**Working with Collections**

*By Steven Feuerstein Oracle ACE Director*

**Part 8 in a series of articles on understanding and using PL/SQL**

In the previous article in this series, I showed you how to work with a PL/SQL record, which is a *composite datatype* composed of one or more fields. In this article, I will explore another composite datatype, the collection. An Oracle PL/SQL collection is a single-dimensional array; it consists of one or more elements accessible through an index value.

Collections are used in some of the most important performance optimization features of PL/SQL, such as

* **BULK COLLECT.** SELECT statements that retrieve multiple rows with a single fetch, increasing the speed of data retrieval.
* **FORALL.** Inserts, updates, and deletes that use collections to change multiple rows of data very quickly
* **Table functions.** PL/SQL functions that return collections and can be called in the FROM clause of a SELECT statement.

You can also use collections to work with lists of data in your program that are not stored in database tables.

This article introduces you to collections and gives you a solid foundation in both collection syntax and features.

**Collection Concepts and Terminology**

Before exploring collections, it is helpful to have a common collections vocabulary that includes the following terms.

**Index value.** The location of the data in a collection. Index values are usually integers but for one type of collection can also be strings.

**Element.** The data stored at a specific index value in a collection. Elements in a collection are always of the same type (all of them are strings, dates, or records). PL/SQL collections are *homogeneous*.

**Sparse.** A collection is sparse if there is at least one index value between the lowest and highest defined index values that is not defined. For example, a sparse collection has an element assigned to index value 1 and another to index value 10 but nothing in between. The opposite of a sparse collection is a *dense* one.

**Method.** A collection method is a procedure or function that either provides information about the collection or changes the contents of the collection. Methods are attached to the collection variable with dot notation (object-oriented syntax), as in *my\_collection*.FIRST.

**Types of Collections**

Collections were first introduced in Oracle7 Server and have been enhanced in several ways through the years and across Oracle Database versions. There are now three types of collections to choose from, each with its own set of characteristics and each best suited to a different circumstance.

**Associative array.** The first type of collection available in PL/SQL, this was originally called a “PL/SQL table” and can be used only in PL/SQL blocks. Associative arrays can be sparse or dense and can be indexed by integer or string.

**Nested table.** Added in Oracle8 Database, the nested table can be used in PL/SQL blocks, in SQL statements, and as the datatype of columns in tables. Nested tables can be sparse but are almost always dense. They can be indexed only by integer. You can use the MULTISET operator to perform set operations and to perform equality comparisons on nested tables.

**Varray.**Added in Oracle8 Database, the varray (variable-size array) can be used in PL/SQL blocks, in SQL statements, and as the datatype of columns in tables. Varrays are always dense and indexed by integer. When a varray type is defined, you must specify the maximum number of elements allowed in a collection declared with that type.

You will rarely encounter a need for a varray (How many times do you know in advance the *maximum* number of elements you will define in your collection?). The associative array is the most commonly used collection type, but nested tables have some powerful, unique features (such as MULTISET operators) that can simplify the code you need to write to use your collection.

**Nested Table Example**

Let’s take a look at the simple example in Listing 1, which introduces the many aspects of collections explored later in the article.

**Code Listing 1:** Nested table example

1 DECLARE

2 TYPE list\_of\_names\_t IS TABLE OF VARCHAR2 (100);

3

4 happyfamily list\_of\_names\_t := list\_of\_names\_t ();

5 children list\_of\_names\_t := list\_of\_names\_t ();

6 parents list\_of\_names\_t := list\_of\_names\_t ();

7 BEGIN

8 happyfamily.EXTEND (4);

9 happyfamily (1) := ‘Veva’;

10 happyfamily (2) := ‘Chris’;

11 happyfamily (3) := ‘Eli’;

12 happyfamily (4) := ‘Steven’;

13

14 children.EXTEND;

15 children (children.LAST) := ‘Chris’;

16 children.EXTEND;

17 children (children.LAST) := ‘Eli’;

18

19 parents := happyfamily MULTISET EXCEPT children;

20

21 FOR l\_row IN 1 .. parents.COUNT

22 LOOP

23 DBMS\_OUTPUT.put\_line (parents (l\_row));

24 END LOOP;

25 END;

|  |  |
| --- | --- |
| **Lines** | **Explanation** |
| 2 | I declare a new nested table type. Each element in a collection declared with this type is a string whose maximum length is 100. |
| 4–6 | I declare three nested tables—happyfamily, children, and parents—based on my new collection type. Note that I also assign a default value to each variable by calling a constructor function that has the same name as the type. |
| 8 | I “make room” in my happyfamily collection for four elements by calling the EXTEND method. |
| 9–12 | I assign the names of the members of my immediate family (my wife, Veva; my two sons, Chris and Eli; and myself). Note the use of typical single-dimension array syntax to identify an element in the array: array\_name (index\_value). |
| 14–17 | Now I populate the children nested table with just the names of my sons. Rather than do a “bulk” extend as on line 8, I extend one index value at a time. Then I assign the name to the just-added index value by calling the LAST method, which returns the highest defined index value in the collection. Unless you know how many elements you need in advance, this approach of extending one row and then assigning a value to the new highest index value is the way to go. |
| 19 | Both of my children are adults and have moved out of the ancestral home. So who’s left in this place with too many bedrooms? Start with the happyfamily and subtract (with the MULTISET EXCEPT operator) the children. Assign the result of this set operation to the parents collection. It should be just Veva and Steven. |
| 21–24 | The result of a MULTISET operation is always either empty or densely filled and starts with index value 1. So I will iterate through all the elements in the collection, from 1 to the COUNT (the number of elements defined in the collection) and display the element found at each index value. |

When I run the block in Listing 1, I see the following output:

Veva

Steven

Listing 1 also includes references to the lines in the code block and descriptions of how those lines contribute to the nested table example.

**Declare Collection Types and Variables**

Before you can declare and use a collection variable, you need to define the *type* on which it is based. Oracle Database predefines several collection types in supplied packages such as DBMS\_SQL and DBMS\_UTILITY. So if you need, for example, to declare an associative array of strings whose maximum length is 32767, you could write the following:

l\_names DBMS\_UTILITY.maxname\_array;

In most cases, however, you will declare your own application-specific collection types. Here are examples of declaring each of the different types of collections:

1. Declare an associative array of numbers, indexed by integer:

TYPE numbers\_aat IS TABLE OF NUMBER

INDEX BY PLS\_INTEGER;

1. Declare an associative array of numbers, indexed by string:

TYPE numbers\_aat IS TABLE OF NUMBER

INDEX BY VARCHAR2(100);

1. Declare a nested table of numbers:

TYPE numbers\_nt IS TABLE OF NUMBER;

1. Declare a varray of numbers:

TYPE numbers\_vat IS VARRAY(10)

OF NUMBER;

Note: I use the suffixes \_aat, \_nt, and \_vat, for *associative array type, nested table type*, and *varray type*, respectively.

|  |
| --- |
| **Answers to the Challenge**  The PL/SQL Challenge question in last issue’s “Working with Records in PL/SQL” article tested your knowledge of how to declare a record variable based on a table or a cursor. The question asked which of the following could be used in the question’s code block so that a value (“Keyboard”) from the question’s table would be displayed. All the choices are listed below; only (c) and (d) are correct.  **a.**    l\_part plch\_parts%TYPE;    **b.**  l\_part plch\_parts;    **c.**  l\_part plch\_parts%ROWTYPE;    **d.**    CURSOR parts\_cur  IS  SELECT \* FROM plch\_parts;  l\_part parts\_cur%ROWTYPE; |

You might be wondering why the syntax for defining a collection type does not use the word *collection*. The answer is that the IS TABLE OF syntax was first introduced in Oracle7 Server, when there was just one type of collection, the PL/SQL table.

From these examples, you can draw the following conclusions about collection types:

* If the TYPE statement contains an INDEX BY clause, the collection type is an associative array.
* If the TYPE statement contains the VARRAY keyword, the collection type is a varray.
* If the TYPE statement does not contain an INDEX BY clause or a VARRAY keyword, the collection type is a nested table.
* Only the associative array offers a choice of indexing datatypes. Nested tables as well as varrays are always indexed by integer.
* When you define a varray type, you specify the maximum number of elements that can be defined in a collection of that type.

Once you’ve declared a collection type, you can use it to declare a collection variable as you would declare any other kind of variable:

DECLARE

TYPE numbers\_nt IS TABLE OF NUMBER;

l\_numbers numbers\_nt;

**Initializing Collections**

When you work with nested tables and varrays, you must *initialize* the collection variable before you can use it. You do this by calling the constructor function for that type. This function is created automatically by Oracle Database when you declare the type. The constructor function constructs an instance of the type associated with the function. You can call this function with no arguments, or you can pass it one or more expressions of the same type as the elements of the collection, and they will be inserted into your collection.

Here is an example of initializing a nested table of numbers with three elements (1, 2, and 3):

DECLARE

TYPE numbers\_nt IS TABLE OF NUMBER;

l\_numbers numbers\_nt;

BEGIN

l\_numbers := numbers\_nt (1, 2, 3);

END;

If you neglect to initialize your collection, Oracle Database will raise an error when you try to use that collection:

SQL> DECLARE

2 TYPE numbers\_nt IS TABLE OF NUMBER;

3 l\_numbers numbers\_nt;

4 BEGIN

5 l\_numbers.EXTEND;

6 l\_numbers(1) := 1;

7 END;

8 /

DECLARE

\*

ERROR at line 1:

ORA-06531: Reference to uninitialized collection

ORA-06512: at line 5

You do *not* need to initialize an associative array before assigning values to it.

**Populating Collections**

You can assign values to elements in a collection in a variety of ways:

* Call a constructor function (for nested tables and varrays).
* Use the assignment operator, for single elements as well as entire collections.
* Pass the collection to a subprogram as an OUT or IN OUT parameter, and then assign the value inside the subprogram.
* Use a BULK COLLECT query.

The previous section included an example that used a constructor function. Following are examples of the other approaches:

1. Assign a number to a single index value. Note that with an associative array, it is not necessary to use EXTEND or start with index value 1.

DECLARE

TYPE numbers\_aat IS TABLE OF NUMBER

INDEX BY PLS\_INTEGER;

l\_numbers numbers\_aat;

BEGIN

l\_numbers (100) := 12345;

END;

1. Assign one collection to another. As long as both collections are declared with the same type, you can perform collection-level assignments.

DECLARE

TYPE numbers\_aat IS TABLE OF NUMBER

INDEX BY PLS\_INTEGER;

l\_numbers1 numbers\_aat;

l\_numbers2 numbers\_aat;

BEGIN

l\_numbers1 (100) := 12345;

l\_numbers2 := l\_numbers1;

END;

1. Pass a collection as an IN OUT argument, and remove all the elements from that collection:

DECLARE

TYPE numbers\_aat IS TABLE OF NUMBER

INDEX BY PLS\_INTEGER;

l\_numbers numbers\_aat;

PROCEDURE empty\_collection (

numbers\_io IN OUT numbers\_aat)

IS

BEGIN

numbers\_io.delete;

END;

BEGIN

l\_numbers (100) := 123;

empty\_collection (l\_numbers);

END;

1. Fill a collection directly from a query with BULK COLLECT (covered in more detail in the next article in this series):

DECLARE

TYPE numbers\_aat IS TABLE OF NUMBER

INDEX BY PLS\_INTEGER;

l\_numbers numbers\_aat;

BEGIN

SELECT employee\_id

BULK COLLECT INTO l\_numbers

FROM employees

ORDER BY last\_name;

END;

**Iterating Through Collections**

A very common collection operation is to iterate through all of a collection’s elements. Reasons to perform a “full collection scan” include displaying information in the collection, executing a data manipulation language (DML) statement with data in the element, and searching for specific data.

The kind of code you write to iterate through a collection is determined by the type of collection with which you are working and how it was populated. Generally, you will choose between a numeric FOR loop and a WHILE loop.

Use a numeric FOR loop when

* Your collection is densely filled (every index value between the lowest and the highest is defined)
* You want to scan the entire collection, not terminating your scan if some condition is met

Conversely, use a WHILE loop when

* Your collection may be sparse
* You might terminate the loop before you have iterated through all the elements in the collection

You should use a numeric FOR loop with dense collections to avoid a NO\_DATA\_FOUND exception. Oracle Database will also raise this exception, however, if you try to “read” an element in a collection at an undefined index value.

The following block, for example, raises a NO\_DATA\_FOUND exception:

DECLARE

TYPE numbers\_aat IS TABLE OF NUMBER

INDEX BY PLS\_INTEGER;

l\_numbers numbers\_aat;

BEGIN

DBMS\_OUTPUT.PUT\_LINE (l\_numbers (100));

END;

When, however, you know for certain that your collection is—and will always be—densely filled, the FOR loop offers the simplest code for getting the job done. The procedure in Listing 2, for example, displays all the strings found in a collection whose type is defined in the DBMS\_UTILITY package.

**Code Listing 2:** Display all strings in a collection

CREATE OR REPLACE PROCEDURE show\_contents (

names\_in IN DBMS\_UTILITY.maxname\_array)

IS

BEGIN

FOR indx IN names\_in.FIRST .. names\_in.LAST

LOOP

DBMS\_OUTPUT.put\_line (names\_in (indx));

END LOOP;

END;

/

This procedure calls two *methods*: FIRST and LAST. FIRST returns the lowest defined index value in the collection, and LAST returns the highest defined index value in the collection.

The following block will display three artists’ names; note that the index values do not need to start at 1.

DECLARE

l\_names DBMS\_UTILITY.maxname\_array;

BEGIN

l\_names (100) := ‘Picasso’;

l\_names (101) := ‘O’’Keefe’;

l\_names (102) := ‘Dali’;=

show\_contents (l\_names);

END;

/

If your collection may be sparse or you want to terminate the loop conditionally, a WHILE loop will be the best fit. The procedure in Listing 3 shows this approach.

**Code Listing 3:** Use WHILE to iterate through a collection

CREATE OR REPLACE PROCEDURE show\_contents (

names\_in IN DBMS\_UTILITY.maxname\_array)

IS

l\_index PLS\_INTEGER := names\_in.FIRST;

BEGIN

WHILE (l\_index IS NOT NULL)

LOOP

DBMS\_OUTPUT.put\_line (names\_in (l\_index));

l\_index := names\_in.NEXT (l\_index);

END LOOP;

END;

/

In this procedure, my iterator (l\_index) is initially set to the lowest defined index value. If the collection is empty, both FIRST and LAST will return NULL. The WHILE loop terminates when l\_index is NULL. I then display the name at the current index value and call the NEXT method to get the next defined index value higher than l\_index. This function returns NULL when there is no higher index value.

I call this procedure in the following block, with a collection that is not sequentially filled. It will display the three names without raising NO\_DATA\_FOUND:

DECLARE

l\_names DBMS\_UTILITY.maxname\_array;

BEGIN

l\_names (-150) := 'Picasso';

l\_names (0) := 'O''Keefe';

l\_names (307) := 'Dali';

show\_contents (l\_names);

END;

/

I can also scan the contents of a collection in reverse, starting with LAST and using the PRIOR method, as shown in Listing 4.

**Code Listing 4:** Scan a collection in reverse

CREATE OR REPLACE PROCEDURE show\_contents (

names\_in IN DBMS\_UTILITY.maxname\_array)

IS

l\_index PLS\_INTEGER := names\_in.LAST;

BEGIN

WHILE (l\_index IS NOT NULL)

LOOP

DBMS\_OUTPUT.put\_line (names\_in (l\_index));

l\_index := names\_in.PRIOR (l\_index);

END LOOP;

END;

/

|  |
| --- |
| **Next Steps**  **DOWNLOAD**[**Oracle Database 11*g***](http://www.oracle.com/technetwork/database/enterprise-edition/downloads/index.html)  **TEST**[**your PL/SQL knowledge**](http://www.plsqlchallenge.com/)  **READ**[**PL/SQL 101, Parts 1-7**](http://www.oracle.com/technetwork/issue-archive/index-087690.html)  **READ more Feuerstein** [**stevenfeuerstein.com**](http://www.stevenfeuerstein.com/) [**toadworld.com/sf**](http://www.toadworld.com/sf) [***Oracle PL/SQL Programming***](http://amzn.to/IDhqef) [***Oracle PL/SQL Language Pocket Reference***](http://amzn.to/KuEMl6) [***Oracle PL/SQL Best Practices***](http://amzn.to/JTXAcg) |

**Deleting Collection Elements**

PL/SQL offers a DELETE method, which you can use to remove all, one, or some elements from a collection. Here are some examples:

1. Remove all elements from a collection; use the DELETE method without any arguments. This form of DELETE works with all three kinds of collections.

l\_names.DELETE;

1. Remove the first element in a collection; to remove one element, pass the index value to DELETE. This form of DELETE can be used only with an associative array or a nested table.

l\_names.DELETE (l\_names.FIRST);

1. Remove all the elements between the specified low and high index values. This form of DELETE can be used only with an associative array or a nested table.

l\_names.DELETE (100, 200);

If you specify an undefined index value, Oracle Database will *not* raise an error.

You can also use the TRIM method with varrays and nested tables to remove elements from the end of the collection. You can trim one or many elements:

l\_names.TRIM;

l\_names.TRIM (3);

**Get Comfy with Collections**

It is impossible to take full advantage of PL/SQL, including some of its powerful features, if you do not use collections. This article has provided a solid foundation for working with collections, but there are still several advanced features to explore, including string-indexed and nested collections, which will be covered in a future article.

The next article in this PL/SQL 101 series will explore how to *use* collections with PL/SQL’s most important performance-related PL/SQL features: FORALL and BULK COLLECT.

|  |
| --- |
| **Take the Challenge!**  Each PL/SQL 101 article offers a quiz to test your knowledge of the information provided in it. The quiz appears below and also at PL/SQL Challenge ([plsqlchallenge.com](http://www.plsqlchallenge.com/)), a Website that offers online quizzes on the PL/SQL language as well as SQL and Oracle Application Express.    **Question**  Which of the following blocks will display these three lines after execution:    Strawberry  Raspberry  Blackberry    **a.**  DECLARE  l\_names DBMS\_UTILITY.maxname\_array;  BEGIN  l\_names (1) := 'Strawberry';  l\_names (10) := 'Blackberry';  l\_names (2) := 'Raspberry';  FOR indx IN 1 .. l\_names.COUNT  LOOP  DBMS\_OUTPUT.put\_line (l\_names (indx));  END LOOP;  END;  /    **b.**    DECLARE  l\_names DBMS\_UTILITY.maxname\_array;  BEGIN  l\_names (1) := 'Strawberry';  l\_names (10) := 'Blackberry';  l\_names (2) := 'Raspberry';  indx := l\_names.FIRST;  WHILE (indx IS NOT NULL)  LOOP  DBMS\_OUTPUT.put\_line (l\_names (indx));  indx := l\_names.NEXT (indx);  END LOOP;  END;  /    **c.**    DECLARE  l\_names DBMS\_UTILITY.maxname\_array;  BEGIN  l\_names (1) := 'Strawberry';  l\_names (10) := 'Blackberry';  l\_names (2) := 'Raspberry';  DECLARE  indx PLS\_INTEGER := l\_names.FIRST;  BEGIN  WHILE (indx IS NOT NULL)  LOOP  DBMS\_OUTPUT.put\_line (l\_names (indx));  indx := l\_names.NEXT (indx);  END LOOP;  END;  END;  /    **d.**    DECLARE  l\_names DBMS\_UTILITY.maxname\_array;  BEGIN  l\_names (1) := 'Strawberry';  l\_names (10) := 'Blackberry';  l\_names (2) := 'Raspberry';  FOR indx IN l\_names.FIRST .. l\_names.LAST  LOOP  DBMS\_OUTPUT.put\_line (l\_names (indx));  END LOOP;  END;  / |