**Collections**

(Notes #6a)

Topics:

1. Introduction
2. Associative Array (PL/SQL table)

Define, Variable, Populating and Referencing; diff index datatypes

1. Collection Constructors (p5)
2. Varray (p5)
3. Nested Table (p5)
4. Collection Methods (p8)
5. Bulk Collect (p13)
6. Collection Comparisons (p14)

Appendix A, review the methods (p16)

Appendix B, Optional notes about BULK SQL (page 17)

1. **Introduction of the concept of Collections**

PL/SQL provides two kind *composite* data types (with *internal* components):

Collection and Record.

In Collection, the internal components always have the same data type, and are called elements. The way to access the element is by variable\_name (index/subscript).

These are similar to the array in traditional language. Elements are homogeneous, also called one dimensional. Also said, it likes a two-column table, first column is for index, second are for values of elements.

There are three kinds of Collection, listed below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Collection**  **Type** | **Number of**  **Elements** | **Index**  **Type** | **Dense or Sparse** | Uninitialized **Status** | **Where defined** |
| Associative Array (Index-by Table) | Unspecified  (as DB table) | PLS\_INTEGER  Or  STRING | Either | Empty  (as DB table) | In PL/SQL Block  or Package |
| VARRAY  (variable-size Array) | Specified | INTEGER | Always Dense | Null (not exist) refer to p6 | In PL/SQL Block  or Package  or at Schema level |
| Nested Table | Unspecified | INTEGER | Starts dense, can become sparse | Null | In PL/SQL Block  or Package  or at Schema level |

Notes:

Varray and Nested table will be stored in database, if CREATE in DB,

Associative Array can only work in PL/SQL code.

# Associative Arrays (Available after Oracle version 9)

It was called PL/SQL Table (Oracle 7, 1992),

Also was called **index-by table** (version 8, 1997)

After version 9 (2001), the name is changed to associative Arrays, as INDEX BY syntax (besides Binary\_Integer) could be used to “associate” or index by VARCHAR2 (or by PLS\_INTEGER).

Associative Array is a set of **key-value** pairs. Each key is a unique index, used to locate the associated value with the syntax *variable\_name* (*index*). Indexes are stored in sorted order. (Oracle does not enforce the index value to be unique; if you have duplicated values of index, it will be difficult to handle the element values).

Associative Array is specified in PL/SQL, cannot be manipulated with DML statements.

* 1. **Defining an associative array type & Declaring associative array variables**

The Syntax:

TYPE type\_name IS **TABLE** OF Element\_datatype [NOT NULL]

INDEX BY {BINARY\_INTEGER | PLS\_INTEGER | VARCHAR2 (size\_limit)};

**Example**

TYPE EnameTabType IS TABLE OF VARCHAR2(10) /\* datatype \*/

INDEX BY BINARY\_INTEGER;

Or

TYPE EnameTabType IS TABLE OF emp.ename%TYPE /\* or use %type \*/

INDEX BY BINARY\_INTEGER;

ename\_tab EnameTabType;

/\* ename\_tab represents an entire PL/SQL table. \*/

/\* ename\_tab is a variable. \*/

* 1. **Populating and Referencing Associative Arrays**

Every reference to an element includes an associative array variable and a subscript enclosed in parentheses. The subscript determines which element is processed.

The syntax is as follows:

associative\_array\_variable(subscript)

The allowed subscript ranges are:

* For associative arrays with a numeric key, -231 ... 231.
* For associative arrays with a string key, the length of the key depends on the STRING length limit in the type declaration.

**Example 1A.** Based on table emp, build an associative array, the values are employee names in department 20, using binary integer as index (subscript).

DECLARE

TYPE t\_emp\_name IS TABLE OF emp.ename%TYPE

INDEX BY BINARY\_INTEGER;

v\_emp\_name t\_emp\_name;

CURSOR c\_emp\_name IS

SELECT ename

FROM emp

WHERE deptno = 20

ORDER BY 1;

v\_n INTEGER := 0;

BEGIN

FOR name\_rec IN c\_emp\_name LOOP

-- Cursor for loop takes care of advancing to next record;

v\_n := v\_n + 1;

-- programmer codes take care of the subscript values of the AA

v\_emp\_name(v\_n) := name\_rec.ename;

-- AA\_variable\_name (subscript) is assigned the value of cursor-column

DBMS\_OUTPUT.PUT\_LINE(TO\_CHAR(v\_n)|| '. ' || v\_emp\_name(v\_n));

END LOOP;

END;

RESULT:

1. ADAMS

2. FORD

3. JONES

4. SCOTT

5. SMITH

**Example 1B.**  Using the employee number as index, (similar/same as Example 1A)

DECLARE

TYPE t\_emp\_name IS TABLE OF emp.ename%TYPE

INDEX BY BINARY\_INTEGER;

v\_emp t\_emp\_name;

**i emp.empno%TYPE ;**

CURSOR c\_emp IS

SELECT empno, ename

FROM emp

WHERE deptno = 20

order by 2;

BEGIN

DBMS\_OUTPUT.PUT\_LINE( 'The printout is in order as insertion.') ;

FOR emp\_rec IN c\_emp LOOP

v\_emp (emp\_rec.**empno**) := emp\_rec.ename;

**-- “empno”** acts as subscript (index)

DBMS\_OUTPUT.PUT\_LINE(TO\_CHAR(emp\_rec.empno) || ' ' ||

v\_emp(emp\_rec.empno));

END LOOP;

DBMS\_OUTPUT.PUT\_LINE (' - - - - - ');

DBMS\_OUTPUT.PUT\_LINE( 'This printout is in order of the index.') ;

i := v\_emp.FIRST; -- Get first element of array

-- “variable\_name.FIRST”: we will discuss “Method” momentarily

WHILE i IS NOT NULL LOOP

DBMS\_Output.PUT\_LINE (TO\_CHAR(i)|| ' ' || v\_emp(i));

i := v\_emp.NEXT(i); -- Get next element of array

END LOOP;

END;

Result:

The printout is in order as insertion.

7876 ADAMS

7902 FORD

7566 JONES

7788 SCOTT

7369 SMITH

- - - - -

This printout is in order of the index.

7369 SMITH

7566 JONES

7788 SCOTT

7876 ADAMS

7902 FORD

Here, you can see that inside the associative array, the array is in sort order of the index - in this sample, it is the empno. You also can see this in the example below, where index is of string type.

**Example 2.** Using the employee name (char) as index - assume name is unique here

DECLARE

TYPE t\_job\_title IS TABLE OF emp.job%TYPE

INDEX BY VARCHAR2(9);

v\_job t\_job\_title;

i varchar2(9);

CURSOR c\_job\_title IS

SELECT ename, job

FROM emp

WHERE deptno = 20

order by 2;

BEGIN

DBMS\_OUTPUT.PUT\_LINE (' Print out in order of insertion ');

FOR job\_rec IN c\_job\_title LOOP

v\_job (job\_rec.ename) := job\_rec.job;

DBMS\_OUTPUT.PUT\_LINE (RPAD (job\_rec.ename, 10) ||

v\_job (job\_rec.ename));

END LOOP;

DBMS\_OUTPUT.PUT\_LINE (' ');

i := v\_job.FIRST; -- Get first element of array

DBMS\_OUTPUT.PUT\_LINE ('Print out in order of index ');

DBMS\_OUTPUT.PUT\_LINE ('Name Job Title ');

DBMS\_OUTPUT.PUT\_LINE ('------- ------------- ');

WHILE i IS NOT NULL LOOP

DBMS\_Output.PUT\_LINE

( RPAD(i, 10) || v\_job(i));

i := v\_job.NEXT(i); -- Get next element of array

END LOOP;

END;

/

RESULT:

Print out in order of insertion

SCOTT ANALYST

FORD ANALYST

ADAMS CLERK

SMITH CLERK

JONES MANAGER

Print out in order of index

Name Job Title

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

ADAMS CLERK

FORD ANALYST

JONES MANAGER

SCOTT ANALYST

SMITH CLERK

**An associative array is appropriate for:**

• A relatively small lookup table, which can be constructed in memory each time you

invoke the subprogram or initialize the package that declares it

• Passing collections to and from the database server

# Collection Constructors (Only for Varray and nested table, not for associative array)

A collection constructor is a system-defined function with the same name as a collection type, which returns a collection of that type.

The syntax of a constructor invocation is:

*collection\_type* ([ *value* [, *value*] ...]);

Refer to Example #3 on next page. If the parameter list is empty, the constructor returns an empty collection.

1. **Varray**

The Syntax for Declare the TYPE:

DECLARE

TYPE type\_name IS VARRY(max\_elements) OF element\_datatype [NOT NULL];

If it is declared as “NOT NULL”, then each element cannot be null, and the variable must be initialized with value for each element.

A **varray (variable-size array)** is an array whose number of elements can vary from

zero (empty) to the declared maximum size.

Unlike an associative array and nested table, a VARRAY always has a fixed number of elements (bounded) and never has gaps between the elements (not sparse).

Before using a Varry variable, you must initialize it. Otherwise, you will receive an error.

If not defined as not null, you may initialize it empty as below:

Variable\_name type\_name := type\_name( ); -- constructor

This makes the initial size of array as 0. This collection after constructor with () does exist, is said empty, as it has no element. Before calling constructor (), the collection is null, it does not exist. You need to use the function EXTEND to add one more element each time to populate the varray.

To access an element of a varray variable, use the syntax ***variable\_name*(*index*).**

The lower bound of *index* is 1; the upper bound is the current number of elements.

A Varray can be created, and is accessible globally in the database.

CREATE [OR REPLACE] TYPE type\_name AS | IS

VARRAY(max\_elements) OF element\_datatype [NOT NULL];

**Example 3.** declare in PL/SQL block

DECLARE

TYPE BridgeTeam IS VARRAY(4) OF VARCHAR2(15); -- VARRAY type

-- varray variable initialized/populated with constructor:

team1 BridgeTeam := BridgeTeam

('John', 'Mary', 'Alberto', 'Juanita');

team2 BridgeTeam := BridgeTeam( );

-- initialize to empty, will give values to each element later in execution section

BEGIN

DBMS\_OUTPUT.PUT\_LINE ('Team1 members:');

FOR i IN 1..4 LOOP

DBMS\_OUTPUT.PUT\_LINE(i || '.' || team1(i));

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('---');

team2 := team1;

team2(3) := 'Pierre'; -- Change values of two elements

team2(4) := 'Yvonne';

DBMS\_OUTPUT.PUT\_LINE('Team2:');

FOR i IN 1..4 LOOP

DBMS\_OUTPUT.PUT\_LINE(i || '.' || team2(i));

END LOOP;

END;

/

OUTPUT

Team1 members:

1.John

2.Mary

3.Alberto

4.Juanita

---

Team2:

1.John

2.Mary

3.Pierre

4.Yvonne

A varray is appropriate when:

• You know the maximum number of elements.

• You usually access the elements sequentially.

Because you must store or retrieve all elements at the same time, a varray might be impractical for large numbers of elements.

1. **Nested Table**

The Syntax:

DECLARE

TYPE type\_name IS **TABLE** OF Element\_datatype [NOT NULL];

(comparing with the declaration of associative array, this has no “index by”)

To create a nested table datatype that will be stored in the database (that will not just live in PL/SQL block, rather will be stored in database; and remember that you can “create” NT and Varray, but not AA):

CREATE [OR REPLACE] TYPE *type\_name* AS | IS

TABLE OF element\_datatype [NOT NULL];

**Example 4**. Nested table of local type (only lives in PL/SQL code, not in DB)

DECLARE

TYPE Roster IS TABLE OF VARCHAR2(15); -- nested table type

-- nested table variable initialized with constructor:

names Roster := Roster('D Caruso', 'J Hamil', 'D Piro',

'R Singh');

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Initial Values:');

FOR i IN names.FIRST .. names.LAST LOOP DBMS\_OUTPUT.PUT\_LINE(names(i));

END LOOP; -- The loop display the contents of the nested table

DBMS\_OUTPUT.PUT\_LINE ('---');

names(3) := 'P Perez'; -- Change value of one element, then print out again

DBMS\_OUTPUT.PUT\_LINE ('After changing the element #3:');

FOR i IN names.FIRST .. names.LAST LOOP

DBMS\_OUTPUT.PUT\_LINE (names(i));

END LOOP;

-- This time, rebuild the entire table, then print out

DBMS\_OUTPUT.PUT\_LINE ('---');

names := Roster('A Jansen', 'B Gupta');

DBMS\_OUTPUT.PUT\_LINE('After Changing the entire nested table:');

FOR i IN names.FIRST .. names.LAST LOOP DBMS\_OUTPUT.PUT\_LINE(names(i));

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('---');

END;

/

Result:

Initial Values:

D Caruso

J Hamil

D Piro

R Singh

---

After changing the element #3:

D Caruso

J Hamil

P Perez

R Singh

---

After Changing the entire nested table:

A Jansen

B Gupta

---

A nested table is appropriate when:

* The number of elements is not set.
* Index values are not consecutive.
* You must delete or update some elements, but not all elements simultaneously.

Important Differences Between Nested Tables and Arrays.

Conceptually, a nested table is like a one-dimensional array with an arbitrary number

of elements. However, a nested table differs from an array in these important ways:

* An array has a declared number of elements, but a nested table does not. The

size of a nested table can increase dynamically.

* An array is always dense. A nested array is dense initially, but it can become

sparse, because you can delete elements from it.

On page 5-15 manual, Figure 5-2 shows the important differences between a nested table and an array.

1. **Collection Methods (manual 5.9)**

A collection method is a built-in function or procedure that operates on collections (associative arrays, nested tables, and variable-size arrays).

The basic syntax of a collection invocation is:

*collection\_name*.*method* -- Not as standard function syntax

DELETE Procedure Deletes elements from collection.

FIRST/LAST Function Returns first/last index in collection.

PRIOR/NEXT Function Returns index that precedes/succeeds specified index.

COUNT Function Returns number of elements in collection (those with

values, not included deleted or trimmed).

EXISTS Function Returns TRUE if and only if specified element of a

collection exists.

LIMIT Function Returns maximum number of elements that collection

(Varray) can have. (for AA/nested table, it returns null).

TRIM Procedure Deletes elements from end of varray or nested table.

(trim does not change the size, the value of LIMIT)

EXTEND Procedure Adds elements to end of varray or nested table. (for

Varray, you cannot extend more than the maximum size)

For associative arrays, these methods include DELETE, FIRST/LAST, PRIO/NEXT, COUNT and EXISTS. (No TRIM, EXTEND, neither LIMIT)

**[DELETE] (procedure)**

DELETE deletes all elements from a collection of any type. (free the memory)

DELETE(n) deletes the element which index is n; if n does not exist, does nothing.

DELETE(m,n) deletes all elements whose index are in the range m .. n,

if both m and n exist and n > m; otherwise, does nothing.

Note: with actual argument n, or (m,n), it actually keeps the place holder for the “removed” element(s). For varray, only delete (all) is allowed.

**[EXISTS] function,**

EXISTS(*n*) returns TRUE if the *n*th **element** of the collection exists and FALSE otherwise. For a deleted element, EXISTS(*n*) returns FALSE

This method can be used to avoid raising SUBSCRIPT\_OUTSIDE\_LIMIT exceptions.

In a loop to go through a sparse collection, if not using .NEXT function, then we need to check if that element is deleted/null as in example 7 below:

IF Var\_ename.EXISTS(i) THEN -- This IF is a must when sparse

DBMS\_Output.PUT\_LINE (RPAD (i, 4) || Var\_ename (i));

END IF;

**[FIRST and LAST] function**

returns the index of the first/last element. If the collection is empty, it returns NULL.

(ignoring deleted elements)

**Example 5.** using same Associative Array as in Example 1A.

DECLARE

TYPE T\_ename IS TABLE OF emp.ename%TYPE

INDEX BY BINARY\_INTEGER;

Var\_ename T\_ename;

CURSOR c\_ename IS SELECT ename FROM emp;

indx\_n Integer := 0;

i integer;

last\_indx integer;

BEGIN

FOR name\_rec IN c\_ename LOOP

indx\_n := indx\_n + 1;

Var\_ename(indx\_n) := name\_rec.ename;

END LOOP; -- populated the AA variable,

DBMS\_OUTPUT.PUT\_LINE ('Total # after population is: '

||Var\_ename.count);

-- When the AA with integer as index, and dense,

last\_indx := indx\_n; -- we may use an integer as loop index,

i := 1;

DBMS\_OUTPUT.PUT\_LINE ('Print out after population. ');

DBMS\_OUTPUT.PUT\_LINE ('No. Name');

DBMS\_OUTPUT.PUT\_LINE ('-- --------');

WHILE i <= last\_indx LOOP

IF Var\_ename.EXISTS(i) THEN -- This IF is a must when sparse

DBMS\_Output.PUT\_LINE (RPAD (i, 4) || Var\_ename (i));

END IF;

i := i+1;

END LOOP;

/\*

i := Var\_ename.FIRST; -- Using “NEXT(i)”, it works when sparse

WHILE i IS NOT NULL LOOP

DBMS\_Output.PUT\_LINE (RPAD (i, 4) || Var\_ename (i));

i := Var\_ename.NEXT(i);

END LOOP;

\*/

DBMS\_OUTPUT.PUT\_LINE (' - - - - - ');

Var\_ename.DELETE (5); -- sample of using delete (n),

i := Var\_ename.FIRST; -- same loop code

DBMS\_OUTPUT.PUT\_LINE ('Print out after delete #5.');

DBMS\_OUTPUT.PUT\_LINE ('No. Name');

DBMS\_OUTPUT.PUT\_LINE ('-- --------');

WHILE i IS NOT NULL LOOP

DBMS\_Output.PUT\_LINE (RPAD (i, 4) || Var\_ename (i));

i := Var\_ename.NEXT(i);

END LOOP;

DBMS\_OUTPUT.PUT\_LINE (' - - - - - ');

Var\_ename.DELETE (8,10); -- sample of using delete (m, n),

i := Var\_ename.FIRST; -- same loop code

DBMS\_OUTPUT.PUT\_LINE ('Print out after delete #5, then 8-10.');

DBMS\_OUTPUT.PUT\_LINE ('No. Name');

DBMS\_OUTPUT.PUT\_LINE ('-- --------');

WHILE i IS NOT NULL LOOP

DBMS\_Output.PUT\_LINE (RPAD (i, 4) || Var\_ename (i));

i := Var\_ename.NEXT(i);

END LOOP;

DBMS\_OUTPUT.PUT\_LINE (' ');

DBMS\_OUTPUT.PUT\_LINE ('Next ind# after 7 is: '||Var\_ename.next(7));

DBMS\_OUTPUT.PUT\_LINE ('Total # of elements after deletions: '

||Var\_ename.count);

DBMS\_OUTPUT.PUT\_LINE ('1st ind# of elements is: '||Var\_ename.first);

DBMS\_OUTPUT.PUT\_LINE ('Last ind# of elements is: '||Var\_ename.last);

DBMS\_OUTPUT.PUT\_LINE ('Limit for this AA: '

||NVL (to\_char(Var\_ename.limit), 'Not available'));

END;

/

OUTPUT

Total # after population is: 14

Print out after population.

No. Name

-- --------

1 SMITH

2 ALLEN

...

Print out after delete #5, then 8-10.

No. Name

-- --------

1 SMITH

2 ALLEN

3 WARD

4 JONES

6 BLAKE

7 CLARK

11 ADAMS

12 JAMES

13 FORD

14 MILLER

Next ind# after 7 is: 11

Total # of elements after deletions: 10

1st ind# of elements is: 1

Last ind# of elements is: 14

Limit for this AA: Not available

Imaging what the AA like after the deletion of element 5th, then 8-10.

|  |  |  |
| --- | --- | --- |
| 1 | SMITH | <- First index |
| 2 | ALLEN |  |
| 3 | WARD |  |
| 4 | JONES |  |
| ~~5~~ | ~~MARTIN~~ | <- The space holder is still there |
| 6 | BLAKE |  |
| 7 | CLARK | <- The “next” index of element 7 is 11, not 8 |
| ~~8~~ | ~~SCOTT~~ |  |
| ~~9~~ | ~~KING~~ |  |
| ~~10~~ | ~~TURNER~~ |  |
| 11 | ADAMS | <- The “prior” index of element 11 is 7, not 10 |
| 12 | JAMES |  |
| 13 | FORD |  |
| 14 | MILLER | <- After deleted 5, 8 -10, the last index is still 14 |

total count is 10;

**[COUNT]**

Returns the number of elements in the collection.

(Not include the deleted elements)

Refer the statement with “ ||Var\_ename.count)” statements at the end of Example 5,

The AA was size of 14 after population, after deletions (5, 8, 9, 10), the print out is

Total # of elements after deletions: 10

**[LIMIT]**

Returns the **maximum** number of elements that the collection (Varray) can have.

Only a varray has a maximum size.

A Varray (such as defined as Varray (20)) may only have 14 elements with value; its size is as originally defined (here is 20).

Trim (method) may delete the values of elements, but it will not change the maximum size of that varray (return value of limit is same as maximum size).

Associative Array and nested table have no maximum number of elements, the LIMIT returns NULL. See example 5.

**[PRIOR and NEXT function]**

Collect\_variable\_name.prior (index)

returns the index of the preceding/succeeding **existing** elements of the collection (if one exists). Return NULL otherwise.

Next/Prior helps to run the loop when the collection is sparse - to skip the deleted place holder - otherwise, system will get a no data error when accessing a deleted element.

**[EXTEND]** Procedure Adds elements to end of varray or nested table. (Not for AA)

The EXTEND method (procedure) has these forms: (overloaded)

EXTEND appends one null element to the collection.

EXTEND(*n*) appends *n* null elements to the collection.

EXTEND(*n*,*i*) appends *n* copies of the *i*th element to the collection. (for collections

with NOT NULL elements)

Adding an element to Varray or nested table requires two separate steps:

make a memory slot (when initialization does not provide sufficient number of

element, then “extend” method is needed);

assign a value to it.

**Example 6**

DECLARE

TYPE va\_emp\_name IS VARRAY(15) OF emp.ename%TYPE ;

v\_emp\_name va\_emp\_name := va\_emp\_name ( ); -- constructor, empty

CURSOR c IS SELECT ename FROM emp WHERE deptno = 20;

counter integer := 0;

BEGIN

-- Method 1

v\_emp\_name.EXTEND (15); -- add 15 elements once

FOR indx IN c LOOP

counter := counter + 1;

v\_emp\_name(counter) := indx.ename;

DBMS\_OUTPUT.PUT\_LINE (counter|| ': ' || v\_emp\_name(counter));

END LOOP;

/\* Method 2

FOR indx IN c LOOP

counter := counter + 1;

v\_emp\_name.extend; -- add one element at the end each time inside the loop.

v\_emp\_name(counter) := indx.ename;

DBMS\_OUTPUT.PUT\_LINE (counter|| ': ' || v\_emp\_name(counter));

END LOOP;

\*/

END;

**[TRIM]** A procedure that deletes elements from the **end** of a varray or nested table.

This method has two forms:

* TRIM removes **one** element from the **end** of the collection, if the collection has at

least one element; otherwise, it raises the predefined exception

SUBSCRIPT\_BEYOND\_COUNT.

* TRIM **(*n*)** removes *n* elements from the end of the collection, if there are at least *n*

elements at the end; otherwise, it raises the predefined exception

SUBSCRIPT\_BEYOND\_COUNT.

TRIM operates on the internal size of a collection.

That is, if DELETE deletes an element but keeps a placeholder for it, then TRIM considers the element to exist. Therefore, TRIM can delete a deleted element.

PL/SQL does NOT keep placeholders for trimmed elements. Therefore, trimmed

elements are not included in the internal size of the collection, and you cannot restore

a trimmed element by assigning a valid value to it.

1. **Using BULK COLLECT** (Manual 12.4 Bulk SQL and Bulk Binding)

You can **bulk-fetch** from a cursor into an associative array.

DECLARE

TYPE emp\_name IS TABLE OF emp.ename%TYPE

INDEX BY BINARY\_INTEGER;

v\_emp\_name emp\_name;

CURSOR c\_ename IS SELECT ename FROM emp;

BEGIN

OPEN c\_ename;

FETCH c\_ename **BULK COLLECT INTO** v\_emp\_name;

CLOSE c\_ename;

END;

/

Cursor points to the select result in a special area, usually, retrieve one row at a time.

With bulk collect into, system will fetch all the rows in this area.

The system initializes associative arrays for you. Then, starting at index 1, it inserts elements consecutively**.**

**Example 7**

CREATE TABLE num\_table (id NUMBER(6));

BEGIN

FOR i in 1..100000 LOOP

INSERT INTO num\_table VALUES(i);

END LOOP;

END;

/

DECLARE

TYPE t\_1 IS TABLE OF num\_table.id%TYPE INDEX BY BINARY\_INTEGER;

v\_1 t\_1;

CURSOR c\_1 IS

SELECT id

FROM num\_table

ORDER BY id;

BEGIN

FOR idx IN c\_1 LOOP

v\_1(idx.id) := idx.id;

END LOOP;

/\* this bulk collect assign values for table variable v\_1; this commented codes

play the same role as the above three-line codes.

OPEN c\_1;

**FETCH c\_1 BULK COLLECT INTO v\_1;**

CLOSE c\_1;

\*/

FOR idx in v\_1.FIRST..v\_1.LAST LOOP

IF v\_1(idx) MOD 9000 = 0 THEN

DBMS\_OUTPUT.PUT\_LINE(v\_1(idx));

END IF;

END LOOP;

END;

1. **Collection Comparisons**

You cannot compare **associative array** **variables** to each other. (not legal if you have code like if (variable\_AA\_name1 = variable\_AA\_name2); but check (if is NULL) is fine)

You can compare varray and nested table variables to the value NULL with the "IS

[NOT] NULL Operator", but not with the relational operators equal (=) and not equal

(<>, !=, ~=, or ^=).

Two nested table variables are equal if and only if they have the same set of elements

(in any order).

If two nested table variables have the same nested table type, and that nested table type does not have elements of a *record* type, then you can compare the two variables for equality or inequality with the relational operators equal (=) and not equal (<>, !=, ~=, ^=).

**Example 8**

DECLARE

TYPE BridgeTeam IS VARRAY(4) OF VARCHAR2(15); -- VARRAY type

team BridgeTeam; -- varray variable

TYPE Roster IS TABLE OF VARCHAR2(15); -- nested table type

names Roster := Roster('Adams', 'Patel'); -- nested table variable

name2 Roster;

BEGIN

**IF team IS NULL THEN**

DBMS\_OUTPUT.PUT\_LINE('team IS NULL');

ELSE

DBMS\_OUTPUT.PUT\_LINE('team IS NOT NULL');

END IF;

**IF names IS NOT NULL THEN**

DBMS\_OUTPUT.PUT\_LINE('names IS NOT NULL');

ELSE

DBMS\_OUTPUT.PUT\_LINE('names IS NULL');

END IF;

name2 := names;

-- name2 (1) := 'Fred' ;

**IF (names = name2) THEN**

DBMS\_OUTPUT.PUT\_LINE('names equals to name2');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Not equal');

END IF;

END;

OUTPUT

team IS NULL

names IS NOT NULL

names equals to name2

Recall that

In example 3 on page 6, we have varray team2 := team1;

Same rule applies to AA.

**Appendix A,** Review

Invoke the Method: (using the invoking format in object) variable\_name.first

not use normal function call format: function\_name (parameter list)

|  |  |  |  |
| --- | --- | --- | --- |
|  | AA | Varray | Nested |
| Where defined | Not in schema (no create); PL/SQL, Packaged are ok. | All three:  Schema, PL/SQL, Packaged, | |
| Constructor () | N/A | := Type\_name () ( may need extend too) | |
| DELETE | Delete all elements, free memory | | |
| DELETE (n) | Delete nth element. (index = n). | NO | ok |
| DELETE (m, n) | Range from m to n, inclusive. | NO | ok |
| FIRST/LAST | Returns the index of the first/last element; if empty, returns NULL.  If sparse, “For (numeric) loop” could be not right. | | |
| PRIOR (index) NEXT (index) | Returns the index of the preceding/succeeding existing elements  Return NULL otherwise. (good for sparse collection) | | |
| COUNT | Returns the number of elements of the collection; up to current, the total number of elements (Varray, include null element); not included deleted or trimmed. | | |
| EXISTS (index) | Returns TRUE if the *n*th element exists; false otherwise | | |
| LIMIT | N/A,  return null | Maximum size | N/A,  return null |
| TRIM | N/A | Delete one element from end; It does not keep placeholder. | |
| TRIM (n) | N/A | Remove n elements from the end, | |
| EXTEND | N/A | appends one null element to the collection. | |
| EXTEND (n) | N/A | appends *n* null elements to the collection | |
| EXTEND (n, i) | N/A | appends *n* copies of the *i*th element to the collection. (for collections with NOT NULL elements) | |

**Appendix B**: Optional notes about BULK SQL

(refer to Manual chapter 12.4 Bulk SQL and Bulk Binding)

BULK COLLECT: SELECT statements that retrieve multiple rows with a single fetch, improving the speed of data retrieval

FORALL: INSERTs, UPDATEs, and DELETEs that use collections to change multiple rows of data very quickly

**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.

using **FORALL** (instead of for loop)

send batch of commands to



return the result

**Bulk collect (into)**

FOR indx in lower .. upper loop

statement ;

END loop

FORALL indx in lower .. upper

statement ;

(Forall for sparse collections, manual 12.4.1.1 skipped)

The BULK COLLECT clause can appear in:

SELECT INTO statement

FETCH statement

RETURNING INTO clause of:

DELETE statement

INSERT statement

UPDATE statement

EXECUTE IMMEDIATE statement

**Example 12-7** 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; -- Type Varray

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); -- i = 10, 30, 70 in turn

END LOOP; -- i is the bind variable,

END;

/

**Example 12-8** Same as 12-7**, except FORALL,** 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 -- Difference: FORALL

DELETE FROM employees\_temp

WHERE department\_id = depts(i); -- It is not a LOOP, so no “end loop”

END;

/

**Example** 12-26 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; -- Type NT

depts NumList := NumList(10,20,30);

TYPE enum\_t IS TABLE OF employees.employee\_id%TYPE; -- Type NT

e\_ids enum\_t;

TYPE dept\_t IS TABLE OF employees.department\_id%TYPE; -- Type NT

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; -- populate 2 NT’s

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;

**Example** 12-27 => almost same code as 12-26, except marked yellow part.

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;

/