i IN INTEGER) AUTHID DEFINER IS
  numbers1  numbers_type := numbers_type(1,2,3,4,5);

  CURSOR c IS
    SELECT a.COLUMN_VALUE
    FROM TABLE(numbers1) a
    WHERE a.COLUMN_VALUE > p.i
    ORDER BY a.COLUMN_VALUE;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Before FETCH statement');
    DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());

    FOR j IN 1..numbers1.COUNT() LOOP
      DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
    END LOOP;
```

```
  OPEN c;
  FETCH c BULK COLLECT INTO numbers1;
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE('After FETCH statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());

  IF numbers1.COUNT() > 0 THEN
    FOR j IN 1..numbers1.COUNT() LOOP
      DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
    END LOOP;
  END IF;
END p;
/
```

Invoke `p`:

```
BEGIN
  p(2);
END;
/
```

Result:

```
Before FETCH statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
After FETCH statement
numbers1.COUNT() = 3
numbers1(1) = 3
numbers1(2) = 4
numbers1(3) = 5
```

Invoke `p`:

```
BEGIN
  p(10);
END;
/
```

Result:

```
Before FETCH statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
After FETCH statement
numbers1.COUNT() = 0
```

Example 13-20 Second Collection Workaround for Example 13-18

```
CREATE OR REPLACE TYPE numbers_type IS
  TABLE OF INTEGER
/
CREATE OR REPLACE PROCEDURE p (i IN INTEGER) AUTHID DEFINER IS
```

```
  numbers1  numbers_type := numbers_type(1,2,3,4,5);
 numbers2  numbers_type := numbers_type(0,0,0,0,0);

BEGIN
  DBMS_OUTPUT.PUT_LINE('Before SELECT statement');

  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());

  FOR j IN 1..numbers1.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('numbers2.COUNT() = ' || numbers2.COUNT());

  FOR j IN 1..numbers2.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers2(' || j || ') = ' || numbers2(j));
  END LOOP;

  SELECT a.COLUMN_VALUE
  BULK COLLECT INTO numbers2      -- numbers2 appears here
  FROM TABLE(numbers1) a         -- numbers1 appears here
  WHERE a.COLUMN_VALUE > p.i
  ORDER BY a.COLUMN_VALUE;

  DBMS_OUTPUT.PUT_LINE('After SELECT statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());

  IF numbers1.COUNT() > 0 THEN
    FOR j IN 1..numbers1.COUNT() LOOP
      DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
    END LOOP;
  END IF;

  DBMS_OUTPUT.PUT_LINE('numbers2.COUNT() = ' || numbers2.COUNT());

  IF numbers2.COUNT() > 0 THEN
    FOR j IN 1..numbers2.COUNT() LOOP
      DBMS_OUTPUT.PUT_LINE('numbers2(' || j || ') = ' || numbers2(j));
    END LOOP;
  END IF;
END p;
/
```

Invoke p:

```
BEGIN
  p(2);
 END;
/
```

Result:

```
Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 5
numbers2(1) = 0
numbers2(2) = 0
numbers2(3) = 0
```

```
numbers2(4) = 0
numbers2(5) = 0
After SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 3
numbers2(1) = 3
numbers2(2) = 4
numbers2(3) = 5

PL/SQL procedure successfully completed.
```

Invoke `p`:

```
BEGIN
  p(10);
END;
/
```

Result:

```
Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 5
numbers2(1) = 0
numbers2(2) = 0
numbers2(3) = 0
numbers2(4) = 0
numbers2(5) = 0
After SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 0
```

Row Limits for SELECT BULK COLLECT INTO Statements

A `SELECT BULK COLLECT INTO` statement that returns a large number of rows produces a large collection. To limit the number of rows and the collection size, use one of these:

- `ROWNUM` pseudocolumn (described in *Oracle Database SQL Language Reference*)
- `SAMPLE` clause (described in *Oracle Database SQL Language Reference*)
- `FETCH FIRST` clause (described in *Oracle Database SQL Language Reference*)

Example 13-21 shows several ways to limit the number of rows that a `SELECT BULK COLLECT INTO` statement returns.

Example 13-21 Limiting Bulk Selection with ROWNUM, SAMPLE, and FETCH FIRST

```
DECLARE
  TYPE SalList IS TABLE OF employees.salary%TYPE;
  sals SalList;
BEGIN
  SELECT salary BULK COLLECT INTO sals FROM employees
    WHERE ROWNUM <= 50;

  SELECT salary BULK COLLECT INTO sals FROM employees
    SAMPLE (10);

  SELECT salary BULK COLLECT INTO sals FROM employees
    FETCH FIRST 50 ROWS ONLY;
END;
/
```

Guidelines for Looping Through Collections

When a result set is stored in a collection, it is easy to loop through the rows and refer to different columns. This technique can be very fast, but also very memory-intensive. If you use it often:

- To loop once through the result set, use a cursor `FOR LOOP` (see "Processing Query Result Sets With Cursor FOR LOOP Statements").

  This technique avoids the memory overhead of storing a copy of the result set.

- Instead of looping through the result set to search for certain values or filter the results into a smaller set, do the searching or filtering in the query of the `SELECT INTO` statement.

  For example, in simple queries, use `WHERE` clauses; in queries that compare multiple result sets, use set operators such as `INTERSECT` and `MINUS`. For information about set operators, see *Oracle Database SQL Language Reference*.

- Instead of looping through the result set and running another query for each result row, use a subquery in the query of the `SELECT INTO` statement (see "Processing Query Result Sets with Subqueries").
- Instead of looping through the result set and running another DML statement for each result row, use the `FORALL` statement (see "FORALL Statement").

FETCH Statement with BULK COLLECT Clause

The `FETCH` statement with the `BULK COLLECT` clause (also called the `FETCH BULK COLLECT` statement) fetches an entire result set into one or more collection variables.

For more information, see "FETCH Statement".

Example 13-22 uses a `FETCH BULK COLLECT` statement to fetch an entire result set into two collections (nested tables).

Example 13-23 uses a `FETCH BULK COLLECT` statement to fetch a result set into a collection (nested table) of records.

Example 13-22 Bulk-Fetching into Two Nested Tables

```
DECLARE
  TYPE NameList IS TABLE OF employees.last_name%TYPE;
  TYPE SalList IS TABLE OF employees.salary%TYPE;
```

```
  CURSOR c1 IS
    SELECT last_name, salary
    FROM employees
    WHERE salary > 10000
    ORDER BY last_name;

  names  NameList;
  sals   SalList;

  TYPE RecList IS TABLE OF c1%ROWTYPE;
  recs RecList;

  v_limit PLS_INTEGER := 10;

  PROCEDURE print_results IS
  BEGIN
    -- Check if collections are empty:

    IF names IS NULL OR names.COUNT = 0 THEN
      DBMS_OUTPUT.PUT_LINE('No results!');
    ELSE
      DBMS_OUTPUT.PUT_LINE('Result: ');
      FOR i IN names.FIRST .. names.LAST
      LOOP
        DBMS_OUTPUT.PUT_LINE('  Employee ' || names(i) || ': $' || sals(i));
      END LOOP;
    END IF;
  END;

BEGIN
  DBMS_OUTPUT.PUT_LINE ('--- Processing all results simultaneously ---');
  OPEN c1;
  FETCH c1 BULK COLLECT INTO names, sals;
  CLOSE c1;
  print_results();
  DBMS_OUTPUT.PUT_LINE ('--- Processing ' || v_limit || ' rows at a time
---');
  OPEN c1;
  LOOP
    FETCH c1 BULK COLLECT INTO names, sals LIMIT v_limit;
    EXIT WHEN names.COUNT = 0;
    print_results();
  END LOOP;
  CLOSE c1;
  DBMS_OUTPUT.PUT_LINE ('--- Fetching records rather than columns ---');
  OPEN c1;
  FETCH c1 BULK COLLECT INTO recs;
  FOR i IN recs.FIRST .. recs.LAST
  LOOP
    -- Now all columns from result set come from one record
    DBMS_OUTPUT.PUT_LINE (
      '  Employee ' || recs(i).last_name || ': $' || recs(i).salary
    );
  END LOOP;
END;
/
```

Result:

```
--- Processing all results simultaneously ---
Result:
Employee Abel: $11000
Employee Cambrault: $11000
Employee Errazuriz: $12000
Employee Garcia: $17000
Employee Gruenberg: $12008
Employee Higgins: $12008
Employee King: $24000
Employee Li: $11000
Employee Martinez: $13000
Employee Ozer: $11500
Employee Partners: $13500
Employee Singh: $14000
Employee Vishney: $10500
Employee Yang: $17000
Employee Zlotkey: $10500
--- Processing 10 rows at a time ---
Result:
Employee Abel: $11000
Employee Cambrault: $11000
Employee Errazuriz: $12000
Employee Garcia: $17000
Employee Gruenberg: $12008
Employee Higgins: $12008
Employee King: $24000
Employee Li: $11000
Employee Martinez: $13000
Employee Ozer: $11500
Result:
Employee Partners: $13500
Employee Singh: $14000
Employee Vishney: $10500
Employee Yang: $17000
Employee Zlotkey: $10500
--- Fetching records rather than columns ---
Employee Abel: $11000
Employee Cambrault: $11000
Employee Errazuriz: $12000
Employee Garcia: $17000
Employee Gruenberg: $12008
Employee Higgins: $12008
Employee King: $24000
Employee Li: $11000
Employee Martinez: $13000
Employee Ozer: $11500
Employee Partners: $13500
Employee Singh: $14000
Employee Vishney: $10500
Employee Yang: $17000
Employee Zlotkey: $10500
```

Example 13-23 Bulk-Fetching into Nested Table of Records

```
DECLARE
  CURSOR c1 IS
    SELECT first_name, last_name, hire_date
    FROM employees;

  TYPE NameSet IS TABLE OF c1%ROWTYPE;
  stock_managers  NameSet;  -- nested table of records

  TYPE cursor_var_type is REF CURSOR;
  cv cursor_var_type;

BEGIN
  -- Assign values to nested table of records:

  OPEN cv FOR
    SELECT first_name, last_name, hire_date
    FROM employees
    WHERE job_id = 'ST_MAN'
    ORDER BY hire_date;

  FETCH cv BULK COLLECT INTO stock_managers;
  CLOSE cv;

  -- Print nested table of records:

    FOR i IN stock_managers.FIRST .. stock_managers.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (
        stock_managers(i).hire_date || ' ' ||
        stock_managers(i).last_name  || ', ' ||
        stock_managers(i).first_name
      );
    END LOOP;END;
/
```

Result:

```
01-MAY-13 Kaufling, Payam
18-JUL-14 Weiss, Matthew
10-APR-15 Fripp, Adam
10-OCT-15 Vollman, Shanta
16-NOV-17 Mourgos, Kevin
```

Row Limits for FETCH BULK COLLECT Statements

A `FETCH BULK COLLECT` statement that returns a large number of rows produces a large collection. To limit the number of rows and the collection size, use the `LIMIT` clause.

In Example 13-24, with each iteration of the `LOOP` statement, the `FETCH` statement fetches ten rows (or fewer) into associative array `empids` (overwriting the previous values). Note the exit condition for the `LOOP` statement.

Example 13-24 Limiting Bulk FETCH with LIMIT

```
DECLARE
  TYPE numtab IS TABLE OF NUMBER INDEX BY PLS_INTEGER;

  CURSOR c1 IS
    SELECT employee_id
    FROM employees
    WHERE department_id = 80
    ORDER BY employee_id;

  empids  numtab;
BEGIN
  OPEN c1;
  LOOP  -- Fetch 10 rows or fewer in each iteration
    FETCH c1 BULK COLLECT INTO empids LIMIT 10;
    DBMS_OUTPUT.PUT_LINE ('------- Results from One Bulk Fetch --------');
    FOR i IN 1..empids.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE ('Employee Id: ' || empids(i));
    END LOOP;
    EXIT WHEN c1%NOTFOUND;
  END LOOP;
  CLOSE c1;
END;
/
```

Result:

```
------- Results from One Bulk Fetch --------
Employee Id: 145
Employee Id: 146
Employee Id: 147
Employee Id: 148
Employee Id: 149
Employee Id: 150
Employee Id: 151
Employee Id: 152
Employee Id: 153
Employee Id: 154
------- Results from One Bulk Fetch --------
Employee Id: 155
Employee Id: 156
Employee Id: 157
Employee Id: 158
Employee Id: 159
Employee Id: 160
Employee Id: 161
Employee Id: 162
Employee Id: 163
Employee Id: 164
------- Results from One Bulk Fetch --------
Employee Id: 165
Employee Id: 166
Employee Id: 167
Employee Id: 168
Employee Id: 169
Employee Id: 170
Employee Id: 171
Employee Id: 172
Employee Id: 173
Employee Id: 174
------- Results from One Bulk Fetch --------
Employee Id: 175
```

```
Employee Id: 176
Employee Id: 177
Employee Id: 179
```

RETURNING INTO Clause with BULK COLLECT Clause

The `RETURNING INTO` clause with the `BULK COLLECT` clause (also called the `RETURNING BULK COLLECT INTO` clause) can appear in an `INSERT`, `UPDATE`, `MERGE`, `DELETE`, or `EXECUTE IMMEDIATE` statement. With the `RETURNING BULK COLLECT INTO` clause, the statement stores its result set in one or more collections.

For more information, see "RETURNING INTO Clause".

Example 13-25 uses a `DELETE` statement with the `RETURNING BULK COLLECT INTO` clause to delete rows from a table and return them in two collections (nested tables).

Example 13-26 uses the keywords `OLD` and `NEW` to return the values of employee salaries before and after an `UPDATE` statement with the `RETURNING BULK COLLECT INTO` clause.

Example 13-25 Returning Deleted Rows in Two Nested Tables

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp AS
SELECT * FROM employees
ORDER BY employee_id;

DECLARE
  TYPE NumList IS TABLE OF employees.employee_id%TYPE;
  enums  NumList;
  TYPE NameList IS TABLE OF employees.last_name%TYPE;
  names  NameList;
BEGIN
  DELETE FROM emp_temp
  WHERE department_id = 30
  RETURNING employee_id, last_name
  BULK COLLECT INTO enums, names;

  DBMS_OUTPUT.PUT_LINE ('Deleted ' || SQL%ROWCOUNT || ' rows:');
  FOR i IN enums.FIRST .. enums.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Employee #' || enums(i) || ': ' || names(i));
  END LOOP;
END;
/
```

Result:

```
Deleted 6 rows:
Employee #114: Li
Employee #115: Khoo
Employee #116: Baida
Employee #117: Tobias
Employee #118: Himuro
Employee #119: Colmenares
```

Example 13-26 Returning NEW and OLD Values of Updated Rows

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp AS
SELECT * FROM employees
ORDER BY employee_id;

DECLARE
  TYPE SalList IS TABLE OF employees.salary%TYPE;
  old_sals SalList;
  new_sals SalList;
  TYPE NameList IS TABLE OF employees.last_name%TYPE;
  names NameList;
BEGIN
  UPDATE emp_temp SET salary = salary * 1.15
  WHERE salary < 2500
  RETURNING OLD salary, NEW salary, last_name
  BULK COLLECT INTO old_sals, new_sals, names;

  DBMS_OUTPUT.PUT_LINE('Updated ' || SQL%ROWCOUNT || ' rows: ');
  FOR i IN old_sals.FIRST .. old_sals.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE(names(i) || ': Old Salary $' || old_sals(i) ||
            ', New Salary $' || new_sals(i));
  END LOOP;
END;
/
```

Result:

```
Landry: Old Salary $2400, New Salary $2760
Markle: Old Salary $2200, New Salary $2530
Olson: Old Salary $2100, New Salary $2415
Gee: Old Salary $2400, New Salary $2760
Philtanker: Old Salary $2200, New Salary $2530
```

Using FORALL Statement and BULK COLLECT Clause Together

In a `FORALL` statement, the DML statement can have a `RETURNING BULK COLLECT INTO` clause. For each iteration of the `FORALL` statement, the DML statement stores the specified values in the specified collections—without overwriting the previous values, as the same DML statement would do in a `FOR LOOP` statement.

In Example 13-27, the `FORALL` statement runs a `DELETE` statement that has a `RETURNING BULK COLLECT INTO` clause. For each iteration of the `FORALL` statement, the `DELETE` statement stores the `employee_id` and `department_id` values of the deleted row in the collections `e_ids` and `d_ids`, respectively.

Example 13-28 is like Example 13-27 except that it uses a `FOR LOOP` statement instead of a `FORALL` statement.

Example 13-27 DELETE with RETURN BULK COLLECT INTO in FORALL Statement

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp AS
```

```
SELECT * FROM employees
ORDER BY employee_id, department_id;

DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  depts  NumList := NumList(10,20,30);

  TYPE enum_t IS TABLE OF employees.employee_id%TYPE;
  e_ids  enum_t;

  TYPE dept_t IS TABLE OF employees.department_id%TYPE;
  d_ids  dept_t;

BEGIN
  FORALL j IN depts.FIRST..depts.LAST
    DELETE FROM emp_temp
    WHERE department_id = depts(j)
    RETURNING employee_id, department_id
    BULK COLLECT INTO e_ids, d_ids;

  DBMS_OUTPUT.PUT_LINE ('Deleted ' || SQL%ROWCOUNT || ' rows:');

  FOR i IN e_ids.FIRST .. e_ids.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE (
      'Employee #' || e_ids(i) || ' from dept #' || d_ids(i)
    );
  END LOOP;
END;
/
```

Result:

```
Deleted 9 rows:
Employee #200 from dept #10
Employee #201 from dept #20
Employee #202 from dept #20
Employee #114 from dept #30
Employee #115 from dept #30
Employee #116 from dept #30
Employee #117 from dept #30
Employee #118 from dept #30
Employee #119 from dept #30
```

Example 13-28 DELETE with RETURN BULK COLLECT INTO in FOR LOOP Statement

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp AS
SELECT * FROM employees
ORDER BY employee_id, department_id;

DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  depts  NumList := NumList(10,20,30);
```

```
  TYPE enum_t IS TABLE OF employees.employee_id%TYPE;
  e_ids  enum_t;

  TYPE dept_t IS TABLE OF employees.department_id%TYPE;
  d_ids  dept_t;

BEGIN
  FOR j IN depts.FIRST..depts.LAST LOOP
    DELETE FROM emp_temp
    WHERE department_id = depts(j)
    RETURNING employee_id, department_id
    BULK COLLECT INTO e_ids, d_ids;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Deleted ' || SQL%ROWCOUNT || ' rows:');

  FOR i IN e_ids.FIRST .. e_ids.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE (
      'Employee #' || e_ids(i) || ' from dept #' || d_ids(i)
    );
  END LOOP;
END;
/
```

Result:

```
Deleted 6 rows:
Employee #114 from dept #30
Employee #115 from dept #30
Employee #116 from dept #30
Employee #117 from dept #30
Employee #118 from dept #30
Employee #119 from dept #30
```

Client Bulk-Binding of Host Arrays

Client programs (such as OCI and Pro*C programs) can use PL/SQL anonymous blocks to bulk-bind input and output host arrays. This is the most efficient way to pass collections to and from the database server.

In the client program, declare and assign values to the host variables to be referenced in the anonymous block. In the anonymous block, prefix each host variable name with a colon (:) to distinguish it from a PL/SQL collection variable name. When the client program runs, the database server runs the PL/SQL anonymous block.

In Example 13-29, the anonymous block uses a `FORALL` statement to bulk-bind a host input array. In the `FORALL` statement, the `DELETE` statement refers to four host variables: scalars `lower`, `upper`, and `emp_id` and array `depts`.

Example 13-29 Anonymous Block Bulk-Binds Input Host Array

```
BEGIN
  FORALL i IN :lower..:upper
    DELETE FROM employees
    WHERE department_id = :depts(i);
```

```
END;
/
```

Chaining Pipelined Table Functions for Multiple Transformations

Chaining pipelined table functions is an efficient way to perform multiple transformations on data.

> **Note:**
>
> You cannot run a pipelined table function over a database link. The reason is that the return type of a pipelined table function is a SQL user-defined type, which can be used only in a single database (as explained in *Oracle Database Object-Relational Developer's Guide*). Although the return type of a pipelined table function might appear to be a PL/SQL type, the database actually converts that PL/SQL type to a corresponding SQL user-defined type.

Topics

- Overview of Table Functions
- Creating Pipelined Table Functions
- Pipelined Table Functions as Transformation Functions
- Chaining Pipelined Table Functions
- Fetching from Results of Pipelined Table Functions
- Passing CURSOR Expressions to Pipelined Table Functions
- DML Statements on Pipelined Table Function Results
- NO_DATA_NEEDED Exception

Overview of Table Functions

A **table function** is a user-defined PL/SQL function that returns a collection of rows (an associative array, nested table or varray).

You can select from this collection as if it were a database table by invoking the table function inside the `TABLE` clause in a `SELECT` statement. The TABLE operator is optional.

For example:

```
SELECT * FROM TABLE(table_function_name(parameter_list))
```

Alternatively, the same query can be written without the TABLE operator as follow:

```
SELECT * FROM table_function_name(parameter_list)
```

A table function can take a collection of rows as input (that is, it can have an input parameter that is a nested table, varray, or cursor variable). Therefore, output from table function `tf1` can be input to table function `tf2`, and output from `tf2` can be input to table function `tf3`, and so on.

To improve the performance of a table function, you can:

- Enable the function for parallel execution, with the `PARALLEL_ENABLE` option.

  Functions enabled for parallel execution can run concurrently.
- Stream the function results directly to the next process, with Oracle Streams.

  Streaming eliminates intermediate staging between processes.
- Pipeline the function results, with the `PIPELINED` option.

  A **pipelined table function** returns a row to its invoker immediately after processing that row and continues to process rows. Response time improves because the entire collection need not be constructed and returned to the server before the query can return a single result row. (Also, the function needs less memory, because the object cache need not materialize the entire collection.)

  > **Caution:**
  >
  > A pipelined table function always references the current state of the data. If the data in the collection changes after the cursor opens for the collection, then the cursor reflects the changes. PL/SQL variables are private to a session and are not transactional. Therefore, read consistency, well known for its applicability to table data, does not apply to PL/SQL collection variables.

> **See Also:**
>
> - Chaining Pipelined Table Functions
> - *Oracle Database SQL Language Reference* for more information about the `TABLE` clause of the `SELECT` statement
> - *Oracle Database Data Cartridge Developer's Guide* for information about using pipelined and parallel table functions

Creating Pipelined Table Functions

A pipelined table function must be either a standalone function or a package function.

PIPELINED Option (Required)

For a standalone function, specify the `PIPELINED` option in the `CREATE FUNCTION` statement (for syntax, see "CREATE FUNCTION Statement"). For a package function, specify the `PIPELINED` option in both the function declaration and function definition (for syntax, see "Function Declaration and Definition").

PARALLEL_ENABLE Option (Recommended)

To improve its performance, enable the pipelined table function for parallel execution by specifying the `PARALLEL_ENABLE` option.

AUTONOMOUS_TRANSACTION Pragma

If the pipelined table function runs DML statements, then make it autonomous, with the `AUTONOMOUS_TRANSACTION` pragma (described in "AUTONOMOUS_TRANSACTION Pragma").

Then, during parallel execution, each instance of the function creates an independent transaction.

DETERMINISTIC Option (Recommended)

Multiple invocations of a pipelined table function, in either the same query or separate queries, cause multiple executions of the underlying implementation. If the function is deterministic, specify the `DETERMINISTIC` option, described in "DETERMINISTIC Clause".

Parameters

Typically, a pipelined table function has one or more cursor variable parameters. For information about cursor variables as function parameters, see "Cursor Variables as Subprogram Parameters".

> **See Also:**
>
> - "Cursor Variables" for general information about cursor variables
> - "Subprogram Parameters" for general information about subprogram parameters

RETURN Data Type

The data type of the value that a pipelined table function returns must be a collection type defined either at schema level or inside a package (therefore, it cannot be an associative array type). The elements of the collection type must be SQL data types, not data types supported only by PL/SQL (such as `PLS_INTEGER`). For information about collection types, see "Collection Types". For information about SQL data types, see *Oracle Database SQL Language Reference*.

You can use SQL data types `ANYTYPE`, `ANYDATA`, and `ANYDATASET` to dynamically encapsulate and access type descriptions, data instances, and sets of data instances of any other SQL type, including object and collection types. You can also use these types to create unnamed types, including anonymous collection types. For information about these types, see *Oracle Database PL/SQL Packages and Types Reference*.

PIPE ROW Statement

Inside a pipelined table function, use the `PIPE ROW` statement to return a collection element to the invoker without returning control to the invoker. See "PIPE ROW Statement" for its syntax and semantics.

RETURN Statement

As in every function, every execution path in a pipelined table function must lead to a `RETURN` statement, which returns control to the invoker. However, in a pipelined table function, a `RETURN` statement need not return a value to the invoker. See "RETURN Statement" for its syntax and semantics.

Example

Example 13-30 Creating and Invoking Pipelined Table Function

This example creates a package that includes a pipelined table function, `f1`, and then selects from the collection of rows that `f1` returns.

```
CREATE OR REPLACE PACKAGE pkg1 AUTHID DEFINER AS
  TYPE numset_t IS TABLE OF NUMBER;
  FUNCTION f1(x NUMBER) RETURN numset_t PIPELINED;
END pkg1;
/
```

Create a pipelined table function f1 that returns a collection of elements (1,2,3,... x).

```
CREATE OR REPLACE PACKAGE BODY pkg1 AS
  FUNCTION f1(x NUMBER) RETURN numset_t PIPELINED IS
  BEGIN
    FOR i IN 1..x LOOP
      PIPE ROW(i);
    END LOOP;
    RETURN;
  END f1;
END pkg1;
/
```

```
SELECT * FROM TABLE(pkg1.f1(5));
```

Result:

```
COLUMN_VALUE
------------
           1
           2
           3
           4
           5

5 rows selected.
```

```
SELECT * FROM pkg1.f1(2);
```

Result:

```
COLUMN_VALUE
------------
           1
           2
```

Pipelined Table Functions as Transformation Functions

A pipelined table function with a cursor variable parameter can serve as a transformation function. Using the cursor variable, the function fetches an input row. Using the `PIPE ROW` statement, the function pipes the transformed row or rows to the invoker. If the `FETCH` and `PIPE ROW` statements are inside a `LOOP` statement, the function can transform multiple input rows.

In Example 13-31, the pipelined table function transforms each selected row of the `employees` table to two nested table rows, which it pipes to the `SELECT` statement that invokes it. The actual parameter that corresponds to the formal cursor variable parameter is a `CURSOR` expression; for information about these, see "Passing CURSOR Expressions to Pipelined Table Functions".

Example 13-31 Pipelined Table Function Transforms Each Row to Two Rows

```
CREATE OR REPLACE PACKAGE refcur_pkg AUTHID DEFINER IS
  TYPE refcur_t IS REF CURSOR RETURN employees%ROWTYPE;
  TYPE outrec_typ IS RECORD (
    var_num    NUMBER(6),
    var_char1  VARCHAR2(30),
    var_char2  VARCHAR2(30)
  );
  TYPE outrecset IS TABLE OF outrec_typ;
  FUNCTION f_trans (p refcur_t) RETURN outrecset PIPELINED;
END refcur_pkg;
/

CREATE OR REPLACE PACKAGE BODY refcur_pkg IS
  FUNCTION f_trans (p refcur_t) RETURN outrecset PIPELINED IS
    out_rec outrec_typ;
    in_rec  p%ROWTYPE;
  BEGIN
    LOOP
      FETCH p INTO in_rec;  -- input row
      EXIT WHEN p%NOTFOUND;

      out_rec.var_num := in_rec.employee_id;
      out_rec.var_char1 := in_rec.first_name;
      out_rec.var_char2 := in_rec.last_name;
      PIPE ROW(out_rec);  -- first transformed output row

      out_rec.var_char1 := in_rec.email;
      out_rec.var_char2 := in_rec.phone_number;
      PIPE ROW(out_rec);  -- second transformed output row
    END LOOP;
    CLOSE p;
    RETURN;
  END f_trans;
END refcur_pkg;
/

SELECT * FROM TABLE (
  refcur_pkg.f_trans (
    CURSOR (SELECT * FROM employees WHERE department_id = 60)
```

```
  )
);
```

Result:

```
   VAR_NUM VAR_CHAR1                      VAR_CHAR2
---------- ------------------------------ ------------------------------
       103 Alexander                      James
       103 AJAMES                         1.590.555.0103
       104 Bruce                          Miller
       104 BMILLER                        1.590.555.0104
       105 David                          Williams
       105 DWILLIAMS                      1.590.555.0105
       106 Valli                          Jackson
       106 VJACKSON                       1.590.555.0106
       107 Diana                          Nguyen
       107 DNGUYEN                        1.590.555.0107

10 rows selected.
```

Chaining Pipelined Table Functions

To **chain** pipelined table functions `tf1` and `tf2` is to make the output of `tf1` the input of `tf2`. For example:

```
SELECT * FROM TABLE(tf2(CURSOR(SELECT * FROM TABLE(tf1()))));
```

The rows that `tf1` pipes out must be compatible actual parameters for the formal input parameters of `tf2`.

If chained pipelined table functions are enabled for parallel execution, then each function runs in a different process (or set of processes).

> **See Also:**
>
> "Passing CURSOR Expressions to Pipelined Table Functions"

Fetching from Results of Pipelined Table Functions

You can associate a named cursor with a query that invokes a pipelined table function. Such a cursor has no special fetch semantics, and such a cursor variable has no special assignment semantics.

However, the SQL optimizer does not optimize across PL/SQL statements. Therefore, in Example 13-32, the first PL/SQL statement is slower than the second—despite the overhead of running two SQL statements in the second PL/SQL statement, and even if function results are piped between the two SQL statements in the first PL/SQL statement.

In Example 13-32, assume that `f` and `g` are pipelined table functions, and that each function accepts a cursor variable parameter. The first PL/SQL statement associates cursor variable `r` with a query that invokes `f`, and then passes `r` to `g`. The second PL/SQL statement passes `CURSOR` expressions to both `f` and `g`.

See Also:

"Cursor Variables as Subprogram Parameters"

Example 13-32 Fetching from Results of Pipelined Table Functions

```
DECLARE
  r SYS_REFCURSOR;
  ...
  -- First PL/SQL statement (slower):
BEGIN
  OPEN r FOR SELECT * FROM TABLE(f(CURSOR(SELECT * FROM tab)));
  SELECT * BULK COLLECT INTO rec_tab FROM TABLE(g(r));

  -- NOTE: When g completes, it closes r.
END;

-- Second PL/SQL statement (faster):

SELECT * FROM TABLE(g(CURSOR(SELECT * FROM
  TABLE(f(CURSOR(SELECT * FROM tab))))));
/
```

Passing CURSOR Expressions to Pipelined Table Functions

As Example 13-32 shows, the actual parameter for the cursor variable parameter of a pipelined table function can be either a cursor variable or a `CURSOR` expression, and the latter is more efficient.

Note:

When a SQL `SELECT` statement passes a `CURSOR` expression to a function, the referenced cursor opens when the function begins to run and closes when the function completes.

See Also:

"CURSOR Expressions" for general information about `CURSOR` expressions

Example 13-33 creates a package that includes a pipelined table function with two cursor variable parameters and then invokes the function in a `SELECT` statement, using `CURSOR` expressions for actual parameters.

Example 13-34 uses a pipelined table function as an aggregate function, which takes a set of input rows and returns a single result. The `SELECT` statement selects the function result. (For information about the pseudocolumn `COLUMN_VALUE`, see *Oracle Database SQL Language Reference*.)

Example 13-33 Pipelined Table Function with Two Cursor Variable Parameters

```
CREATE OR REPLACE PACKAGE refcur_pkg AUTHID DEFINER IS
  TYPE refcur_t1 IS REF CURSOR RETURN employees%ROWTYPE;
  TYPE refcur_t2 IS REF CURSOR RETURN departments%ROWTYPE;
  TYPE outrec_typ IS RECORD (
    var_num    NUMBER(6),
    var_char1  VARCHAR2(30),
    var_char2  VARCHAR2(30)
  );
  TYPE outrecset IS TABLE OF outrec_typ;
  FUNCTION g_trans (p1 refcur_t1, p2 refcur_t2) RETURN outrecset PIPELINED;
END refcur_pkg;
/
```

```
CREATE OR REPLACE PACKAGE BODY refcur_pkg IS
  FUNCTION g_trans (
    p1 refcur_t1,
    p2 refcur_t2
  ) RETURN outrecset PIPELINED
  IS
    out_rec outrec_typ;
    in_rec1 p1%ROWTYPE;
    in_rec2 p2%ROWTYPE;
  BEGIN
    LOOP
      FETCH p2 INTO in_rec2;
      EXIT WHEN p2%NOTFOUND;
    END LOOP;
    CLOSE p2;
    LOOP
      FETCH p1 INTO in_rec1;
      EXIT WHEN p1%NOTFOUND;
      -- first row
      out_rec.var_num := in_rec1.employee_id;
      out_rec.var_char1 := in_rec1.first_name;
      out_rec.var_char2 := in_rec1.last_name;
      PIPE ROW(out_rec);
      -- second row
      out_rec.var_num := in_rec2.department_id;
      out_rec.var_char1 := in_rec2.department_name;
      out_rec.var_char2 := TO_CHAR(in_rec2.location_id);
      PIPE ROW(out_rec);
    END LOOP;
    CLOSE p1;
    RETURN;
  END g_trans;
END refcur_pkg;
/
```

```
SELECT * FROM TABLE (
  refcur_pkg.g_trans (
```

```
    CURSOR (SELECT * FROM employees WHERE department_id = 60),
    CURSOR (SELECT * FROM departments WHERE department_id = 60)
  )
);
```

Result:

```
   VAR_NUM VAR_CHAR1                      VAR_CHAR2
---------- ------------------------------ ------------------------------
       103 Alexander                      James
        60 IT                             1400
       104 Bruce                          Miller
        60 IT                             1400
       105 David                          Williams
        60 IT                             1400
       106 Valli                          Jackson
        60 IT                             1400
       107 Diana                          Nguyen
        60 IT                             1400

10 rows selected.
```

Example 13-34 Pipelined Table Function as Aggregate Function

```
DROP TABLE gradereport;
CREATE TABLE gradereport (
  student VARCHAR2(30),
  subject VARCHAR2(30),
  weight NUMBER,
  grade NUMBER
);

INSERT INTO gradereport (student, subject, weight, grade)
VALUES ('Mark', 'Physics', 4, 4);

INSERT INTO gradereport (student, subject, weight, grade)
VALUES ('Mark','Chemistry', 4, 3);

INSERT INTO gradereport (student, subject, weight, grade)
VALUES ('Mark','Maths', 3, 3);

INSERT INTO gradereport (student, subject, weight, grade)
VALUES ('Mark','Economics', 3, 4);



CREATE OR REPLACE PACKAGE pkg_gpa AUTHID DEFINER IS
  TYPE gpa IS TABLE OF NUMBER;
  FUNCTION weighted_average(input_values SYS_REFCURSOR)
    RETURN gpa PIPELINED;
END pkg_gpa;
/

CREATE OR REPLACE PACKAGE BODY pkg_gpa IS
  FUNCTION weighted_average (input_values SYS_REFCURSOR)
```

```
    RETURN gpa PIPELINED
  IS
    grade         NUMBER;
    total         NUMBER := 0;
    total_weight  NUMBER := 0;
    weight        NUMBER := 0;
  BEGIN
    LOOP
      FETCH input_values INTO weight, grade;
      EXIT WHEN input_values%NOTFOUND;
      total_weight := total_weight + weight;  -- Accumulate weighted average
      total := total + grade*weight;
    END LOOP;
    PIPE ROW (total / total_weight);
    RETURN; -- returns single result
  END weighted_average;
END pkg_gpa;
/
```

This query shows how the table function can be invoked without the optional TABLE operator.

```
SELECT w.column_value "weighted result"
FROM pkg_gpa.weighted_average (
    CURSOR (SELECT weight, grade FROM gradereport)
  ) w;
```

Result:

```
weighted result
---------------
            3.5

1 row selected.
```

DML Statements on Pipelined Table Function Results

The "table" that a pipelined table function returns cannot be the target table of a `DELETE`, `INSERT`, `UPDATE`, or `MERGE` statement. However, you can create a view of such a table and create `INSTEAD OF` triggers on the view. For information about `INSTEAD OF` triggers, see "INSTEAD OF DML Triggers".

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about the `CREATE VIEW` statement

NO_DATA_NEEDED Exception

You must understand the predefined exception `NO_DATA_NEEDED` in two cases:

- You include an `OTHERS` exception handler in a block that includes a `PIPE ROW` statement

- Your code that feeds a PIPE ROW statement must be followed by a clean-up procedure

  Typically, the clean-up procedure releases resources that the code no longer needs.

When the invoker of a pipelined table function needs no more rows from the function, the PIPE ROW statement raises NO_DATA_NEEDED. If the pipelined table function does not handle NO_DATA_NEEDED, as in Example 13-35, then the function invocation terminates but the invoking statement does not terminate. If the pipelined table function handles NO_DATA_NEEDED, its exception handler can release the resources that it no longer needs, as in Example 13-36.

In Example 13-35, the pipelined table function pipe_rows does not handle the NO_DATA_NEEDED exception. The SELECT statement that invokes pipe_rows needs only four rows. Therefore, during the fifth invocation of pipe_rows, the PIPE ROW statement raises the exception NO_DATA_NEEDED. The fifth invocation of pipe_rows terminates, but the SELECT statement does not terminate.

If the exception-handling part of a block that includes a PIPE ROW statement includes an OTHERS exception handler to handle unexpected exceptions, then it must also include an exception handler for the expected NO_DATA_NEEDED exception. Otherwise, the OTHERS exception handler handles the NO_DATA_NEEDED exception, treating it as an unexpected error. The following exception handler reraises the NO_DATA_NEEDED exception, instead of treating it as a irrecoverable error:

```
EXCEPTION
  WHEN NO_DATA_NEEDED THEN
    RAISE;
  WHEN OTHERS THEN
    -- (Put error-logging code here)
    RAISE_APPLICATION_ERROR(-20000, 'Irrecoverable error.');
END;
```

In Example 13-36, assume that the package External_Source contains these public items:

- Procedure Init, which allocates and initializes the resources that Next_Row needs
- Function Next_Row, which returns some data from a specific external source and raises the user-defined exception Done (which is also a public item in the package) when the external source has no more data
- Procedure Clean_Up, which releases the resources that Init allocated

The pipelined table function get_external_source_data pipes rows from the external source by invoking External_Source.Next_Row until either:

- The external source has no more rows.

  In this case, the External_Source.Next_Row function raises the user-defined exception External_Source.Done.

- get_external_source_data needs no more rows.

  In this case, the PIPE ROW statement in get_external_source_data raises the NO_DATA_NEEDED exception.

In either case, an exception handler in block b in get_external_source_data invokes External_Source.Clean_Up, which releases the resources that Next_Row was using.

Example 13-35 Pipelined Table Function Does Not Handle NO_DATA_NEEDED

```
CREATE TYPE t IS TABLE OF NUMBER
/
CREATE OR REPLACE FUNCTION pipe_rows RETURN t PIPELINED AUTHID DEFINER IS
```

```
  n NUMBER := 0;
BEGIN
  LOOP
    n := n + 1;
    PIPE ROW (n);
  END LOOP;
END pipe_rows;
/
SELECT COLUMN_VALUE
  FROM TABLE(pipe_rows())
  WHERE ROWNUM < 5
/
```

Result:

```
COLUMN_VALUE
------------
           1
           2
           3
           4

4 rows selected.
```

Example 13-36 Pipelined Table Function Handles NO_DATA_NEEDED

```
CREATE OR REPLACE FUNCTION get_external_source_data
  RETURN t PIPELINED AUTHID DEFINER IS
BEGIN
  External_Source.Init();           -- Initialize.
  <<b>> BEGIN
    LOOP                            -- Pipe rows from external source.
      PIPE ROW (External_Source.Next_Row());
    END LOOP;
  EXCEPTION
    WHEN External_Source.Done THEN  -- When no more rows are available,
      External_Source.Clean_Up();   --  clean up.
    WHEN NO_DATA_NEEDED THEN        -- When no more rows are needed,
      External_Source.Clean_Up();   --  clean up.
      RAISE NO_DATA_NEEDED;           -- Optional, equivalent to RETURN.
  END b;
END get_external_source_data;
/
```

Overview of Polymorphic Table Functions

Polymorphic table functions (PTF) are table functions whose operands can have more than one type. The return type is determined by the PTF invocation arguments list. The actual arguments to the table type usually determines the row output shape, but not always.

Introduction to Polymorphic Table Functions

Polymorphic Table Functions (PTF) are user-defined functions that can be invoked in the `FROM` clause of a SQL query block. They are capable of processing tables whose row type is not declared at definition time and producing a result table whose row type may or may not be declared at definition time. Polymorphic table functions leverage dynamic SQL capabilities to create powerful and complex custom functions. This is useful for applications demanding an interface with generic extensions which work for arbitrary input tables or queries.

A PTF author creates an interface to a procedural mechanism that defines a table. The PTF author defines, documents, and implements the PTF.

The query author can only describe the published interface and invoke the PTF function in queries.

The database is the PTF conductor. It manages the compilation and execution states of the PTF. The database and the PTF author can see a family of related SQL invoked procedures, called the PTF component procedures, and possibly additional private data (such as variables and cursors).

Types of Polymorphic Table Functions

The polymorphic table function type is specified based on their formal arguments list semantics:

- If an input `TABLE` argument has `Row Semantics`, the input is a single row.
- If an input `TABLE` argument has `Table Semantics`, the input is a set of rows. When a `Table Semantics` PTF is called from a query, the table argument can optionally be extended with either a `PARTITION BY` clause or an `ORDER BY` clause or both.

Polymorphic Table Function Definition

The PTF author defines, documents, and implements the Polymorphic Table Function (PTF).

A PTF has two parts:

1. The PL/SQL package which contains the client interface for the PTF implementation.

2. The standalone or package function naming the PTF and its associated implementation package.

Polymorphic Table Function Implementation

The Polymorphic Table Function (PTF) implementation client interface is a set of subprograms with fixed names that every PTF must provide.

Steps to Implement a Polymorphic Table Function

1. Create the implementation package containing the `DESCRIBE` function (required) and the `OPEN`, `FETCH_ROWS` , and `CLOSE` procedures (optional).
2. Create the function specification naming the PTF. The function can be created at the top-level after the package has been created, or as a package function in the implementation package (the package created in the first step). Polymorphic table functions do not have a function definition (a `FUNCTION BODY`), the definition is encapsulated in the associated implementation package.

   The function definition specifies :

   - The Polymorphic Table Function (PTF) name
   - Exactly one formal argument of type `TABLE` and any number of non `TABLE` arguments
   - The return type of the PTF as `TABLE`
   - The type of PTF function (`row` or `table semantics`)
   - The PTF implementation package name

> **See Also:**
>
> - *Oracle Database PL/SQL Packages and Types Reference* for information about a `DESCRIBE` Only polymorphic table function
> - *Oracle Database PL/SQL Packages and Types Reference* for more information about how to specify the PTF implementation package and use the `DBMS_TF` utilities
> - PIPELINED Clause for the standalone or package polymorphic table function creation syntax and semantic

Polymorphic Table Function Invocation

A polymorphic table function is invoked by specifying its name followed by the argument list in the `FROM` clause of a SQL query block.

The PTF arguments can be the standard scalar arguments that can be passed to a regular table function, but PTF's can additionally take a table argument. A table argument is either a `WITH` clause query or a schema-level object that is allowed in a `FROM` clause (such as tables, views, or table functions).

Syntax

table_argument ::= *table* [PARTITION BY *column_list*] [ORDER BY *order_column_list*]

column_list ::= *identifier* | (*identifier*[, *identifier*…])

order_column_list ::= *order_column_name* | (*order_column_name* [, *order_column_name*…])

order_column_name ::= *identifier* [ASC | DESC][NULLS FIRST | NULLS LAST]

Semantics

Each identifier is a column in the corresponding table.

The PTF has `Table Semantics`.

Query can optionally partition and order `Table Semantics` PTF input. This is disallowed for `Row Semantics` PTF input.

A polymorphic table function (PTF) cannot be the target of a DML statement. Any table argument of a PTF is passed in by name.

For example, the noop PTF can be used in a query such as :

```
SELECT *
FROM noop(emp);
```

or

```
WITH e AS
 (SELECT * FROM emp NATURAL JOIN dept)
SELECT t.* FROM noop(e) t;
```

The input table argument must be a basic table name.

The name resolution rules of the table identifier are (in priority order) as follows :

1. Identifier is resolved as a column name (such as a correlated column from an outer query block).
2. Identifier is resolved as a Common Table Expression (CTE) name in the current or some outer query-block. CTE is commonly known as the `WITH` clause.
3. Identifier is resolved as a schema-level table, view, or table-function (regular or polymorphic, and defined either at the schema-level or inside a package).

Many types of table expressions otherwise allowed in the `FROM` clause cannot be directly used as a table argument for a PTF (such as ANSI Joins, bind-variables, in-line views, `CURSOR` operators, `TABLE` operators). To use such table expressions as a PTF argument, these table expressions must be passed indirectly into a PTF by wrapping them in a CTE and then passing the CTE name into the PTF.

A PTF can be used as a table reference in the `FROM` clause and thus can be part of the ANSI Join and LATERAL syntax. Additionally, a PTF can be the source table for PIVOT/UNPIVOT and MATCH_RECOGNIZE. Some table modification clauses that are meant for tables and views (such as SAMPLING, PARTITION, CONTAINERS) are disallowed for PTF.

Direct function composition of PTF is allowed (such as nested PTF cursor expression invocation or PTF(TF()) nesting). However, nested PTF is disallowed (such as PTF(PTF()) nesting).

The scalar arguments of a PTF can be any SQL scalar expression. While the constant scalar values are passed as-is to the `DESCRIBE` function, all other values are passed as NULLs. This is usually not a problem for the PTF implementation if these values are not row shape determining, but otherwise the `DESCRIBE` function can raise an error; typically the documentation accompanying the PTF will state which scalar parameters, if any, are shape defining and thus must have constant non-null values. Note, that during query execution (during `OPEN`, `FETCH_ROWS`, `CLOSE`) the expressions are evaluated and their actual values are passed to these PTF execution procedures. The return type is determined by the PTF invocation arguments list.

Query arguments are passed to PTF using a `WITH` clause.

The `TABLE` operator is optional when the table function arguments list or empty list () appears.

Variadic Pseudo-Operators

A variadic pseudo-operator operates with a variable number of operands.

Starting with Oracle Database Release 18c, we introduce the concept of variadic pseudo-operator into the SQL expression language to support Polymorphic Table Functions (PTF). A pseudo-operator can be used to pass list of identifiers (such as column name) to a PTF. A pseudo-operator can only appear as arguments to PTFs, and are parsed by the SQL compiler like other SQL operators or PL/SQL function invocation. A pseudo-operator has a variable number of arguments but must have at least one. The pseudo-operator does not have any execution function associated with it, and they are completely removed from the SQL cursor after the PTF compilation is finished. During SQL compilation, the pseudo-operators are converted to corresponding DBMS_TF types and then passed to the `DESCRIBE` method. There is no output type associated with these operators. It is not possible to embed a pseudo-operator inside a general SQL expression.

COLUMNS Pseudo-Operator

You can use the COLUMNS pseudo-operator to specify arguments to a Polymorphic Table Function (PTF) invocation in the `FROM` clause of a SQL query block.

The `COLUMNS` pseudo-operator arguments specify the list of column names, or the list of column names with associated types.

Syntax

column_operator ::= COLUMNS (*column_list*)

column_list ::= *column_name_list* | *column_type_list*

column_name_list ::= *identifier* [, *identifier* ...]

column_type_list::= *identifier column_type* [, *identifier column_type*…]

Semantics

The `COLUMNS` pseudo-operator can only appear as an argument to a PTF. It cannot appear in any other SQL expression than the PTF expression itself.

The *column_type* must be a scalar type.

Polymorphic Table Function Compilation and Execution

The database fulfills the Polymorphic Table Functions (PTF) conductor role. As such, it is responsible for the PTF compilation, execution and its related states.

The database manages :

- The compilation state : This is the immutable state that is generated by `DESCRIBE` which is needed before execution.
- The execution state: This is the state used by the execution procedures of a `Table semantics` PTF.

.

> **See Also:**
>
> - *Oracle Database PL/SQL Packages and Types Reference* for more information about how the database manages the compilation and execution states of the PTFs

Polymorphic Table Function Optimization

A polymorphic table function (PTF) provides an efficient and scalable mechanism to extend the analytical capabilities of the database.

The key benefits are:

- Minimal data-movement: Only columns of interest are passed to PTF

- Predicates/Projections/Partitioning are/is pushed into underlying table/query (where semantically possible)
- Bulk data transfer into and out of PTF
- Parallelism is based on type of PTF and query specified partitioning (if any)

Skip_col Polymorphic Table Function Example

This PTF example demonstrates Row Semantics, Describe Only, package table function, and overloading features.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more Polymorphic Table Function (PTF) examples

Example 13-37 Skip_col Polymorphic Table Function Example

The skip_col Polymorphic Table Function (PTF) returns all the columns in a table except the columns specified in the PTF input argument. The skip_col PTF skips columns based on column names (overload 1) or columns data type (overload 2).

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at 18c Skip_col Polymorphic Table Function

Create the implementation package named skip_col_pkg containing the `DESCRIBE` function for the skip_col polymorphic table function (PTF). The `DESCRIBE` function is invoked to determine the row shape produced by the PTF. It returns a `DBMS_TF.DESCRIBE_T` table. It is overloaded. The `FETCH_ROWS` procedure is not required because it does need to produce associated new column values for a given subset of rows.

```
CREATE PACKAGE skip_col_pkg AS

  -- OVERLOAD 1: Skip by name --
  FUNCTION skip_col(tab TABLE,
                    col COLUMNS)
           RETURN TABLE PIPELINED ROW POLYMORPHIC USING skip_col_pkg;

  FUNCTION describe(tab IN OUT DBMS_TF.TABLE_T,
                    col        DBMS_TF.COLUMNS_T)
           RETURN DBMS_TF.DESCRIBE_T;

  -- OVERLOAD 2: Skip by type --
  FUNCTION skip_col(tab       TABLE,
                    type_name VARCHAR2,
                    flip      VARCHAR2 DEFAULT 'False')
           RETURN TABLE PIPELINED ROW POLYMORPHIC USING skip_col_pkg;
```

```
  FUNCTION describe(tab       IN OUT DBMS_TF.TABLE_T,
                    type_name        VARCHAR2,
                    flip             VARCHAR2 DEFAULT 'False')
           RETURN DBMS_TF.DESCRIBE_T;

END skip_col_pkg;
```

Create the implementation package body which contains the polymorphic table function definition.

```
CREATE PACKAGE BODY skip_col_pkg AS

/* OVERLOAD 1: Skip by name
 * Package PTF name:  skip_col_pkg.skip_col
 * Standalone PTF name: skip_col_by_name
 *
 * PARAMETERS:
 * tab - The input table
 * col - The name of the columns to drop from the output
 *
 * DESCRIPTION:
 *   This PTF removes all the input columns listed in col from the output
 *   of the PTF.
*/
 FUNCTION  describe(tab IN OUT DBMS_TF.TABLE_T,
                    col        DBMS_TF.COLUMNS_T)
            RETURN DBMS_TF.DESCRIBE_T
  AS
    new_cols DBMS_TF.COLUMNS_NEW_T;
    col_id   PLS_INTEGER := 1;
  BEGIN
    FOR i IN 1 .. tab.column.count() LOOP
      FOR j IN 1 .. col.count() LOOP
        tab.column(i).PASS_THROUGH := tab.column(i).DESCRIPTION.NAME !=
col(j);
        EXIT WHEN NOT tab.column(i).PASS_THROUGH;
      END LOOP;
    END LOOP;

    RETURN NULL;
  END;

/* OVERLOAD 2: Skip by type
 * Package PTF name:  skip_col_pkg.skip_col
 * Standalone PTF name: skip_col_by_type
 *
 * PARAMETERS:
 *   tab       - Input table
 *   type_name - A string representing the type of columns to skip
 *   flip      - 'False' [default] => Match columns with given type_name
 *               otherwise         => Ignore columns with given type_name
 *
 * DESCRIPTION:
 *   This PTF removes the given type of columns from the given table.
*/
```

```
  FUNCTION describe(tab       IN OUT DBMS_TF.TABLE_T,
                    type_name        VARCHAR2,
                    flip             VARCHAR2 DEFAULT 'False')
           RETURN DBMS_TF.DESCRIBE_T
  AS
    typ CONSTANT VARCHAR2(1024) := UPPER(TRIM(type_name));
  BEGIN
    FOR i IN 1 .. tab.column.count() LOOP
       tab.column(i).PASS_THROUGH :=
         CASE UPPER(SUBSTR(flip,1,1))
           WHEN 'F' THEN DBMS_TF.column_type_name(tab.column(i).DESCRIPTION)!
=typ
           ELSE          DBMS_TF.column_type_name(tab.column(i).DESCRIPTION)
=typ
         END /* case */;
    END LOOP;

    RETURN NULL;
  END;

END skip_col_pkg;
```

Create a standalone polymorphic table function named skip_col_by_name for overload 1. Specify exactly one formal argument of type `TABLE`, specify the return type of the PTF as `TABLE`, specify a `Row Semantics` PTF type, and indicate the PTF implementation package to use is skip_col_pkg.

```
CREATE FUNCTION skip_col_by_name(tab TABLE,
                                 col COLUMNS)
                  RETURN TABLE PIPELINED ROW POLYMORPHIC USING skip_col_pkg;
```

Create a standalone polymorphic table function named skip_col_by_type for overload 2. Specify exactly one formal argument of type `TABLE`, specify the return type of the PTF as `TABLE`, specify a `Row Semantics` PTF type, and indicate the PTF implementation package to use is skip_col_pkg.

```
CREATE FUNCTION skip_col_by_type(tab TABLE,
                                 type_name VARCHAR2,
                                 flip VARCHAR2 DEFAULT 'False')
                  RETURN TABLE PIPELINED ROW POLYMORPHIC USING skip_col_pkg;
```

Invoke the package skip_col PTF (overload 1) to report from the `SCOTT.DEPT` table only columns whose type is not `NUMBER`.

```
SELECT * FROM skip_col_pkg.skip_col(scott.dept, 'number');
```

```
DNAME          LOC
-------------- -------------
ACCOUNTING     NEW YORK
RESEARCH       DALLAS
SALES          CHICAGO
OPERATIONS     BOSTON
```

The same result can be achieved by invoking the standalone skip_col_by_type PTF to report from the `SCOTT.DEPT` table only columns whose type is not `NUMBER`.

```
SELECT * FROM skip_col_by_type(scott.dept, 'number');


DNAME          LOC
-------------- -------------
ACCOUNTING     NEW YORK
RESEARCH       DALLAS
SALES          CHICAGO
OPERATIONS     BOSTON
```

Invoke the package skip_col PTF (overload 2) to report from the `SCOTT.DEPT` table only columns whose type is `NUMBER`.

```
SELECT * FROM skip_col_pkg.skip_col(scott.dept, 'number', flip => 'True');


   DEPTNO
----------
        10
        20
        30
        40
```

The same result can be achieved by invoking the standalone skip_col_by_type PTF to report from the `SCOTT.DEPT` table only columns whose type is `NUMBER`.

```
SELECT * FROM skip_col_by_type(scott.dept, 'number', flip => 'True');


   DEPTNO
----------
        10
        20
        30
        40
```

Invoke the package skip_col PTF to report all employees in department 20 from the `SCOTT.EMP` table all columns except `COMM`, `HIREDATE` and `MGR`.

```
SELECT *
FROM skip_col_pkg.skip_col(scott.emp, COLUMNS(comm, hiredate, mgr))
WHERE deptno = 20;


    EMPNO ENAME      JOB              SAL     DEPTNO
---------- ---------- --------- ---------- ----------
      7369 SMITH      CLERK            800         20
      7566 JONES      MANAGER         2975         20
      7788 SCOTT      ANALYST         3000         20
      7876 ADAMS      CLERK           1100         20
      7902 FORD       ANALYST         3000         20
```

To_doc Polymorphic Table Function Example

The to_doc PTF example combines a list of specified columns into a single document column.

Example 13-38   To_doc Polymorphic Table Function Example

The to_doc PTF combines a list of columns into a document column constructed like a JSON object.

> **Live SQL:**
>
> You can view and run this example on Oracle Live SQL at 18c To_doc Polymorphic Table Function

Create the implementation package to_doc_p containing the DESCRIBE function and FETCH_ROWS procedure for the to_doc polymorphic table function (PTF).

The PTF parameters are :

- tab : The input table (The tab parameter is of type DBMS_TF.TABLE_T, a table descriptor record type)
- cols (optional) : The list of columns to convert to document. (The cols parameter is type DBMS_TF.COLUMNS_T , a column descriptor record type)

```
CREATE PACKAGE to_doc_p AS
   FUNCTION describe(tab      IN OUT DBMS_TF.TABLE_T,
                     cols     IN     DBMS_TF.COLUMNS_T DEFAULT NULL)
               RETURN DBMS_TF.DESCRIBE_T;

   PROCEDURE fetch_rows;
END to_doc_p;
```

Create the package containing the DESCRIBE function and FETCH_ROWS procedure. The FETCH_ROWS procedure is required to produce a new column named DOCUMENT in the output rowset. The DESCRIBE function indicates the read columns by annotating them in the input table descriptor, TABLE_T. Only the indicated read columns will be fetched and thus available for processing during FETCH_ROWS. The PTF invocation in a query can use the COLUMNS pseudo-operator to indicate which columns the query wants the PTF to read, and this information is passed to the DESCRIBE function which then in turn sets the COLUMN_T.FOR_READ boolean flag. Only scalar SQL data types are allowed for the read columns. The COLUMN_T.PASS_THROUGH boolean flag indicates columns that are passed from the input table of the PTF to the output, without any modifications.

```
CREATE PACKAGE BODY to_doc_p AS

FUNCTION describe(tab      IN OUT DBMS_TF.TABLE_T,
                  cols     IN     DBMS_TF.COLUMNS_T DEFAULT NULL)
            RETURN DBMS_TF.DESCRIBE_T AS
BEGIN
  FOR i IN 1 .. tab.column.count LOOP
     CONTINUE WHEN NOT DBMS_TF.SUPPORTED_TYPE(tab.column(i).DESCRIPTION.TYPE);

      IF cols IS NULL THEN
         tab.column(i).FOR_READ     := TRUE;
         tab.column(i).PASS_THROUGH := FALSE;
         CONTINUE;
       END IF;
```

```
      FOR j IN 1 .. cols.count LOOP
        IF (tab.column(i).DESCRIPTION.NAME = cols(j)) THEN
            tab.column(i).FOR_READ     := TRUE;
            tab.column(i).PASS_THROUGH := FALSE;
        END IF;
      END LOOP;

  END LOOP;

  RETURN DBMS_TF.describe_t(new_columns => DBMS_TF.COLUMNS_NEW_T(1 =>
                              DBMS_TF.COLUMN_METADATA_T(name
=>'DOCUMENT')));
END;

 PROCEDURE fetch_rows AS
      rst DBMS_TF.ROW_SET_T;
      col DBMS_TF.TAB_VARCHAR2_T;
      rct PLS_INTEGER;
 BEGIN
      DBMS_TF.GET_ROW_SET(rst, row_count => rct);
      FOR rid IN 1 .. rct LOOP
           col(rid) := DBMS_TF.ROW_TO_CHAR(rst, rid);
      END LOOP;
      DBMS_TF.PUT_COL(1, col);
 END;

END to_doc_p;
```

Create the standalone to_doc PTF. Specify exactly one formal argument of type `TABLE`, specify the return type of the PTF as `TABLE`, specify a Row Semantics PTF type, and indicate the PTF implementation package to use is to_doc_p.

```
CREATE FUNCTION to_doc(
                 tab  TABLE,
                       cols  COLUMNS DEFAULT NULL)
                       RETURN TABLE
    PIPELINED ROW POLYMORPHIC USING to_doc_p;
```

Invoke the to_doc PTF to display all columns of table `SCOTT.DEPT` as one combined `DOCUMENT` column.

```
SELECT * FROM to_doc(scott.dept);
```

```
DOCUMENT
---------------------------------------------------
{"DEPTNO":10, "DNAME":"ACCOUNTING", "LOC":"NEW YORK"}
{"DEPTNO":20, "DNAME":"RESEARCH", "LOC":"DALLAS"}
{"DEPTNO":30, "DNAME":"SALES", "LOC":"CHICAGO"}
{"DEPTNO":40, "DNAME":"OPERATIONS", "LOC":"BOSTON"}
```

For all employees in departments 10 and 30, display the `DEPTNO`, `ENAME` and `DOCUMENT` columns ordered by `DEPTNO` and `ENAME`. Invoke the to_doc PTF with the `COLUMNS` pseudo-operator to select columns EMPNO, JOB, MGR, HIREDATE, SAL and COMM of table SCOTT.EMP . The PTF combines these columns into the DOCUMENT column.

```
SELECT deptno, ename, document
FROM   to_doc(scott.emp, COLUMNS(empno,job,mgr,hiredate,sal,comm))
WHERE  deptno IN (10, 30)
ORDER BY 1, 2;
```

```
DEPTNO ENAME      DOCUMENT
------ ----------
---------------------------------------------------------------------------------
    10 CLARK      {"EMPNO":7782, "JOB":"MANAGER", "MGR":7839, "HIREDATE":"09-JUN-81",
"SAL":2450}
    10 KING       {"EMPNO":7839, "JOB":"PRESIDENT", "HIREDATE":"17-NOV-81", "SAL":5000}
    10 MILLER     {"EMPNO":7934, "JOB":"CLERK", "MGR":7782, "HIREDATE":"23-JAN-82",
"SAL":1300}
    30 ALLEN      {"EMPNO":7499, "JOB":"SALESMAN", "MGR":7698, "HIREDATE":"20-FEB-81",
"SAL":1600, "COMM":300}
    30 BLAKE      {"EMPNO":7698, "JOB":"MANAGER", "MGR":7839, "HIREDATE":"01-MAY-81",
"SAL":2850}
    30 JAMES      {"EMPNO":7900, "JOB":"CLERK", "MGR":7698, "HIREDATE":"03-DEC-81",
"SAL":950}
    30 MARTIN     {"EMPNO":7654, "JOB":"SALESMAN", "MGR":7698, "HIREDATE":"28-SEP-81",
"SAL":1250, "COMM":1400}
    30 TURNER     {"EMPNO":7844, "JOB":"SALESMAN", "MGR":7698, "HIREDATE":"08-SEP-81",
"SAL":1500, "COMM":0}
    30 WARD       {"EMPNO":7521, "JOB":"SALESMAN", "MGR":7698, "HIREDATE":"22-FEB-81",
"SAL":1250, "COMM":500}
```

With the subquery named E, display the DOC_ID and DOCUMENT columns. Report all clerk employees, their salary, department and department location. Use the to_doc PTF to combine the NAME, SAL, DEPTNO and LOC columns into the DOCUMENT column.

```
WITH e AS (
       SELECT ename name, sal, deptno, loc
         FROM scott.emp NATURAL JOIN scott.dept
        WHERE job = 'CLERK')
      SELECT ROWNUM doc_id, t.*
        FROM to_doc(e) t;
```

```
    DOC_ID DOCUMENT
---------- -------------------------------------------------------
         1 {"NAME":"MILLER", "SAL":1300, "DEPTNO":10, "LOC":"NEW YORK"}
         2 {"NAME":"SMITH", "SAL":800, "DEPTNO":20, "LOC":"DALLAS"}
         3 {"NAME":"ADAMS", "SAL":1100, "DEPTNO":20, "LOC":"DALLAS"}
         4 {"NAME":"JAMES", "SAL":950, "DEPTNO":30, "LOC":"CHICAGO"}
```

Use a subquery block to display c1, c2, c3 column values converted into the DOCUMENT column.

```
WITH t(c1,c2,c3)  AS (
    SELECT NULL, NULL, NULL FROM dual
    UNION ALL
    SELECT    1, NULL, NULL FROM dual
    UNION ALL
    SELECT NULL,    2, NULL FROM dual
    UNION ALL
    SELECT    0, NULL,    3 FROM dual)
```

```
  SELECT *
    FROM to_doc(t);

DOCUMENT
---------------
{}
{"C1":1}
{"C2":2}
{"C1":0, "C3":3}
```

For all employees in department 30, display the values of the member with property names `ENAME` and `COMM`. The PTF invocation reporting from the `SCOTT.EMP` table produces the `DOCUMENT` column which can be used as input to the `JSON_VALUE` function. This function selects a scalar value from some JSON data.

```
SELECT JSON_VALUE(document, '$.ENAME') ename,
       JSON_VALUE(document, '$.COMM')  comm
FROM   to_doc(scott.emp)
WHERE  JSON_VALUE(document, '$.DEPTNO') = 30;

ENAME      COMM
---------- ----
ALLEN      300
WARD       500
MARTIN     1400
BLAKE
TURNER     0
JAMES
```

Implicit_echo Polymorphic Table Function Example

The implicit_echo PTF example demonstrates that the USING clause is optional when the Polymorphic Table Function and the DESCRIBE function are defined in the same package.

Example 13-39 Implicit_echo Polymorphic Table Function Example

The `implicit_echo` PTF, takes in a table and a column and produces a new column with the same value.

This PTF returns the column in the input table tab, and adds to it the column listed in cols but with the column names prefixed with "`ECHO_`".

Create the implementation package `implicit_echo_package` containing the `DESCRIBE` function, `implicit_echo` polymorphic table function (PTF) and `FETCH_ROWS` procedure.

```
CREATE PACKAGE implicit_echo_package AS
  prefix   DBMS_ID := '"ECHO_';

  FUNCTION DESCRIBE(tab   IN OUT DBMS_TF.TABLE_T,
                    cols  IN     DBMS_TF.COLUMNS_T)
           RETURN DBMS_TF.DESCRIBE_T;

  PROCEDURE FETCH_ROWS;

  -- PTF FUNCTION: WITHOUT USING CLAUSE --
  FUNCTION implicit_echo(tab TABLE, cols COLUMNS)
           RETURN TABLE PIPELINED ROW POLYMORPHIC;
```

```
END implicit_echo_package;
```

Create the package containing the `DESCRIBE` function containing the input table parameter and the column parameter to be read. This function is invoked to determine the type of rows produced by the Polymorphic Table Function. The function returns a table `DBMS_TF.DESCRIBE_T`. The `FETCH_ROWS` procedure is required to produce the indicated read column along with a new column prefixed with "`ECHO_`" in the output rowset. The `implicit_echo` is the PTF function and contains two arguments, `tab` and `cols`, whose values are obtained from the query and this information is passed to the `DESCRIBE` function. The Row semantics specifies a PTF type but without the `USING` clause. This function is invoked from the SQL query.

Create the implementation package body `implicit_echo_package` which contains the PTF definition.

```
CREATE PACKAGE BODY implicit_echo_package AS

FUNCTION DESCRIBE(tab  IN  OUT DBMS_TF.TABLE_T,
                  cols IN      DBMS_TF.COLUMNS_T)
          RETURN DBMS_TF.DESCRIBE_T
AS
  new_cols DBMS_TF.COLUMNS_NEW_T;
  col_id   PLS_INTEGER := 1;

BEGIN
 FOR i in 1 .. tab.column.COUNT LOOP

   FOR j in 1 .. cols.COUNT LOOP

     IF (tab.column(i).description.name = cols(j)) THEN

       IF (NOT DBMS_TF.SUPPORTED_TYPE(tab.column(i).description.type)) THEN
            RAISE_APPLICATION_ERROR(-20102, 'Unsupported column type['||
                                    tab.column(i).description.type||']');
       END IF;

       tab.column(i).for_read := TRUE;
       new_cols(col_id)       := tab.column(i).description;
       new_cols(col_id).name  := prefix||

REGEXP_REPLACE(tab.column(i).description.name,

'^"|"$');
       col_id                 := col_id + 1;
       EXIT;

     END IF;

    END LOOP;

 END LOOP;

/* VERIFY ALL COLUMNS WERE FOUND */
 IF (col_id - 1 != cols.COUNT) then
```

```
    RAISE_APPLICATION_ERROR(-20101,'Column mismatch['||col_id-1||'],
                                                ['||cols.COUNT||']');
 END IF;

 RETURN DBMS_TF.DESCRIBE_T(new_columns => new_cols);

END;

 PROCEDURE FETCH_ROWS AS
     rowset DBMS_TF.ROW_SET_T;
 BEGIN
         DBMS_TF.GET_ROW_SET(rowset);
         DBMS_TF.PUT_ROW_SET(rowset);
 END;

END implicit_echo_package;
```

Invoke the PTF to display ENAME column of table `SCOTT.EMP` and display it along with another column `ECHO_ENAME` having the same value.

```
SELECT ENAME, ECHO_ENAME
FROM implicit_echo_package.implicit_echo(SCOTT.EMP, COLUMNS(SCOTT.ENAME));


ENAME      ECHO_ENAME
---------- ----------
SMITH      SMITH
ALLEN      ALLEN
WARD       WARD
JONES      JONES
MARTIN     MARTIN
BLAKE      BLAKE
CLARK      CLARK
SCOTT      SCOTT
KING       KING
TURNER     TURNER
ADAMS      ADAMS
JAMES      JAMES
FORD       FORD
MILLER     MILLER
```

Updating Large Tables in Parallel

The `DBMS_PARALLEL_EXECUTE` package lets you incrementally update the data in a large table in parallel, in two high-level steps:

1. Group sets of rows in the table into smaller chunks.
2. Apply the desired `UPDATE` statement to the chunks in parallel, committing each time you have finished processing a chunk.

This technique is recommended whenever you are updating a lot of data. Its advantages are:

- You lock only one set of rows at a time, for a relatively short time, instead of locking the entire table.
- You do not lose work that has been done if something fails before the entire operation finishes.

- You reduce rollback space consumption.
- You improve performance.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_PARALLEL_EXECUTE` package

Collecting Data About User-Defined Identifiers

PL/Scope extracts, organizes, and stores data about PL/SQL and SQL identifiers and SQL statements from PL/SQL source text. You can retrieve the identifiers and statements data with the static data dictionary views `*_IDENTIFIERS` and *_STATEMENTS.

> **See Also:**
>
> - PL/SQL Units and Compilation Parameters for more information about PLSQL_SETTINGS parameter
> - *Oracle Database Development Guide* for more information about using PL/Scope

Profiling and Tracing PL/SQL Programs

To help you isolate performance problems in large PL/SQL programs, PL/SQL provides these tools, implemented as PL/SQL packages.

Table 13-1 Profiling and Tracing Tools Summary

Tool	Package	Description
Profiler interface	`DBMS_PROFILER`	Computes the time that your PL/SQL program spends at each line and in each subprogram. You must have `CREATE` privileges on the units to be profiled. Saves runtime statistics in database tables, which you can query.
Trace interface	`DBMS_TRACE`	Traces the order in which subprograms run. You can specify the subprograms to trace and the tracing level. Saves runtime statistics in database tables, which you can query.

Table 13-1 (Cont.) Profiling and Tracing Tools Summary

Tool	Package	Description
PL/SQL hierarchical profiler	DBMS_HPROF	Reports the dynamic execution program profile of your PL/SQL program, organized by subprogram invocations. Accounts for SQL and PL/SQL execution times separately. Requires no special source or compile-time preparation. Generates reports in HTML. Provides the option of storing profiler data and results in relational format in database tables for custom report generation (such as third-party tools offer).
SQL trace	DBMS_APPLICATION_INFO	Uses the DBMS_APPLICATION_INFO package with Oracle Trace and the SQL trace facility to record names of executing modules or transactions in the database for later use when tracking the performance of various modules and debugging.
PL/SQL Basic Block Coverage	DBMS_PLSQL_CODE_COVERAGE	Collects and analyzes basic block coverage data.
Call Stack Utilities	UTL_CALL_STACK	Provides information about currently executing subprograms (such as subprogram names, unit names, owner names, edition names, and error stack information) that you can use to create more revealing error logs and application execution traces.

Related Topics

- *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_APPLICATION_INFO package
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_HPROF package
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_PLSQL_CODE_COVERAGE package
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_PROFILER package
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_TRACE package

  Oracle Database PL/SQL Packages and Types Reference for more information about the UTL_CALL_STACK package
- COVERAGE Pragma for the syntax and semantics of COVERAGE PRAGMA
- *Oracle Database Development Guide* for more information about using PL/SQL basic block coverage
- *Oracle Database Development Guide* for a detailed description of PL/SQL hierarchical profiler
- *Oracle Database Development Guide* for more information about analyzing and debugging stored subprograms

Compiling PL/SQL Units for Native Execution

You can usually speed up PL/SQL units by compiling them into native code (processor-dependent system code), which is stored in the SYSTEM tablespace.

You can natively compile any PL/SQL unit of any type, including those that Oracle Database supplies.

Natively compiled program units work in all server environments, including shared server configuration (formerly called "multithreaded server") and Oracle Real Application Clusters (Oracle RAC).

On most platforms, PL/SQL native compilation requires no special set-up or maintenance. On some platforms, the DBA might want to do some optional configuration.

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for information about configuring a database
> - Platform-specific configuration documentation for your platform

You can test to see how much performance gain you can get by enabling PL/SQL native compilation.

If you have determined that PL/SQL native compilation will provide significant performance gains in database operations, Oracle recommends compiling the entire database for native mode, which requires DBA privileges. This speeds up both your own code and calls to the PL/SQL packages that Oracle Database supplies.

Topics

- Determining Whether to Use PL/SQL Native Compilation
- How PL/SQL Native Compilation Works
- Dependencies, Invalidation, and Revalidation
- Setting Up a New Database for PL/SQL Native Compilation*
- Compiling the Entire Database for PL/SQL Native or Interpreted Compilation*

* Requires DBA privileges.

Determining Whether to Use PL/SQL Native Compilation

Whether to compile a PL/SQL unit for native or interpreted mode depends on where you are in the development cycle and on what the program unit does.

While you are debugging program units and recompiling them frequently, interpreted mode has these advantages:

- You can use PL/SQL debugging tools on program units compiled for interpreted mode (but not for those compiled for native mode).
- Compiling for interpreted mode is faster than compiling for native mode.

After the debugging phase of development, in determining whether to compile a PL/SQL unit for native mode, consider:

- PL/SQL native compilation provides the greatest performance gains for computation-intensive procedural operations. Examples are data warehouse applications and applications with extensive server-side transformations of data for display.
- PL/SQL native compilation provides the least performance gains for PL/SQL subprograms that spend most of their time running SQL.
- When many program units (typically over 15,000) are compiled for native execution, and are simultaneously active, the large amount of shared memory required might affect system performance.

How PL/SQL Native Compilation Works

Without native compilation, the PL/SQL statements in a PL/SQL unit are compiled into an intermediate form, system code, which is stored in the catalog and interpreted at run time.

With PL/SQL native compilation, the PL/SQL statements in a PL/SQL unit are compiled into native code and stored in the catalog. The native code need not be interpreted at run time, so it runs faster.

Because native compilation applies only to PL/SQL statements, a PL/SQL unit that uses only SQL statements might not run faster when natively compiled, but it does run at least as fast as the corresponding interpreted code. The compiled code and the interpreted code make the same library calls, so their action is the same.

The first time a natively compiled PL/SQL unit runs, it is fetched from the SYSTEM tablespace into shared memory. Regardless of how many sessions invoke the program unit, shared memory has only one copy it. If a program unit is not being used, the shared memory it is using might be freed, to reduce memory load.

Natively compiled subprograms and interpreted subprograms can invoke each other.

PL/SQL native compilation works transparently in an Oracle Real Application Clusters (Oracle RAC) environment.

The `PLSQL_CODE_TYPE` compilation parameter determines whether PL/SQL code is natively compiled or interpreted. For information about this compilation parameters, see "PL/SQL Units and Compilation Parameters".

Dependencies, Invalidation, and Revalidation

Recompilation is automatic with invalidated PL/SQL modules. For example, if an object on which a natively compiled PL/SQL subprogram depends changes, the subprogram is invalidated. The next time the same subprogram is called, the database recompiles the subprogram automatically. Because the `PLSQL_CODE_TYPE` setting is stored inside the library unit for each subprogram, the automatic recompilation uses this stored setting for code type.

Explicit recompilation does not necessarily use the stored `PLSQL_CODE_TYPE` setting. For the conditions under which explicit recompilation uses stored settings, see "PL/SQL Units and Compilation Parameters".

Setting Up a New Database for PL/SQL Native Compilation

If you have DBA privileges, you can set up a new database for PL/SQL native compilation by setting the compilation parameter `PLSQL_CODE_TYPE` to `NATIVE`. The performance benefits apply to the PL/SQL packages that Oracle Database supplies, which are used for many database operations.

> **Note:**
>
> If you compile the whole database as `NATIVE`, Oracle recommends that you set `PLSQL_CODE_TYPE` at the system level.

Compiling the Entire Database for PL/SQL Native or Interpreted Compilation

If you have DBA privileges, you can recompile all PL/SQL modules in an existing database to `NATIVE` or `INTERPRETED`, using the `dbmsupgnv.sql` and `dbmsupgin.sql` scripts respectively during the process explained in this section. Before making the conversion, review "Determining Whether to Use PL/SQL Native Compilation".

> **Note:**
>
> - If you compile the whole database as `NATIVE`, Oracle recommends that you set `PLSQL_CODE_TYPE` at the system level.
> - If Database Vault is enabled, then you can run `dbmsupgnv.sql` only if the Database Vault administrator has granted you the `DV_PATCH_ADMIN` role.
> - The conversion process described here affects only the current container's units. Units in other containers are not affected.

During the conversion to native compilation, `TYPE` specifications are not recompiled by `dbmsupgnv.sql` to `NATIVE` because these specifications do not contain executable code.

Package specifications seldom contain executable code so the runtime benefits of compiling to `NATIVE` are not measurable. You can use the `TRUE` command-line parameter with the `dbmsupgnv.sql` script to exclude package specs from recompilation to `NATIVE`, saving time in the conversion process.

When converting to interpreted compilation, the `dbmsupgin.sql` script does not accept any parameters and does not exclude any PL/SQL units.

> **Note:**
>
> The following procedure describes the conversion to native compilation. If you must recompile all PL/SQL modules to interpreted compilation, make these changes in the steps.
>
> - Skip the first step.
> - Set the `PLSQL_CODE_TYPE` compilation parameter to `INTERPRETED` rather than `NATIVE`.
> - Substitute `dbmsupgin.sql` for the `dbmsupgnv.sql` script.

1. Ensure that a test PL/SQL unit can be compiled. For example:

   ```
   ALTER PROCEDURE my_proc COMPILE PLSQL_CODE_TYPE=NATIVE REUSE SETTINGS;
   ```

2. Shut down application services, the listener, and the database.
   - Shut down all of the Application services including the Forms Processes, Web Servers, Reports Servers, and Concurrent Manager Servers. After shutting down all of the Application services, ensure that all of the connections to the database were terminated.
   - Shut down the TNS listener of the database to ensure that no new connections are made.
   - Shut down the database in normal or immediate mode as the user `SYS`. See *Oracle Database Administrator's Guide*.

3. Set `PLSQL_CODE_TYPE` to `NATIVE` in the compilation parameter file. If the database is using a server parameter file, then set this after the database has started.

   The value of `PLSQL_CODE_TYPE` does not affect the conversion of the PL/SQL units in these steps. However, it does affect all subsequently compiled units, so explicitly set it to the desired compilation type.

4. Start up the database in upgrade mode, using the `UPGRADE` option. For information about SQL*Plus `STARTUP`, see *SQL*Plus User's Guide and Reference*.

5. Run this code to list the invalid PL/SQL units. You can save the output of the query for future reference with the SQL `SPOOL` statement:

   ```
   -- To save the output of the query to a file:
     SPOOL pre_update_invalid.log
   SELECT o.OWNER, o.OBJECT_NAME, o.OBJECT_TYPE
   FROM DBA_OBJECTS o, DBA_PLSQL_OBJECT_SETTINGS s
   WHERE o.OBJECT_NAME = s.NAME AND o.STATUS='INVALID';
   -- To stop spooling the output: SPOOL OFF
   ```

   If any Oracle supplied units are invalid, try to validate them by recompiling them. For example:

   ```
   ALTER PACKAGE SYS.DBMS_OUTPUT COMPILE BODY REUSE SETTINGS;
   ```

   If the units cannot be validated, save the spooled log for future resolution and continue.

6. Run this query to determine how many objects are compiled `NATIVE` and `INTERPRETED` (to save the output, use the SQL `SPOOL` statement):

   ```
   SELECT TYPE, PLSQL_CODE_TYPE, COUNT(*)
   FROM DBA_PLSQL_OBJECT_SETTINGS
   WHERE PLSQL_CODE_TYPE IS NOT NULL AND ORIGIN_CON_ID=SYS_CONTEXT('USERENV',
   'CON_ID')
   GROUP BY TYPE, PLSQL_CODE_TYPE
   ORDER BY TYPE, PLSQL_CODE_TYPE;
   ```

   Any objects with a `NULL` `plsql_code_type` are special internal objects and can be ignored.

7. Run the `$ORACLE_HOME/rdbms/admin/dbmsupgnv.sql` script as the user `SYS` to update the `plsql_code_type` setting to `NATIVE` in the dictionary tables for all PL/SQL units. This process also invalidates the units. Use `TRUE` with the script to exclude package specifications; `FALSE` to include the package specifications.

This update must be done when the database is in `UPGRADE` mode. The script is guaranteed to complete successfully or rollback all the changes.

8. Shut down the database and restart in `NORMAL` mode.
9. Before you run the `utlrp.sql` script, Oracle recommends that no other sessions are connected to avoid possible problems. You can ensure this with this statement:

   ```
   ALTER SYSTEM ENABLE RESTRICTED SESSION;
   ```

10. Run the `$ORACLE_HOME/rdbms/admin/utlrp.sql` script as the user `SYS`. This script recompiles all the PL/SQL modules using a default degree of parallelism. See the comments in the script for information about setting the degree explicitly.

    If for any reason the script is terminated atypically, rerun the `utlrp.sql` script to recompile any remaining invalid PL/SQL modules.

11. After the compilation completes successfully, verify that there are no invalid PL/SQL units using the query in step 5. You can spool the output of the query to the `post_upgrade_invalid.log` file and compare the contents with the `pre_upgrade_invalid.log` file, if it was created previously.
12. Re-run the query in step 6. If recompiling with `dbmsupgnv.sql`, confirm that all PL/SQL units, except `TYPE` specifications and package specifications if excluded, are `NATIVE`. If recompiling with `dbmsupgin.sql`, confirm that all PL/SQL units are `INTERPRETED`.
13. Disable the restricted session mode for the database, then start the services that you previously shut down. To disable restricted session mode, use this statement:

    ```
    ALTER SYSTEM DISABLE RESTRICTED SESSION;
    ```