CHAPTER 12
Collections

A *collection* is a data structure that acts like a list or a single-dimensional array. Collections are, in fact, the closest you can get in the PL/SQL language to traditional arrays. This chapter will help you decide which of the three different types of collection (associative array, nested table, and VARRAY) best fits your program's requirements and show you how to define and manipulate those structures.

Here are some of the ways I've found collections handy:

To maintain in-program lists of data
: Most generally, I use collections to keep track of lists of data elements within my programs. Yes, you could use relational tables or global temporary tables (which would involve many context switches) or delimited strings, but collections are very efficient structures that can be manipulated with very clean, maintainable code.

To improve multirow SQL operations by an order of magnitude or more
: You can use collections in conjunction with FORALL and BULK COLLECT to dramatically improve the performance of multirow SQL operations. These "bulk" operations are covered in detail in Chapter 21.

To cache database information
: Collections are appropriate for caching database information that is static and frequently queried in a single session (or simply queried repeatedly in a single program) to speed up the performance of those queries.

I have noticed over the years that relatively few developers know about and use collections. This always comes as a surprise, because I find them to be so handy. A primary reason for this limited usage is that collections are relatively complicated. Three different types of collections, multiple steps involved in defining and using them, usage in both PL/SQL programs and database objects, more complex syntax than simply working with individual variables—all of these factors conspire to limit usage of collections.

I have organized this chapter to be comprehensive in my treatment of collections, avoid redundancy in treatment of similar topics across different collection types, and offer guidance in your usage of collections. The resulting chapter is rather long, but I'm confident you will get lots out of it. Here is a quick guide to the remainder of its contents:

Collections overview
: I start by providing an introduction to collections and some orientation: a description of the different types, an explanation of the terminology specific to collections, a robust example of each type of collection, and guidelines for deciding which type of collection to use. If you read no further than this section, you will likely be able to start writing some basic collection logic. I strongly suggest, however, that you *do* read more than this section!

Collection methods
: Next, I explore the many methods (procedures and functions) that Oracle provides to help you examine and manipulate the contents of a collection. Virtually every usage of collections requires usage of these methods, so you want to make sure you are comfortable with what they do and how they work.

Working with collections
: Now it is time to build on all those "preliminaries" to explore some of the nuances of working with collections, including the initialization process necessary for nested tables and VARRAYs, different ways to populate and access collection data, the manipulation of collection columns through the SQL language, and string-indexed collections.

Nested table multiset operations
: Oracle Database 10*g* "filled out" the implementation of nested tables as multisets by providing the ability to manipulate the contents of nested tables as sets (union, intersection, minus, etc.). You can also compare two nested tables for equality and inequality.

Maintaining schema-level collections
: You can define nested table and VARRAY types within the database itself. The database provides a number of data dictionary views you can use to maintain those types.

Collections Overview

Let's start with a review of collection concepts and terminology, a description of the different types of collections, and a number of examples to get you going.

Collections Concepts and Terminology

The following explanations will help you understand collections and more rapidly establish a comfort level with these data structures:

Element and index value
: A collection consists of multiple elements (chunks of data), each element of which is located at a certain index value in the list. You will sometimes see an element also referred to as a *row*, and an index value referred to as the *row number*.

Collection type
: Each collection variable in your program must be declared based on a predefined collection *type*. As I mentioned earlier, there are, very generally, three types of collections: associative arrays, nested tables, and VARRAYs. Within those generic types, there are specific types that you define with a TYPE statement in a block's declaration section. You can then declare and use instances of those types in your programs.

Collection or collection instance
: The term *collection* may refer to any of the following:

  - A PL/SQL variable of type associative array, nested table, or VARRAY
  - A table column of type nested table or VARRAY

  Regardless of the particular type or usage, however, a collection is at its core a single-dimensional list of homogeneous elements.

  A collection instance is an instance of a particular type of collection.

  Partly due to the syntax and names Oracle has chosen to support collections, you will also find them referred to as *arrays* and *tables*.

Homogeneous elements
: The datatype of each row element in a collection is the same; thus, its elements are *homogeneous*. This datatype is defined by the type of collection used to declare the collection itself, but it can be a composite or complex datatype; you can declare a table of records, for example. And starting with Oracle9*i* Database, you can even define multilevel collections, in which the datatype of one collection is itself a collection type, or a record or object whose attribute contains a collection.

One-dimensional or single-dimensional
: A PL/SQL collection always has just a single column of information in each row, and is in this way similar to a one-dimensional array. You cannot define a collection so that it can be referenced as follows:

```
my_collection (10, 44)
```

This is a two-dimensional structure and is not currently supported with that traditional syntax. Instead, you can create multidimensional arrays by declaring collections of collections, in which case the syntax you use will be something like this:

```
my_collection (44) (10)
```

Unbounded versus bounded
: A collection is said to be *bounded* if there are predetermined limits to the possible values for row numbers in that collection. It is *unbounded* if there are no upper or lower limits on those row numbers. VARRAYs or variable-sized arrays are always bounded; when you define such an array, you specify the maximum number of rows allowed in that collection (the first row number is always 1). Nested tables and associative arrays are only *theoretically* bounded. I describe them as unbounded, because from a theoretical standpoint, there is no limit to the number of rows you can define in them. Practically speaking, however, your session will fail with an out-of-memory error if you attempt to come even close to the theoretical limits.

Sparse versus dense
: A collection (or array or list) is called *dense* if all rows between the first and last row are defined and given a value (including NULL). A collection is *sparse* if rows are not defined and populated sequentially; instead, there are gaps between defined rows, as demonstrated in the associative array example in the next section. VARRAYs are always dense. Nested tables always start as dense collections but can be made sparse. Associative arrays can be sparse or dense, depending on how you fill the collection.

  Sparseness is a very valuable feature, as it gives you the flexibility to populate rows in a collection using a primary key or other intelligent key data as the row number. By doing so, you can define an order on the data in a collection or greatly enhance the performance of lookups.

Indexed by integers
: All collections support the ability to reference a row via the row number, an integer value. The associative array TYPE declaration makes that explicit with its INDEX BY clause, but the same rule holds true for the other collection types.

Indexed-by strings
: Starting with Oracle9*i* Database Release 2, it is possible to index an associative array by string values (currently up to 32 KB in length) instead of by numeric row numbers. This feature is not available for nested tables or VARRAYs.

Outer table
: This refers to the *enclosing* table in which you have used a nested table or VARRAY as a column's datatype.

Inner table
: This is the *enclosed* collection that is implemented as a column in a table; it is also known as a *nested table column*.

Store table
: This is the physical table that Oracle creates to hold values of the inner table (a nested table column).

Types of Collections

As mentioned earlier, Oracle supports three different types of collections. While these different types have much in common, they also each have their own particular characteristics, which are summarized as follows:

Associative arrays
: These are single-dimensional, unbounded, sparse collections of homogeneous elements that are available only in PL/SQL. They were called *PL/SQL tables* in PL/SQL 2 (which shipped with Oracle 7) and *index-by tables* in Oracle8 Database and Oracle8*i* Database (because when you declare such collections, you explicitly state that they are "indexed by" the row number). In Oracle9*i* Database Release 1, the name was changed to *associative arrays*. The motivation for the name change was that starting with that release, the INDEX BY syntax could be used to "associate" or index contents by VARCHAR2 or PLS_INTEGER.

Nested tables
: These are also single-dimensional, unbounded collections of homogeneous elements. They are initially dense but can become sparse through deletions. Nested tables can be defined in both PL/SQL and the database (for example, as a column in a table). Nested tables are *multisets*, which means that there is no inherent order to the elements in a nested table.

VARRAYs
: Like the other two collection types, VARRAYs (variable-sized arrays) are single-dimensional collections of homogeneous elements. However, they are always bounded and never sparse. When you define a type of VARRAY, you must also specify the maximum number of elements it can contain. Like nested tables, they can be used in PL/SQL and in the database. Unlike nested tables, when you store and retrieve a VARRAY, its element order is preserved.

Collection Examples

This section provides relatively simple examples of each different type of collection with explanations of the major characteristics.

Using an associative array

In the following example, I declare an associative array type and then a collection based on that type. I populate it with four rows of data and then iterate through the collection, displaying the strings in the collection. A more thorough explanation appears in the table after the code:

```
 1  DECLARE
 2     TYPE list_of_names_t IS TABLE OF person.first_name%TYPE
 3        INDEX BY PLS_INTEGER;
 4     happyfamily   list_of_names_t;
 5     l_row PLS_INTEGER;
 6  BEGIN
 7     happyfamily (2020202020) := 'Eli';
 8     happyfamily (-15070) := 'Steven';
 9     happyfamily (-90900) := 'Chris';
10     happyfamily (88) := 'Veva';
11
12     l_row := happyfamily.FIRST;
13
14     WHILE (l_row IS NOT NULL)
15     LOOP
16        DBMS_OUTPUT.put_line (happyfamily (l_row));
17        l_row := happyfamily.NEXT (l_row);
18     END LOOP;
19  END;
```

The output is:

```
Chris
Steven
Veva
Eli
```

Line(s)	Description
2–3	Declare the associative array TYPE, with its distinctive INDEX BY clause. A collection based on this type contains a list of strings, each of which can be as long as the first_name column in the person table.
4	Declare the happyfamily collection from the list_of_names_t type.
9–10	Populate the collection with four names. Notice that I can use virtually any integer value that I like. The row numbers don't have to be sequential in an associative array; they can even be negative! I hope, however, that *you* will never write code with such bizarre, randomly selected index values. I simply wanted to demonstrate the flexibility of an associative array.
12	Call the FIRST method (a function that is "attached" to the collection) to get the first or lowest defined row number in the collection.
14–18	Use a WHILE loop to iterate through the contents of the collection, displaying each row. Line 17 shows the NEXT method, which is used to move from the current defined row to the next defined row, "skipping over" any gaps.

Using a nested table

In the following example, I first declare a nested table type as a schema-level type. In my PL/SQL block, I declare three nested tables based on that type. I put the names of everyone in my family into the happyfamily nested table. I put the names of my children in the children nested table. I then use the set operator, MULTISET EXCEPT (introduced in Oracle Database 10*g*), to extract just the parents from the happyfamily nested table; finally, I display the names of the parents. A more thorough explanation appears in the table after the code:

```
REM Section A
SQL> CREATE TYPE list_of_names_t IS TABLE OF VARCHAR2 (100);
  2  /
Type created.

REM Section B
  1  DECLARE
  2     happyfamily   list_of_names_t := list_of_names_t ();
  3     children      list_of_names_t := list_of_names_t ();
  4     parents       list_of_names_t := list_of_names_t ();
  5  BEGIN
  6     happyfamily.EXTEND (4);
  7     happyfamily (1) := 'Eli';
  8     happyfamily (2) := 'Steven';
  9     happyfamily (3) := 'Chris';
 10     happyfamily (4) := 'Veva';
 11
 12     children.EXTEND;
 13     children (1) := 'Chris';
 14     children.EXTEND;
 15     children (2) := 'Eli';
 16
 17     parents := happyfamily MULTISET EXCEPT children;
 18
 19     FOR l_row IN parents.FIRST .. parents.LAST
 20     LOOP
 21        DBMS_OUTPUT.put_line (parents (l_row));
 22     END LOOP;
 23  END;
```

The output is:

```
Steven
Veva
```

Line(s)	Description
Section A	The CREATE TYPE statement creates a nested table type in the database itself. By taking this approach, I can use the type to declare nested tables from within any schema that has EXECUTE authority on that type. I can also declare columns in relational tables of this type.

Line(s)	Description
2–4	Declare three different nested tables based on the schema-level type. Notice that in each case I also call a *constructor* function to initialize the nested table. This function always has the same name as the type and is created for us by Oracle. You must initialize a nested table before it can be used.
6	Call the EXTEND method to "make room" in my nested table for the members of my family. Here, in contrast to associative arrays, I must explicitly ask for a row in a nested table before I can place a value in that row.
7–10	Populate the happyfamily collection with our names.
12–15	Populate the children collection. In this case, I extend a single row at a time.
17	To obtain the parents in this family, I simply take the children out of the happyfamily collection. This is straightforward in releases from Oracle Database 10*g* onward, where we have high-level set operators like MULTISET EXCEPT (very similar to the SQL MINUS). Notice that I do not need to call the EXTEND method before filling parents. The database will do this for me automatically, when populating a collection with set operators and SQL operations.
19–22	Because I know that my parents collection is *densely filled* from the MULTISET EXCEPT operation, I can use the numeric FOR loop to iterate through the contents of the collection. This construct will raise a NO_DATA_FOUND exception if used with a sparse collection.

Using a VARRAY

In the following example, I demonstrate the use of VARRAYs as columns in a relational table. First, I declare two different schema-level VARRAY types. I then create a relational table, family, that has two VARRAY columns. Finally, in my PL/SQL code, I populate two local collections and then use them in an INSERT into the family table. A more thorough explanation appears in the table after the code:

```
REM Section A
SQL> CREATE TYPE first_names_t IS VARRAY (2) OF VARCHAR2 (100);
  2  /
Type created.

SQL> CREATE TYPE child_names_t IS VARRAY (1) OF VARCHAR2 (100);
  2  /
Type created.

REM Section B
SQL> CREATE TABLE family (
  2     surname VARCHAR2(1000)
  3   , parent_names first_names_t
  4   , children_names child_names_t
  5   );

Table created.

REM Section C
SQL>
  1  DECLARE
  2     parents    first_names_t := first_names_t ();
  3     children   child_names_t := child_names_t ();
  4  BEGIN
```

```
 5     parents.EXTEND (2);
 6     parents (1) := 'Samuel';
 7     parents (2) := 'Charina';
 8     --
 9     children.EXTEND;
10     children (1) := 'Feather';
11
12     --
13     INSERT INTO family
14                 ( surname, parent_names, children_names )
15          VALUES ( 'Assurty', parents, children );
16  END;
SQL> /

PL/SQL procedure successfully completed.

SQL> SELECT * FROM family
  2  /

SURNAME
PARENT_NAMES
CHILDREN_NAMES
----------------------------------------
Assurty
FIRST_NAMES_T('Samuel', 'Charina')
CHILD_NAMES_T('Feather')
```

Line(s)	Description
Section A	Use CREATE TYPE statements to declare two different VARRAY types. Notice that with a VARRAY, I must specify the maximum length of the collection. Thus, my declarations in essence dictate a form of social policy: you can have at most two parents and at most one child.
Section B	Create a relational table, with three columns: a VARCHAR2 column for the surname of the family and two VARRAY columns, one for the parents and another for the children.
Section C, lines 2–3	Declare two local VARRAYs based on the schema-level type. As with nested tables (and unlike with associative arrays), I must call the constructor function of the same name as the TYPE to initialize the structures.
5–10	Extend and populate the collections with the names of the parents and then the single child. If I try to extend to a second row, the database will raise the *ORA-06532: Subscript outside of limit* error.
13–15	Insert a row into the family table, simply providing the VARRAYs in the list of values for the table. Oracle certainly makes it easy for us to insert collections into a relational table!

Where You Can Use Collections

The following sections describe the different places in your code where a collection can be declared and used. Because a collection type can be defined in the database itself (nested tables and VARRAYs only), you can find collections not only in PL/SQL programs but also inside tables and object types.

Collections as components of a record

Using a collection type in a record is similar to using any other type. You can use associative arrays, nested tables, VARRAYs, or any combination thereof in RECORD datatypes. For example:

```
CREATE OR REPLACE TYPE color_tab_t IS TABLE OF VARCHAR2(100)
/

DECLARE
   TYPE toy_rec_t IS RECORD (
      manufacturer INTEGER,
      shipping_weight_kg NUMBER,
      domestic_colors color_tab_t,
      international_colors color_tab_t
   );
```

Collections as program parameters

Collections can also serve as parameters in functions and procedures. The format for the parameter declaration is the same as with any other (see Chapter 17 for more details):

```
parameter_name [ IN | IN OUT | OUT ] parameter_type
   [ [ NOT NULL ] [ DEFAULT | := default_value ] ]
```

PL/SQL does not offer generic, predefined collection types (except in certain supplied packages, such as DBMS_SQL and DBMS_UTILITY). This means that before you can pass a collection as an argument, you must have already defined the collection type that will serve as the parameter type. You can do this by:

- Defining a schema-level type with CREATE TYPE
- Declaring the collection type in a package specification
- Declaring that type in an outer scope from the definition of the module

Here is an example of using a schema-level type:

```
CREATE TYPE yes_no_t IS TABLE OF CHAR(1);
/
CREATE OR REPLACE PROCEDURE act_on_flags (flags_in IN yes_no_t)
IS
BEGIN
    ...
END act_on_flags;
/
```

Here is an example of using a collection type defined in a package specification: there is only one way to declare an associative array of Booleans (and all other base datatypes), so why not define them once in a package specification and reference them throughout my application?

```
/* File on web: aa_types.pks */
CREATE OR REPLACE PACKAGE aa_types
IS
   TYPE boolean_aat IS TABLE OF BOOLEAN INDEX BY PLS_INTEGER;
   ...
END aa_types;
/
```

Notice that when I reference the collection type in my parameter list, I must qualify it with the package name:

```
CREATE OR REPLACE PROCEDURE act_on_flags (
   flags_in IN aa_types.boolean_aat)
IS
BEGIN
    ...
END act_on_flags;
/
```

Finally, here is an example of declaring a collection type in an outer block and then using it in an inner block:

```
DECLARE
   TYPE birthdates_aat IS VARRAY (10) OF DATE;
   l_dates   birthdates_aat := birthdates_aat ();
BEGIN
   l_dates.EXTEND (1);
   l_dates (1) := SYSDATE;

   DECLARE
      FUNCTION earliest_birthdate (list_in IN birthdates_aat) RETURN DATE
      IS
      BEGIN
        ...
      END earliest_birthdate;
   BEGIN
      DBMS_OUTPUT.put_line (earliest_birthdate (l_dates));
   END;
END;
```

Collection as datatype of a function's return value

In the next example, I have defined color_tab_t as the type of a function return value, and also used it as the datatype of a local variable. The same restriction about scope applies to this usage—types must be declared outside the module's scope:

```
FUNCTION true_colors (whose_id IN NUMBER) RETURN color_tab_t
AS
   l_colors color_tab_t;
BEGIN
   SELECT favorite_colors BULK COLLECT INTO l_colors
     FROM personality_inventory
    WHERE person_id = whose_id;
```

```
   RETURN l_colors;
END;
```

(You'll meet BULK COLLECT properly in Chapter 15.)

How would you use this function in a PL/SQL program? Because it acts in the place of a variable of type color_tab_t, you can do one of two things with the returned data:

1. Assign the entire result to a collection variable.
2. Assign a single element of the result to a variable (as long as the variable is of a type compatible with the collection's elements).

Option #1 is easy. Notice, by the way, that this is another circumstance where you don't have to initialize the collection variable explicitly:

```
DECLARE
   color_array color_tab_t;
BEGIN
   color_array := true_colors (8041);
END;
```

With Option #2, I put a subscript after the function call, as follows:

```
DECLARE
   one_of_my_favorite_colors VARCHAR2(30);
BEGIN
   one_of_my_favorite_colors := true_colors (8041) (1);
END;
```

Note that this code has a small problem: if there is no record in the database table where person_id is 8041, the attempt to read its first element will raise a COLLECTION_IS_NULL exception. I must therefore trap and deal with this exception in a way that makes sense to the application.

Collection as "columns" in a database table

Using a nested table or VARRAY, you can store and retrieve nonatomic data in a single column of a table. For example, the employees table used by the HR department could store the date of birth for each employee's dependents in a single column, as shown in Table 12-1.

Table 12-1. Storing a column of dependents as a collection in a table of employees

Id (NUMBER)	Name (VARCHAR2)	Dependents_ages (Dependent_birthdate_t)
10010	Zaphod Beeblebrox	12-JAN-1763
		4-JUL-1977
		22-MAR-2021
10020	Molly Squiggly	15-NOV-1968
		15-NOV-1968

Id (NUMBER)	Name (VARCHAR2)	Dependents_ages (Dependent_birthdate_t)
10030	Joseph Josephs	
10040	Cepheus Usrbin	27-JUN-1995
		9-AUG-1996
		19-JUN-1997
10050	Deirdre Quattlebaum	21-SEP-1997

It's not terribly difficult to create such a table. First I define the collection type:

```
CREATE TYPE Dependent_birthdate_t AS VARRAY(10) OF DATE;
```

Now I can use it in the table definition:

```
CREATE TABLE employees (
   id NUMBER,
   name VARCHAR2(50),
   ...other columns...,
   dependents_ages dependent_birthdate_t
);
```

I can populate this table using the following INSERT syntax, which relies on the type's default *constructor* (discussed later in this chapter) to transform a list of dates into values of the proper datatype:

```
INSERT INTO employees VALUES (10010, 'Zaphod Beeblebrox', ...,
   dependent_birthdate_t('12-JAN-1763', '4-JUL-1977', '22-MAR-2021'));
```

Now let's look at an example of a nested table datatype as a column. When I create the outer table personality_inventory, I must tell the database what I want to call the "store table":

```
CREATE TABLE personality_inventory (
   person_id NUMBER,
   favorite_colors color_tab_t,
   date_tested DATE,
   test_results BLOB)
NESTED TABLE favorite_colors STORE AS favorite_colors_st;
```

The NESTED TABLE...STORE AS clause tells the database that I want the store table for the favorite_colors column to be called favorite_colors_st. There is no preset limit on how large this store table, which is located "out of line" (or separate from the rest of that row's data, to accommodate growth) can grow.

You cannot directly manipulate data in the store table, and any attempt to retrieve or store data directly into favorite_colors_st will generate an error. The only path by which you can read or write the store table's attributes is via the outer table. (See the discussion of collection pseudofunctions in "Working with Collections in SQL" on page 398 for a few examples of doing so.) You cannot even specify storage parameters for the store table; it inherits the physical attributes of its outermost table.

One chief difference between nested tables and VARRAYs surfaces when you use them as column datatypes. Although using a VARRAY as a column's datatype can achieve much the same result as a nested table, VARRAY data must be predeclared to be of a maximum size and is actually stored "inline" with the rest of the table's data. For this reason, Oracle Corporation says that VARRAY columns are intended for "small" arrays, and that nested tables are appropriate for "large" arrays.

Collections as attributes of an object type

In this example, I am modeling automobile specifications. Each Auto_spec_t object will include a list of manufacturer's colors in which you can purchase the vehicle:

```
CREATE TYPE auto_spec_t AS OBJECT (
   make VARCHAR2(30),
   model VARCHAR2(30),
   available_colors color_tab_t
);
```

Because there is no data storage required for the object type, it is not necessary to designate a name for the companion table at the time I issue the CREATE TYPE...AS OBJECT statement.

When the time comes to implement the type as, say, an object table, you could do this:

```
CREATE TABLE auto_specs OF auto_spec_t
   NESTED TABLE available_colors STORE AS available_colors_st;
```

This statement requires a bit of explanation. When you create a "table of objects," the database looks at the object type definition to determine what columns you want. When it discovers that one of the object type's attributes, available_colors, is in fact a nested table, the database treats this table as it did in earlier examples; in other words, it wants to know what to name the store table. So the phrase:

```
...NESTED TABLE available_colors STORE AS available_colors_st
```

says that you want the available_colors column to have a store table named available_colors_st.

See Chapter 26 for more information about Oracle object types.

Choosing a Collection Type

Which collection type makes sense for your application? In some cases, the choice is obvious. In others, there may be several acceptable choices. This section provides some guidance. Table 12-2 illustrates many of the differences between associative arrays, nested tables, and VARRAYs.

As a PL/SQL developer, I find myself leaning toward using associative arrays as a first instinct. Why is this? They involve the least amount of coding. You don't have to initialize or extend them. They have historically been the most efficient collection type (although

this distinction will probably fade over time). However, if you want to store your collection within a database table, you cannot use an associative array. The question then becomes: nested table or VARRAY?

The following guidelines will help you make your choice, although I recommend that you read the rest of the chapter first if you are not very familiar with collections already:

- If you need sparse collections (for example, for "data-smart" storage), your only practical option is an associative array. True, you could allocate and then delete elements of a nested table variable (as illustrated in "The PRIOR and NEXT Methods" on page 362), but it is inefficient to do so for anything but the smallest collections.
- If your PL/SQL application requires negative subscripts, you also have to use associative arrays.
- If you are running Oracle Database 10*g* or later, and you'd find it useful to perform high-level set operations on your collections, choose nested tables over associative arrays.
- If you want to enforce a limit to the number of rows stored in a collection, use VARRAYs.
- If you intend to store large amounts of persistent data in a column collection, your only option is a nested table. The database will then use a separate table behind the scenes to hold the collection data, so you can allow for almost limitless growth.
- If you want to preserve the order of elements stored in the collection column and if your dataset will be small, use a VARRAY.
- Here are some other indications that a VARRAY would be appropriate: you don't want to worry about deletions occurring in the middle of the data set; your data has an intrinsic upper bound; or you expect, in general, to retrieve the entire collection simultaneously.

Table 12-2. Comparing Oracle collection types

Characteristic	Associative array	Nested table	VARRAY
Dimensionality	Single	Single	Single
Usable in SQL?	No	Yes	Yes
Usable as column datatype in a table?	No	Yes; data stored "out of line" (in separate table)	Yes; data stored "inline" (in same table)
Uninitialized state	Empty (cannot be null); elements undefined	Atomically null; illegal to reference elements	Atomically null; illegal to reference elements
Initialization	Automatic, when declared	Via constructor, fetch, assignment	Via constructor, fetch, assignment
Index type	BINARY_INTEGER (and any of its subtypes) or VARCHAR2	Positive integer between 1 and 2,147,483,647	Positive integer between 1 and 2,147,483,647

Characteristic	Associative array	Nested table	VARRAY
Sparse?	Yes	Initially, no; after deletions, yes	No
Bounded?	No	Can be extended	Yes
Can assign value to any element at any time?	Yes	No; may need to EXTEND first	No; may need to EXTEND first, and cannot EXTEND past upper bound
Means of extending	Assign value to element with a new subscript	Use built-in EXTEND procedure (or TRIM to condense), with no predefined maximum	EXTEND (or TRIM), but only up to declared maximum size
Can be compared for equality?	No	Yes, Oracle Database 10*g* and later	No
Can be manipulated with set operators	No	Yes, Oracle Database 10*g* and later	No
Retains ordering and subscripts when stored in and retrieved from database?	N/A	No	Yes

Collection Methods (Built-ins)

PL/SQL offers a number of built-in functions and procedures, known as *collection methods*, that let you obtain information about and modify the contents of collections. Table 12-3 contains the complete list of these programs.

Table 12-3. Collection methods

Method (function or procedure)	Description
COUNT function	Returns the current number of elements in a collection.
DELETE procedure	Removes one or more elements from the collection. Reduces COUNT if the element is not already removed. With VARRAYs, you can delete only the entire contents of the collection.
EXISTS function	Returns TRUE or FALSE to indicate whether the specified element exists.
EXTEND procedure	Increases the number of elements in a nested table or VARRAY. Increases COUNT.
FIRST, LAST functions	Returns the smallest (FIRST) and largest (LAST) subscript in use.
LIMIT function	Returns the maximum number of elements allowed in a VARRAY.
PRIOR, NEXT functions	Returns the subscript immediately before (PRIOR) or after (NEXT) a specified subscript. You should always use PRIOR and NEXT to traverse a collection, especially if you are working with sparse (or potentially sparse) collections.
TRIM procedure	Removes collection elements from the end of the collection (highest defined subscript).

These programs are referred to as *methods* because the syntax for using the collection built-ins is different from the normal syntax used to call procedures and functions. Collection methods employ a *member method* syntax that's common in object-oriented languages such as Java.

To give you a feel for member-method syntax, consider the LAST function. It returns the greatest index value in use in the associative array. Using standard function syntax, you might expect to call LAST as follows:

```
IF LAST (company_table) > 10 THEN ... /* Invalid syntax */
```

In other words, you'd pass the associative array as an argument. In contrast, with the member-method syntax, the LAST function is a method that "belongs to" the object—in this case, the associative array. So the correct syntax for using LAST is:

```
IF company_table.LAST > 10 THEN ... /* Correct syntax */
```

The general syntax for calling these associative array built-ins is either of the following:

- An operation that takes no arguments:

  table_name.operation

- An operation that takes a row index for an argument:

  table_name.operation(index_number [*, index_number*])

The following statement, for example, returns TRUE if the 15th row of the company_tab associative array is defined:

```
company_tab.EXISTS(15)
```

The collection methods are not available from within SQL; they can be used only in PL/SQL programs.

The COUNT Method

Use COUNT to compute the number of elements defined in an associative array, nested table, or VARRAY. If elements have been DELETEd or TRIMmed from the collection, they are not included in COUNT.

The specification for COUNT is:

```
FUNCTION COUNT RETURN PLS_INTEGER;
```

Let's look at an example. Note that before I do anything with my collection, I verify that it contains some information:

```
DECLARE
   volunteer_list volunteer_list_ar := volunteer_list_ar('Steven');
BEGIN
   IF volunteer_list.COUNT > 0
   THEN
      assign_tasks (volunteer_list);
   END IF;
END;
```

Boundary considerations

If COUNT is applied to an initialized collection with no elements, it returns zero. It also returns zero if it's applied to an empty associative array.

Exceptions possible

If COUNT is applied to an uninitialized nested table or a VARRAY, it raises the COLLECTION_IS_NULL predefined exception. Note that this exception is not possible for associative arrays, which do not require initialization.

The DELETE Method

Use DELETE to remove one element, a range of elements, or all elements of an associative array, nested table, or VARRAY. DELETE without arguments removes all of the elements of a collection. DELETE(*i*) removes the *i*th element from the nested table or associative array. DELETE(*i*,*j*) removes all elements in an inclusive range beginning with *i* and ending with *j*. If the collection is a string-indexed associative array, then *i* and *j* are strings; otherwise, *i* and *j* are integers.

When you do provide actual arguments in your invocation of DELETE, it actually keeps a placeholder for the "removed" element, and you can later reassign a value to that element.

In physical terms, PL/SQL releases the memory taken up by an element only when your program deletes a sufficient number of elements to free an *entire page of memory* (unless you DELETE all the elements, which frees all the memory immediately).

When applied to VARRAYs, you can issue DELETE only without arguments (i.e., remove all rows). In other words, you cannot delete individual rows of a VARRAY, possibly making it sparse. The only way to remove a row from a VARRAY is to TRIM from the end of the collection.

The following procedure removes everything but the last element in the collection. It actually uses four collection methods (FIRST, to obtain the first defined row; LAST, to obtain the last defined row; PRIOR, to determine the next-to-last row; and DELETE, to remove all but the last row):

```
PROCEDURE keep_last (the_list IN OUT List_t)
AS
   first_elt PLS_INTEGER := the_list.FIRST;
   next_to_last_elt PLS_INTEGER := the_list.PRIOR(the_list.LAST);
BEGIN
   the_list.DELETE(first_elt, next_to_last_elt);
END;
```

Here are some additional examples:

- Delete all the rows from the names table:

```
names.DELETE;
```

- Delete the 77th row from the globals table:

```
globals.DELETE (77);
```

- Delete all the rows in the temperature readings table between the 0th row and the −15,000th row, inclusively:

```
temp_readings.DELETE (-15000, 0);
```

Boundary considerations

If *i* and/or *j* refer to nonexistent elements, DELETE attempts to "do the right thing" and will not raise an exception. For example, if you have defined elements in a nested table in index values 1, 2, and 3, then DELETE(−5,1), will remove only the item in position 1. DELETE(−5), on the other hand, will not change the collection.

Exceptions possible

If DELETE is applied to an uninitialized nested table or a VARRAY, it raises the COLLECTION_ IS_NULL predefined exception.

The EXISTS Method

Use the EXISTS method with nested tables, associative arrays, and VARRAYs to determine if the specified row exists within the collection. It returns TRUE if the element exists, and FALSE otherwise. It never returns NULL. If you have used TRIM or DELETE to remove a row that existed previously, EXISTS for that row number returns FALSE.

In the following block, I check to see if my row exists, and if so I set it to NULL:

```
DECLARE
   my_list color_tab_t := color_tab_t();
   element PLS_INTEGER := 1;
BEGIN
   ...
   IF my_list.EXISTS(element)
   THEN
     my_list(element) := NULL;
   END IF;
END;
```

Boundary considerations

If EXISTS is applied to an uninitialized (atomically null) nested table or a VARRAY, or an initialized collection with no elements, it simply returns FALSE. You can use EXISTS beyond the COUNT without raising an exception.

Exceptions possible

There are no exceptions for EXISTS.

The EXTEND Method

Adding an element to a nested table or VARRAY requires a separate allocation step. Making a "slot" in memory for a collection element is independent from assigning a value to it. If you haven't initialized the collection with a sufficient number of elements (null or otherwise), you must first use the EXTEND procedure on the variable. Do not use EXTEND with associative arrays.

EXTEND appends element(s) to a collection. EXTEND with no arguments appends a single null element. EXTEND(*n*) appends *n* null elements. EXTEND(*n*,*i*) appends *n* elements and sets each to the same value as the *i*th element; this form of EXTEND is required for collections with NOT NULL elements.

Here is the overloaded specification of EXTEND:

```
PROCEDURE EXTEND (n PLS_INTEGER:=1);
PROCEDURE EXTEND (n PLS_INTEGER, i PLS_INTEGER);
```

In the following example, the push procedure extends my list by a single row and populates it:

```
PROCEDURE push (the_list IN OUT List_t, new_value IN VARCHAR2)
AS
BEGIN
   the_list.EXTEND;
   the_list(the_list.LAST) := new_value;
END;
```

I can also use EXTEND to add 10 new rows to my list, all with the same value. First I extend a single row and populate explicitly. Then I extend again, this time by nine rows, and specify the row number with new_value as the initial value for all my new rows:

```
PROCEDURE push_ten (the_list IN OUT List_t, new_value IN VARCHAR2)
AS
   l_copyfrom PLS_INTEGER;
BEGIN
   the_list.EXTEND;
   l_copyfrom := the_list.LAST;
   the_list(l_copyfrom) := new_value;
   the_list.EXTEND (9, l_copyfrom);
END;
```

Boundary considerations

If *n* is null, EXTEND will do nothing.

Exceptions possible

If EXTEND is applied to an uninitialized nested table or a VARRAY, it raises the COLLECTION_IS_NULL predefined exception. An attempt to EXTEND a VARRAY beyond its declared limit raises the SUBSCRIPT_BEYOND_LIMIT exception.

The FIRST and LAST Methods

Use the FIRST and LAST methods with nested tables, associative arrays, and VARRAYs to return, respectively, the lowest and highest index values defined in the collection. For string-indexed associative arrays, these methods return strings; "lowest" and "highest" are determined by the ordering of the character set in use in that session. For all other collection types, these methods return integers.

The specifications for these functions follow:

```
FUNCTION FIRST RETURN PLS_INTEGER | VARCHAR2;
FUNCTION LAST RETURN PLS_INTEGER | VARCHAR2;
```

For example, the following code scans from the start to the end of my collection:

```
FOR indx IN holidays.FIRST .. holidays.LAST
LOOP
   send_everyone_home (indx);
END LOOP;
```

Please remember that this kind of loop will only work (i.e., not raise a NO_DATA_FOUND exception) if the collection is densely populated.

In the next example, I use COUNT to concisely specify that I want to append a row to the end of an associative array. I use a cursor FOR loop to transfer data from the database to an associative array of records. When the first record is fetched, the companies collection is empty, so the COUNT operator will return 0:

```
FOR company_rec IN company_cur
LOOP
   companies ((companies.COUNT) + 1).company_id
      := company_rec.company_id;
END LOOP;
```

Boundary considerations

FIRST and LAST return NULL when they are applied to initialized collections that have no elements. For VARRAYs, which have at least one element, FIRST is always 1, and LAST is always equal to COUNT.

Exceptions possible

If FIRST and LAST are applied to an uninitialized nested table or a VARRAY, they raise the COLLECTION_ IS_NULL predefined exception.

The LIMIT Method

Use the LIMIT method to determine the maximum number of elements that can be defined in a VARRAY. This function will return NULL if it is applied to initialized nested tables or to associative arrays. The specification for LIMIT is:

```
FUNCTION LIMIT RETURN PLS_INTEGER;
```

The following conditional expression makes sure that there is still room in my VARRAY before extending:

```
IF my_list.LAST < my_list.LIMIT
THEN
   my_list.EXTEND;
END IF;
```

Boundary considerations

There are no boundary considerations for LIMIT.

Exceptions possible

If LIMIT is applied to an uninitialized nested table or a VARRAY, it raises the COLLECTION_ IS_NULL predefined exception.

The PRIOR and NEXT Methods

Use the PRIOR and NEXT methods with nested tables, associative arrays, and VARRAYs to navigate through the contents of a collection.

PRIOR returns the next-lower index value in use relative to *i*; NEXT returns the next higher. In the following example, this function returns the sum of elements in a list_t collection of numbers:

```
FUNCTION compute_sum (the_list IN list_t) RETURN NUMBER
AS
   row_index PLS_INTEGER := the_list.FIRST;
   total NUMBER := 0;
BEGIN
   LOOP
      EXIT WHEN row_index IS NULL;
      total := total + the_list(row_index);
      row_index := the_list.NEXT(row_index);
   END LOOP;
   RETURN total;
END compute_sum;
```

Here is that same program working from the last to the very first defined row in the collection:

```
FUNCTION compute_sum (the_list IN list_t) RETURN NUMBER
AS
   row_index PLS_INTEGER := the_list.LAST;
   total NUMBER := 0;
BEGIN
   LOOP
      EXIT WHEN row_index IS NULL;
      total := total + the_list(row_index);
      row_index := the_list.PRIOR(row_index);
   END LOOP;
   RETURN total;
END compute_sum;
```

In this case, it doesn't matter which direction you move through the collection. In other programs, though, it can make a big difference.

Boundary considerations

If PRIOR and NEXT are applied to initialized collections that have no elements, they return NULL. If *i* is greater than or equal to COUNT, NEXT returns NULL; if *i* is less than or equal to FIRST, PRIOR returns NULL.

Through Oracle Database 12*c*, if the collection has elements, and *i* is greater than COUNT, PRIOR returns LAST; if *i* is less than FIRST, NEXT returns FIRST. However, do not rely on this behavior in future database versions.

Exceptions possible

If PRIOR and NEXT are applied to an uninitialized nested table or a VARRAY, they raise the COLLECTION_ IS_NULL predefined exception.

The TRIM Method

Use TRIM to remove *n* elements from the end of a nested table or VARRAY. Without arguments, TRIM removes exactly one element. As I've already mentioned, confusing behavior occurs if you combine DELETE and TRIM actions on a collection; for example, if an element that you are trimming has previously been DELETEd, TRIM "repeats" the deletion but counts this as part of *n*, meaning that you may be TRIMming fewer actual elements than you think.

Attempting to TRIM an associative array will produce a compile-time error.

The specification for TRIM is:

```
PROCEDURE TRIM (n PLS_INTEGER:=1);
```

The following function pops the last value off of a list and returns it to the invoking block. The "pop" action is implemented by trimming the collection by a single row after extracting the value:

```
FUNCTION pop (the_list IN OUT list_t) RETURN VARCHAR2
AS
   l_value VARCHAR2(30);
BEGIN
   IF the_list.COUNT >= 1
   THEN
      /* Save the value of the last element in the collection
      || so it can be returned
      */
      l_value := the_list(the_list.LAST);
      the_list.TRIM;
   END IF;
   RETURN l_value;
END;
```

Boundary considerations

If *n* is null, TRIM will do nothing.

Exceptions possible

The TRIM method will raise the SUBSCRIPT_BEYOND_COUNT predefined exception if you attempt to TRIM more elements than actually exist. If TRIM is applied to an uninitialized nested table or a VARRAY, it raises the COLLECTION_IS_NULL predefined exception.

If you use TRIM and DELETE on the same collection, you can get some very surprising results. Consider this scenario: if you DELETE an element at the end of a nested table variable and then do a TRIM on the same variable, how many elements have you removed? You might think that you have removed two elements, but in fact you have removed only one. The placeholder that is left by DELETE is what TRIM acts upon. To avoid confusion, Oracle Corporation recommends using either DELETE or TRIM, but not both, on a given collection.

Working with Collections

You now know about the different types of collections and the collection methods. You have seen some examples of working with associative arrays, nested tables, and VARRAYs. Now it is time to dive into the details of manipulating collections in your programs. Topics in this section include:

- Exception handling with collections
- Declaring collection types
- Declaring and initializing collection variables
- Assigning values to collections
- Using collections of complex datatypes, such as collections of other collections
- Working with sequential and nonsequential associative arrays
- The power of string-indexed collections
- Working with PL/SQL collections inside SQL statements

Declaring Collection Types

Before you can work with a collection, you must declare it, and that declaration must be based on a collection type. So the first thing you must learn to do is define a collection type.

There are two ways to create user-defined collection types:

- You can declare the collection type within a PL/SQL program using the TYPE statement. This collection type will then be available for use within the block in which the TYPE is defined. If the TYPE is defined in a package specification, then it is available to any program whose schema has EXECUTE authority on the package.
- You can define a nested table type or VARRAY type as a schema-level object within the Oracle database by using the CREATE TYPE command. This TYPE can then be used as the datatype for columns in database tables and attributes of object types, and to declare variables in PL/SQL programs. Any program in a schema with EXECUTE authority on the TYPE can reference the TYPE.

Declaring an associative array collection type

The TYPE statement for an associative array has the following format:

```
TYPE table_type_name IS TABLE OF datatype [ NOT NULL ]
   INDEX BY index_type;
```

where *table_type_name* is the name of the collection you are creating, *datatype* is the datatype of the single column in the collection, and *index_type* is the datatype of the index used to organize the contents of the collection. You can optionally specify that the collection be NOT NULL, meaning that every row in the table must have a value.

The rules for the table type name are the same as for any identifier in PL/SQL: the name may be up to 30 characters in length, it must start with a letter, and it may include a few special characters (hash sign, underscore, and dollar sign)—and if you surround the name with double quotes, you can use up to 30 characters of *anything*.

Almost any valid PL/SQL datatype can be used as the datatype of the collection's elements. You can use most scalar base datatypes, subtypes, anchored types, and user-defined types. Exceptions include REF CURSOR types (you cannot have a collection of cursor variables) and exceptions.

The *index_type* of the collection determines the type of data you can use to specify the location of the data you are placing in the collection. Prior to Oracle9*i* Database Release 2, the only way you could specify an index for an associative array (a.k.a. index-by table) was:

```
INDEX BY PLS_INTEGER
```

Starting with Oracle9*i* Database Release 2, the INDEX BY datatype can be BINARY_INTEGER, any of its subtypes, VARCHAR2(*n*), or %TYPE against a VARCHAR2 column or variable. In other words, any of the following INDEX BY clauses are now valid:

```
INDEX BY BINARY_INTEGER
INDEX BY PLS_INTEGER
INDEX BY POSITIVE
INDEX BY NATURAL
INDEX BY SIGNTYPE /* Only three index values - -1, 0, and 1 - allowed! */
INDEX BY VARCHAR2(32767)
INDEX BY table.column%TYPE
INDEX BY cursor.column%TYPE
INDEX BY package.variable%TYPE
INDEX BYpackage.subtype
```

Here are some examples of associative array type declarations:

```
-- A list of dates
TYPE birthdays_tt IS TABLE OF DATE INDEX BY PLS_INTEGER;

-- A list of company IDs
TYPE company_keys_tt IS TABLE OF company.company_id%TYPE NOT NULL
   INDEX BY PLS_INTEGER;

-- A list of book records; this structure allows you to make a "local"
-- copy of the book table in your PL/SQL program.
TYPE booklist_tt IS TABLE OF books%ROWTYPE
   INDEX BY NATURAL;
```

```
-- Each collection is organized by the author name.
TYPE books_by_author_tt IS TABLE OF books%ROWTYPE
   INDEX BY books.author%TYPE;

-- A collection of collections
TYPE private_collection_tt IS TABLE OF books_by_author_tt
   INDEX BY VARCHAR2(100);
```

Notice that in the preceding example I declared a very generic type of collection (a list of dates), but gave it a very specific name: birthdays_tt. There is, of course, just one way to declare an associative array type of dates. Rather than have a plethora of collection TYPE definitions that differ only by name scattered throughout your application, you might consider creating a single package that offers a set of predefined, standard collection types. Here is an example, available in the *colltypes.pks* file on the book's website:

```
/* File on web: colltypes.pks */
PACKAGE collection_types
IS
   -- Associative array types
   TYPE boolean_aat IS TABLE OF BOOLEAN INDEX BY PLS_INTEGER;
   TYPE date_aat IS TABLE OF DATE INDEX BY PLS_INTEGER;
   TYPE pls_integer_aat IS TABLE OF PLS_INTEGER INDEX BY PLS_INTEGER;
   TYPE number_aat IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
   TYPE identifier_aat IS TABLE OF VARCHAR2(30)
       INDEX BY PLS_INTEGER;
   TYPE vcmax_aat IS TABLE OF VARCHAR2(32767)
       INDEX BY PLS_INTEGER;

   -- Nested table types
   TYPE boolean_ntt IS TABLE OF BOOLEAN;
   TYPE date_ntt IS TABLE OF DATE;
   TYPE pls_integer_ntt IS TABLE OF PLS_INTEGER;
   TYPE number_ntt IS TABLE OF NUMBER;
   TYPE identifier_ntt IS TABLE OF VARCHAR2(30);
   TYPE vcmax_ntt IS TABLE OF VARCHAR2(32767)
END collection_types;
/
```

With such a package in place, you can grant EXECUTE authority to PUBLIC, and then all developers can use the packaged TYPEs to declare their own collections. Here is an example:

```
DECLARE
   family_birthdays collection_types.date_aat;
```

Declaring a nested table or VARRAY

As with associative arrays, you must define a type before you can declare an actual nested table or VARRAY. You can define these types either in the database or in a PL/SQL block.

To create a nested table datatype that lives in the database (and not just your PL/SQL code), specify:

```
CREATE [ OR REPLACE ] TYPE type_name AS | IS
   TABLE OF element_datatype [ NOT NULL ];
```

To create a VARRAY datatype that lives in the database (and not just your PL/SQL code), specify:

```
CREATE [ OR REPLACE ] TYPE type_name AS | IS
   VARRAY (max_elements) OF element_datatype [ NOT NULL ];
```

To drop a type, specify:

```
DROP TYPE type_name [ FORCE ];
```

To declare a nested table datatype in PL/SQL, use the declaration:

```
TYPE type_name IS TABLE OF element_datatype [ NOT NULL ];
```

To declare a VARRAY datatype in PL/SQL, use the declaration:

```
TYPE type_name IS VARRAY (max_elements)
   OF element_datatype [ NOT NULL ];
```

In these declarations:

OR REPLACE
: Allows you to rebuild an existing type. Including REPLACE, rather than dropping and re-creating the type, preserves all existing grants of privileges.

type_name
: Is a legal SQL or PL/SQL identifier. This will be the identifier to which you refer later when you use it to declare variables or columns.

element_datatype
: Is the type of the collection's elements. All elements are of a single type, which can be most scalar datatypes, an object type, or a REF object type. When defining a schema-level type, this datatype must be a SQL datatype (no Booleans allowed!).

NOT NULL
: Indicates that a variable of this type cannot have any null elements. However, the collection can be atomically null (uninitialized).

max_elements
: Is the maximum number of elements allowed in the VARRAY.

FORCE
: Tells the database to drop the type even if there is a reference to it in another type. For example, if an object type definition uses a particular collection type, you can still drop the collection type using the FORCE keyword.

To execute the CREATE TYPE statement, you must follow it with a slash (/), just as if you were creating a procedure, function, or package.

Note that the only syntactic difference between declaring nested table types and declaring associative array types in a PL/SQL program is the absence of the INDEX BY clause for nested table types.

The syntactic differences between nested table and VARRAY type declarations are:

- The use of the keyword VARRAY
- The limit on VARRAY's number of elements

Changing a nested table of VARRAY characteristics

If you have created a nested table or VARRAY type in the database, you can use the ALTER TYPE command to change several of the type's characteristics.

Use the ALTER TYPE...MODIFY LIMIT syntax to increase the number of elements of a VARRAY type. Here is an example:

```
ALTER TYPE list_vat MODIFY LIMIT 100 INVALIDATE;
/
```

When the element type of a VARRAY or nested table type is variable character, RAW, or numeric, you can increase the size of the variable character or RAW type or increase the precision of the numeric type. Here is an example:

```
CREATE TYPE list_vat AS VARRAY(10) OF VARCHAR2(80);
/
ALTER TYPE list_vat MODIFY ELEMENT TYPE VARCHAR2(100) CASCADE;
/
```

The INVALIDATE and CASCADE options are provided to either invalidate all dependent objects or propagate the change to both the type and any table dependents.

Declaring and Initializing Collection Variables

Once you have created your collection type, you can reference it to declare an instance of that type: the actual collection variable. The general format for a collection declaration is:

```
collection_name collection_type [:= collection_type (...)];
```

where *collection_name* is the name of the collection, and *collection_type* is the name of both the previously declared collection type and (if nested table or VARRAY) a constructor function of the same name.

A constructor has the same name as the type and accepts as arguments a comma-separated list of elements. When you are declaring a nested table or VARRAY, you *must* initialize the collection before using it. Otherwise, you will receive this error:

```
ORA-06531: Reference to uninitialized collection
```

In the following example I create a general collection type to emulate the structure of the company table. I then declare two different collections based on that type:

```
DECLARE
   TYPE company_aat IS TABLE OF company%ROWTYPE INDEX BY PLS_INTEGER;
   premier_sponsor_list company_aat;
   select_sponsor_list company_aat;
BEGIN
   ...
END;
```

If I declare a nested table or VARRAY, I can also immediately initialize the collection by calling its constructor function. Here is an example:

```
DECLARE
   TYPE company_aat IS TABLE OF company%ROWTYPE;
   premier_sponsor_list company_aat := company_aat();
BEGIN
   ...
END;
```

I could also choose to initialize the nested table in my executable section:

```
DECLARE
   TYPE company_aat IS TABLE OF company%ROWTYPE;
   premier_sponsor_list company_aat;
BEGIN
   premier_sponsor_list:= company_aat();
END;
```

I simply must ensure that it is initialized before I try to use the collection. Associative arrays do not need to be initialized before you assign values to them (and indeed cannot be initialized in this way). As you can see, declaring collection variables, or instances of a collection type, is no different from declaring other kinds of variables: simply provide a name, type, and optional default or initial value.

Let's take a closer look at nested table and VARRAY initialization. The previous example showed you how to initialize a collection by calling a constructor function without any parameters. You can also provide an initial set of values. Suppose that I create a schema-level type named color_tab_t:

```
CREATE OR REPLACE TYPE color_tab_t AS TABLE OF VARCHAR2(30)
```

and then declare some PL/SQL variables based on that type:

```
DECLARE
   my_favorite_colors color_tab_t := color_tab_t();
   his_favorite_colors color_tab_t := color_tab_t('PURPLE');
   her_favorite_colors color_tab_t := color_tab_t('PURPLE', 'GREEN');
```

In the first declaration, the collection is initialized as empty; it contains no rows. The second declaration assigns a single value, PURPLE, to row 1 of the nested table. The third declaration assigns two values, PURPLE and GREEN, to rows 1 and 2 of that nested table.

Because I have not assigned any values to my_favorite_colors in the call to the constructor, I will have to *extend* it before I can put elements into it. The his and her collections already have been extended implicitly as needed by the constructor values list.

Assignment via a constructor function is bound by the same constraints that you will encounter in direct assignments. If, for example, your VARRAY has a limit of five elements and you try to initialize it via a constructor with six elements, the database will raise an *ORA-06532: Subscript outside of limit* error.

Initializing implicitly during direct assignment

You can copy the entire contents of one collection to another as long as both are built from the exact same collection type (two different collection types based on the same datatype will *not* work). When you do so, initialization comes along "for free."

Here's an example illustrating the implicit initialization that occurs when I assign wedding_colors to be the value of earth_colors:

```
DECLARE
   earth_colors color_tab_t := color_tab_t ('BRICK', 'RUST', 'DIRT');
   wedding_colors color_tab_t;
BEGIN
   wedding_colors := earth_colors;
   wedding_colors(3) := 'CANVAS';
END;
```

This code initializes wedding_colors and creates three elements that match those in earth_colors. These are independent variables rather than pointers to identical values; changing the third element of wedding_colors to CANVAS does not have any effect on the third element of earth_colors.

This kind of direct assignment is not possible when datatypes are merely "type compatible." Even if you have created two different types with the exact same definition, the fact that they have different names makes them different types. Thus, the following block of code fails to compile:

```
DECLARE
   TYPE tt1 IS TABLE OF employees%ROWTYPE;
```

```
   TYPE tt2 IS TABLE OF employees%ROWTYPE;
   t1   tt1;
   t2   tt2 := tt2();
BEGIN
   /* Fails with error "PLS-00382: expression is of wrong type" */
   t1 := t2;
END;
```

Initializing implicitly via FETCH

If you use a collection as a type in a database table, the Oracle database provides some very elegant ways of moving the collection between PL/SQL and the table. As with direct assignment, when you use FETCH or SELECT INTO to retrieve a collection and drop it into a collection variable, you get automatic initialization of the variable. Collections can turn out to be incredibly useful!

Although I mentioned this briefly in an earlier example, let's take a closer look at how you can read an entire collection in a single fetch. First, I want to create a table containing a collection and populate it with a couple of values:

```
CREATE TABLE color_models (
     model_type VARCHAR2(12)
   , colors color_tab_t
   )
   NESTED TABLE colors STORE AS color_model_colors_tab
/

BEGIN
   INSERT INTO color_models
   VALUES ('RGB', color_tab_t ('RED','GREEN','BLUE'));
END;
/
```

Now I can show off the neat integration features. With one trip to the database, I can retrieve all the values of the colors column for a given row and deposit them into a local variable:

```
DECLARE
   l_colors color_tab_t;
BEGIN
   /* Retrieve all the nested values in a single fetch.
   || This is the cool part.
   */
   SELECT colors INTO l_colors FROM color_models
      WHERE model_type = 'RGB';
   ...
END;
```

Pretty neat, huh? Here are a few important things to notice:

- The database, not the programmer, assigns the subscripts of l_colors when fetched from the database.
- The database's assigned subscripts begin with 1 (as opposed to 0, as in some other languages) and increment by 1; this collection is always densely filled (or empty).
- Fetching satisfies the requirement to initialize the local collection variable before assigning values to elements. I didn't initialize l_colors with a constructor, but PL/SQL knew how to deal with it.

You can also make changes to the contents of the nested table and just as easily move the data back into a database table. Just to be mischievous, let's create a Fuchsia-Green-Blue color model:

```
DECLARE
   color_tab color_tab_t;
BEGIN
   SELECT colors INTO color_tab FROM color_models
      WHERE model_type = 'RGB';

   FOR element IN 1..color_tab.COUNT
   LOOP
      IF color_tab(element) = 'RED'
      THEN
         color_tab(element) := 'FUCHSIA';
      END IF;
   END LOOP;

   /* Here is the cool part of this example. Only one insert
   || statement is needed -- it sends the entire nested table
   || back into the color_models table in the database. */

   INSERT INTO color_models VALUES ('FGB', color_tab);
END;
```

VARRAY integration

Does this database-to-PL/SQL integration work for VARRAYs too? You bet, although there are a couple of differences.

First of all, realize that when you store and retrieve the contents of a nested table in the database, the Oracle database makes no promises about preserving the order of the elements. This makes sense because the server is just putting the nested data into a store table behind the scenes, and we all know that relational databases don't give two hoots about row order. By contrast, storing and retrieving the contents of a VARRAY *do* preserve the order of the elements.

Preserving the order of VARRAY elements is a fairly useful capability. It makes it possible to embed meaning in the order of the data, which is something you cannot do in a pure relational database. For example, if you want to store someone's favorite colors in rank

order, you can do it with a single VARRAY column. Every time you retrieve the column collection, its elements will be in the same order as when you last stored it. In contrast, abiding by a pure relational model, you would need two columns: one for an integer corresponding to the rank and one for the color.

This order preservation of VARRAYs suggests some possibilities for interesting utility functions. For example, you could fairly easily code a tool that would allow the insertion of a new "favorite" at the low end of the list by "shifting up" all the other elements.

A second difference between integration of nested tables and integration of VARRAYs with the database is that some SELECT statements you could use to fetch the contents of a nested table will have to be modified if you want to fetch a VARRAY. (See "Working with Collections in SQL" on page 398 for some examples.)

Populating Collections with Data

A collection is empty after initialization. No elements are defined within it. A collection is, in this way, very much like a relational table. You define an element by assigning a value to it. You can do this assignment through the standard PL/SQL assignment operation, by fetching data from one or more relational tables into a collection, with a RETURNING BULK COLLECT clause, or by performing an aggregate assignment (in essence, copying one collection to another).

If you are working with associative arrays, you can assign a value (of the appropriate type) to any valid index value in the collection. If the index type of the associative array is an integer, then the index value must be between -2^{31} and $2^{31} - 1$. The simple act of assigning the value creates the element and deposits the value at that index.

In contrast to associative arrays, you can't assign values to arbitrarily numbered subscripts of nested tables and VARRAYs; instead, the indexes (at least initially) are monotonically increasing integers, assigned by the PL/SQL engine. That is, if you initialize *n* elements, they will have subscripts 1 through *n*—and those are the only rows to which you can assign a value.

Before you try to assign a value to an index value in a nested table or VARRAY, you must make sure that (1) the collection has been initialized, and (2) that index value has been defined. Use the EXTEND operator, discussed earlier in this chapter, to make new index values available in nested tables and VARRAYs.

Using the assignment operator

You can assign values to a collection with the standard assignment operator of PL/SQL, as shown here:

```
countdown_test_list (43) := 'Internal pressure';
company_names_table (last_name_row + 10) := 'Johnstone Clingers';
```

You can use this same syntax to assign an entire record or complex datatype to an index value in the collection, as you see here:

```
DECLARE
   TYPE emp_copy_t IS TABLE OF employees%ROWTYPE;
   l_emps emp_copy_t := emp_copy_t();
   l_emprec employees%ROWTYPE;
BEGIN
   l_emprec.last_name := 'Steven';
   l_emprec.salary := 10000;
   l_emps.EXTEND;
   l_emps (l_emps.LAST) := l_emprec;
END;
```

As long as the structure of data on the right side of the assignment matches that of the collection type, the assignment will complete without error.

What index values can I use?

When you assign data to an associative array, you must specify the location (index value) in the collection. The type of value, and valid range of values, that you use to indicate this location depend on how you defined the INDEX BY clause of the associative array, and are explained in the following table.

INDEX BY clause	Minimum value	Maximum value
INDEX BY BINARY_INTEGER	-2^{31}	$2^{31} - 1$
INDEX BY PLS_INTEGER	-2^{31}	$2^{31} - 1$
INDEX BY SIMPLE_INTEGER	-2^{31}	$2^{31} - 1$
INDEX BY NATURAL	0	$2^{31} - 1$
INDEX BY POSITIVE	1	$2^{31} - 1$
INDEX BY SIGNTYPE	–1	1
INDEX BY VARCHAR2(*n*)	Any string within specified length	Any string within specified length

You can also index by any subtype of the preceding, or use a type anchored to a VARCHAR2 database column (e.g., *table_name.column_name*%TYPE).

Aggregate assignments

You can also perform an "aggregate assignment" of the contents of an entire collection to another collection of exactly the same type. Here is an example of such a transfer:

```
1  DECLARE
2     TYPE name_t IS TABLE OF VARCHAR2(100) INDEX BY PLS_INTEGER;
3     old_names name_t;
4     new_names name_t;
5  BEGIN
6     /* Assign values to old_names table */
7     old_names(1) := 'Smith';
```

```
 8     old_names(2) := 'Harrison';
 9
10     /* Assign values to new_names table */
11     new_names(111) := 'Hanrahan';
12     new_names(342) := 'Blimey';
13
14     /* Transfer values from new to old */
15     old_names := new_names;
16
17     /* This statement will display 'Hanrahan' */
18     DBMS_OUTPUT.PUT_LINE (
19       old_names.FIRST || ': ' || old_names(old_names.FIRST));
20  END;
```

The output is:

```
111: Hanrahan
```

A collection-level assignment completely replaces the previously defined rows in the collection. In the preceding example, rows 1 and 2 in old_names are defined before the last, aggregate assignment.

After the assignment, only rows 111 and 342 in the old_names collection have values.

Assigning rows from a relational table

You can also populate rows in a collection by querying data from a relational table. The assignment rules described earlier in this section apply to SELECT-driven assignments. The following examples demonstrate various ways of copying data from a relational table into a collection.

I can use an implicit SELECT INTO to populate a single row of data in a collection:

```
DECLARE
   TYPE emp_copy_t IS TABLE OF employees%ROWTYPE;
   l_emps emp_copy_t := emp_copy_t();
BEGIN
   l_emps.EXTEND;
   SELECT *
     INTO l_emps (1)
     FROM employees
    WHERE employee_id = 7521;
END;
```

I can use a cursor FOR loop to move multiple rows into a collection, populating those rows sequentially:

```
DECLARE
   TYPE emp_copy_t IS TABLE OF employees%ROWTYPE;
   l_emps emp_copy_t := emp_copy_t();
BEGIN
   FOR emp_rec IN (SELECT * FROM employees)
   LOOP
```

```
      l_emps.EXTEND;
      l_emps (l_emps.LAST) := emp_rec;
   END LOOP;
END;
```

I can also use a cursor FOR loop to move multiple rows into a collection, populating those rows *nonsequentially*. In this case, I will switch to using an associative array, so that I can assign rows randomly—that is, using the primary key value of each row in the database as the row number in my collection:

```
DECLARE
   TYPE emp_copy_t IS TABLE OF employees%ROWTYPE INDEX BY PLS_INTEGER;
   l_emps emp_copy_t;
BEGIN
   FOR emp_rec IN (SELECT * FROM employees)
   LOOP
      l_emps (emp_rec.employee_id) := emp_rec;
   END  LOOP;
END;
```

I can also use BULK COLLECT (described in Chapter 21) to retrieve all the rows of a table in a single assignment step, depositing the data into any of the three types of collections. When using a nested table or VARRAY, you do *not* need to explicitly initialize the collection. Here is an example:

```
DECLARE
   TYPE emp_copy_nt IS TABLE OF employees%ROWTYPE;
   l_emps emp_copy_nt;
BEGIN
   SELECT * BULK COLLECT INTO l_emps FROM employees;
END;
```

Advantage of nonsequential population of collection

For anyone used to working with traditional arrays, the idea of populating your collection nonsequentially may seem strange. Why would you do such a thing? Consider the following scenario.

In many applications, you will find yourself writing and executing the same queries over and over again. In some cases, the queries are retrieving static data, such as codes and descriptions that rarely (if ever) change. If the data isn't changing—especially during a user session—then why would you want to keep querying the information from the database? Even if the data is cached in the system global area (SGA), you still need to visit the SGA, confirm that the query has already been parsed, find that information in the data buffers, and finally return it to the session program area (the program global area, or PGA).

Here's an idea: set as a rule that for a given static lookup table, a user will never query a row from the table more than once in a session. After the first time, it will be stored in

the session's PGA and be instantly available for future requests. This is very easy to do with collections. Essentially, you use the collection's index as an intelligent key.

Let's take a look at an example. I have a hairstyles table that contains a numeric code (primary key) and a description of the hairstyle (e.g., Pageboy). These styles are timeless and rarely change.

Here is the body of a package that uses a collection to cache code-hairstyle pairs and that minimizes trips to the database:

```
 1 PACKAGE BODY justonce
 2  IS
 3     TYPE desc_t
 4     IS
 5        TABLE OF hairstyles.description%TYPE
 6           INDEX BY PLS_INTEGER;
 7
 8     descriptions   desc_t;
 9
10     FUNCTION description (code_in IN hairstyles.code%TYPE)
11        RETURN hairstyles.description%TYPE
12     IS
13
14        FUNCTION desc_from_database
15           RETURN hairstyles.description%TYPE
16        IS
17           l_description   hairstyles.description%TYPE;
18        BEGIN
19           SELECT description
20             INTO l_description
21             FROM hairstyles
22            WHERE code = code_in;
23           RETURN l_description;
24        END;
25     BEGIN
26        RETURN descriptions (code_in);
27     EXCEPTION
28        WHEN NO_DATA_FOUND
29        THEN
30           descriptions (code_in) := desc_from_database ();
31           RETURN descriptions (code_in);
32     END;
33  END justonce;
```

The following table describes the interesting aspects of this program.

Line(s)	Description
3–8	Declare a collection type and the collection to hold my cached descriptions.
10–11	Header of my retrieval function. The interesting thing about the header is that it is not interesting at all. There is no indication that this function is doing anything but the typical query against the database to retrieve the description for the code. The implementation is hidden, which is just the way you want it.

Line(s)	Description
15–25	That very traditional query from the database. But in this case, it is just a private function within my main function, which is fitting because it is not the main attraction.
27	The entire execution section! Simply return the description that is stored in the row indicated by the code number. The first time I run this function for a given piece of code, the row will not be defined, so PL/SQL will raise a NO_DATA_FOUND exception (see lines 28–31). For all subsequent requests for this code, however, the row is defined, and the function returns the value immediately.
29–32	So the data hasn't yet been queried in this session. Fine. Trap the error, look up the description from the database, and deposit it in the collection. Then return that value. Now I am set to divert all subsequent lookup attempts.

So how much of a difference does this caching make? I ran some tests on my laptop and found that it took just under two seconds to execute 10,000 queries against the hairstyles table. That's efficient, no doubt about it. Yet it took only 0.1 seconds to retrieve that same information 10,000 times using the preceding function. That's more than an order of magnitude improvement!

Here are some final notes on the collection caching technique:

- This technique is a classic tradeoff between CPU and memory. Each session has its own copy of the collection (this is program data and is stored in the PGA). If you have 10,000 users, the total memory required for these 10,000 small caches could be considerable.
- Consider using this technique with any of the following scenarios: small, static tables in a multiuser application; large, static tables of which a given user will access only a small portion; and manipulation of large tables in a batch process (just a single CONNECT taking up possibly a lot of memory).

The concept and implementation options for caching are explored in much greater depth in Chapter 21.

Accessing Data Inside a Collection

There generally isn't much point to putting information into a collection unless you intend to use or access that data. There are several things you need to keep in mind when accessing data inside a collection:

- If you try to read an undefined index value in a collection, the database raises the NO_DATA_FOUND exception. One consequence of this rule is that you should avoid using numeric FOR loops to scan the contents of a collection unless you are certain it is, and always will be, densely filled (no undefined index values between FIRST and LAST). If that collection is not densely filled, the database will fail with NO_DATA_FOUND as soon as it hits a gap between the values returned by the FIRST and LAST methods.

- If you try to read a row that is beyond the limit of EXTENDed rows in a table or VARRAY, the database raises the following exception:

  ```
  ORA-06533: Subscript beyond count
  ```

  When working with nested tables and VARRAYs, you should always make sure that you have extended the collection to encompass the row you want to assign or read.

- If you try to read a row whose index is beyond the limit of the VARRAY type definition, the database raises the following exception:

  ```
  ORA-06532: Subscript outside of limit
  ```

  Remember: you can always call the LIMIT method to find the maximum number of rows that are allowed in a VARRAY. Because the subscript always starts at 1 in this type of collection, you can then easily determine if you still have room for more data in the data structure.

Beyond these cautionary tales, it is very easy to access individual rows in a collection: simply provide the subscript (or subscripts—see "Collections of Complex Datatypes" on page 385 for the syntax needed for collections of collections) after the name of the collection.

Using String-Indexed Collections

Oracle9*i* Database Release 2 greatly expanded the datatypes developers can specify as the index type for associative arrays. VARCHAR2 offers the most flexibility and potential. Since with this datatype I can index by string, I can essentially index by just about *anything*, as long as it can be converted into a string of no more than 32,767 bytes.

Here is a block of code that demonstrates the basics:

```
/* File on web: string_indexed.sql */
DECLARE
   SUBTYPE location_t IS VARCHAR2(64);
   TYPE population_type IS TABLE OF NUMBER INDEX BY location_t;

   l_country_population population_type;
   l_continent_population population_type;

   l_count PLS_INTEGER;
   l_location location_t;
BEGIN
   l_country_population('Greenland') := 100000;
   l_country_population('Iceland') := 750000;

   l_continent_population('Australia') := 30000000;
   l_continent_population('Antarctica') := 1000;
   l_continent_population('antarctica') := 1001;

   l_count := l_country_population.COUNT;
```

```
    DBMS_OUTPUT.PUT_LINE ('COUNT = ' || l_count);

    l_location := l_continent_population.FIRST;
    DBMS_OUTPUT.PUT_LINE ('FIRST row = ' || l_location);
    DBMS_OUTPUT.PUT_LINE ('FIRST value = ' || l_continent_population(l_location));

    l_location := l_continent_population.LAST;
    DBMS_OUTPUT.PUT_LINE ('LAST row = ' || l_location);
    DBMS_OUTPUT.PUT_LINE ('LAST value = ' || l_continent_population(l_location));
END;
```

Here is the output from the script:

```
COUNT = 2
FIRST row = Antarctica
FIRST value = 1000
LAST row = antarctica
LAST value = 1001
```

Points of interest from this code:

- With a string-indexed collection, the values returned by calls to the FIRST, LAST, PRIOR, and NEXT methods are *strings* and not integers.
- Notice that "antarctica" is last, coming after "Antarctica" and "Australia". That's because lowercase letters have a higher ASCII code than uppercase letters. The order in which *your* strings will be stored in your associative array will be determined by your character set.
- There is really no difference in syntax between using string-indexed and integer-indexed collections.
- I carefully defined a subtype, location_t, which I then used as the index type in my collection type declaration, and also to declare the l_location variable. You will find that when you work with string indexed collections, especially multilevel collections, subtypes will be very helpful reminders of precisely *what* data you are using for your index values.

The following sections offer other examples demonstrating the usefulness of this feature.

Simplifying algorithmic logic with string indexes

Careful use of string indexed collections can greatly simplify your programs; in essence, you are transferring complexity from your algorithms to the data structure (and leaving it to the database) to do the "heavy lifting." The following example will give you a clear sense of that transfer.

Through much of 2006 and 2007, I led the effort to build an automated testing tool for PL/SQL, Quest Code Tester for Oracle. One key benefit of this tool is that it generates a test package from your descriptions of the expected behavior of a program. As I gen-

erate the test code, I need to keep track of the names of variables that I have declared so that I do not inadvertently declare another variable with the same name.

My first pass at building a string_tracker package looked like this:

```
      /* File on web: string_tracker0.pkg */
 1    PACKAGE BODY string_tracker
 2    IS
 3       SUBTYPE name_t IS VARCHAR2 (32767);
 4       TYPE used_aat IS TABLE OF name_t INDEX BY PLS_INTEGER;
 5       g_names_used used_aat;
 6
 7       PROCEDURE mark_as_used (variable_name_in IN name_t) IS
 8       BEGIN
 9          g_names_used (g_names_used.COUNT + 1) := variable_name_in;
10       END mark_as_used;
11
12       FUNCTION string_in_use (variable_name_in IN name_t) RETURN BOOLEAN
13       IS
14          c_count   CONSTANT PLS_INTEGER := g_names_used.COUNT;
15          l_index            PLS_INTEGER := g_names_used.FIRST;
16          l_found            BOOLEAN     := FALSE;
17       BEGIN
18          WHILE (NOT l_found AND l_index <= c_count)
19          LOOP
20             l_found := variable_name_in = g_names_used (l_index);
21             l_index := l_index + 1;
22          END LOOP;
23
24          RETURN l_found;
25       END string_in_use;
26    END string_tracker;
```

The following table gives an explanation of the interesting parts of this package body.

Line(s)	Description
3–5	Declare a collection of strings indexed by integer, to hold the list of variable names that I have already used.
7–10	Append the variable name to the end of the array, so as to mark it as "used."
12–25	Scan through the collection, looking for a match on the variable name. If found, then terminate the scan and return TRUE. Otherwise, return FALSE (string is not in use).

Now, certainly, this is not a big, complicated package body. Still, I am writing more code than is necessary, and consuming more CPU cycles than necessary. How do I simplify things and speed them up? By using a string indexed collection.

Here's my second pass at the string_tracker package:

```
      /* File on web: string_tracker1.pkg */
 1    PACKAGE BODY string_tracker
 2    IS
 3       SUBTYPE name_t IS VARCHAR2 (32767);
```

```
 4     TYPE used_aat IS TABLE OF BOOLEAN INDEX BY name_t;
 5     g_names_used used_aat;
 6
 7     PROCEDURE mark_as_used (variable_name_in IN name_t) IS
 8     BEGIN
 9        g_names_used (variable_name_in) := TRUE;
10     END mark_as_used;
11
12     FUNCTION string_in_use (variable_name_in IN name_t) RETURN BOOLEAN
13     IS
14     BEGIN
15        RETURN g_names_used.EXISTS (variable_name_in);
16     END string_in_use;
17  END string_tracker;
```

First of all, notice that my package body has shrunk from 26 lines to 17 lines—a reduction of almost 33%. And, in the process, my code has been greatly simplified. The following table explains the changes.

Line(s)	Description
3–5	This time, I declare a collection of Booleans indexed by *strings*. Actually, it doesn't really matter what kind of data the collection holds. I could create a collection of Booleans, dates, numbers, XML documents, whatever. The only thing that matters (as you will see shortly) is the index value.
7–10	Again, I mark a string as used, but in this version, the variable name serves as the *index value*, and not the value appended to the end of the collection. I assign a value of TRUE to that index value, but as I have noted, I could assign whatever value I like: NULL, TRUE, FALSE. It doesn't matter because...
12–16	To determine if a variable name has already been used, I simply call the EXISTS method for the name of the variable. If an element is defined at that index value, then the name has already been used. In other words, I never actually look at or care about the value *stored* at that index value.

Isn't that simple and elegant? I no longer have to write code to scan through the collection contents looking for a match. Instead, I zoom in directly on that index value and instantly have my answer.

Here's the lesson I took from the experience of building string_tracker: if as I write my program I find myself writing algorithms to search element by element through a collection to find a matching value, I should consider changing that collection (or creating a second collection) so that it uses string indexing to avoid the "full collection scan." The result is a program that is leaner and more efficient, as well as easier to maintain in the future.

Emulating primary keys and unique indexes

One very interesting application of string indexing is to emulate primary keys and unique indexes of a relational table in collections. Suppose that I need to do some heavy processing of employee information in my program. I need to go back and forth over the set of selected employees, searching by the employee ID number, last name, and email address.

Rather than query that data repeatedly from the database, I can cache it in a set of collections and then move much more efficiently through the data. Here is an example of the kind of code I would write:

```
DECLARE
   c_delimiter   CONSTANT CHAR (1) := '^';

   TYPE strings_t IS TABLE OF employees%ROWTYPE
                        INDEX BY employees.email%TYPE;

   TYPE ids_t IS TABLE OF employees%ROWTYPE
                     INDEX BY PLS_INTEGER;

   by_name       strings_t;
   by_email      strings_t;
   by_id         ids_t;

   ceo_name employees.last_name%TYPE
         := 'ELLISON' || c_delimiter || 'LARRY';

   PROCEDURE load_arrays
   IS
   BEGIN
      /* Load up all three arrays from rows in table. */
      FOR rec IN (SELECT *
                     FROM employees)
      LOOP
         by_name (rec.last_name || c_delimiter || rec.first_name) := rec;
         by_email (rec.email) := rec;
         by_id (rec.employee_id) := rec;
      END LOOP;
   END;
BEGIN
   load_arrays;

   /* Now I can retrieve information by name or by ID. */

   IF by_name (ceo_name).salary > by_id (7645).salary
   THEN
      make_adjustment (ceo_name);
   END IF;
END;
```

Performance of string-indexed collections

What kind of price do you pay for using string indexing instead of integer indexing? It depends entirely on how long your strings are. When you use string indexes, the database takes your string and "hashes" (transforms) it into an integer value. So the overhead is determined by the performance of the hash function, as well as the conflict resolution needed, since there is never a *guarantee* that the result of a hash is unique—just extremely *likely* to be unique.

What I have found in my testing (see the *assoc_array_perf.tst* script on the book's website) is the following:

```
Compare String and Integer Indexing, Iterations = 10000 Length = 100
   Index by PLS_INTEGER Elapsed: 4.26 seconds.
   Index by VARCHAR2 Elapsed: 4.75 seconds.
Compare String and Integer Indexing, Iterations = 10000 Length = 1000
   Index by PLS_INTEGER Elapsed: 4.24 seconds.
   Index by VARCHAR2 Elapsed: 6.4 seconds.
Compare String and Integer Indexing, Iterations = 10000 Length = 10000
   Index by PLS_INTEGER Elapsed: 4.06 seconds.
   Index by VARCHAR2 Elapsed: 24.63 seconds.
```

The conclusion: with relatively small strings (100 characters or fewer), there is no significant difference in performance between string and integer indexing. As the string index value gets longer, however, the overhead of hashing grows substantially. So be careful about what strings you use for indexes!

Other examples of string-indexed collections

As you saw in the example of retrieving employee information, it doesn't take a whole lot of code to build multiple highly efficient entry points into cached data transferred from a relational table. Still, to make it even easier for you to implement these techniques in your application, I have built a utility to generate such code for you.

The *genaa.sp* file on the book's website accepts the name of your table as an argument, and from the information stored in the data dictionary for that table (primary key and unique indexes) it generates a package to implement caching for that table. It populates a collection based on the integer primary key and another collection for each unique index defined on the table (indexed by PLS_INTEGER or VARCHAR2, depending on the type(s) of the column(s) in the index).

In addition, the file *summer_reading.pkg*, also available on the book's website, offers an example of the use of VARCHAR2-indexed associative arrays to manipulate lists of information within a PL/SQL program.

Collections of Complex Datatypes

Starting with Oracle9*i* Database Release 2, you can define collection types of arbitrarily complex structures. All of the following structures are supported:

Collections of records based on tables with %ROWTYPE
: These structures allow you to quickly and easily mimic a relational table within a PL/SQL program.

Collections of user-defined records
: The fields of the record can be scalars or complex datatypes in and of themselves. For example, you can define a collection of records where the record TYPE contains a field that is itself another collection.

Collections of object types and other complex types
: The datatype of the collection can be an object type (Oracle's version of an object-oriented class, explored in Chapter 26) previously defined with the CREATE TYPE statement. You can also easily define collections of LOBs, XML documents, etc.

Collections of collections (directly and indirectly)
: You can define multilevel collections, including collections of collections and collections of datatypes that contain, as an attribute or a field, another collection.

Let's take a look at examples of each of these variations.

Collections of records

You define a collection of records by specifying a record type (through either %ROWTYPE or a programmer-defined record type) in the TABLE OF clause of the collection definition. This technique applies only to collection TYPEs that are declared inside a PL/SQL program. Nested table and VARRAY TYPEs defined in the database cannot reference %ROWTYPE record structures.

Here is an example of a collection of records based on a custom record TYPE:

```
PACKAGE compensation_pkg
IS
   TYPE reward_rt IS RECORD (
      nm VARCHAR2(2000), sal NUMBER, comm NUMBER);

   TYPE reward_tt IS TABLE OF reward_rt INDEX BY PLS_INTEGER;

END compensation_pkg;
```

With these types defined in my package specification, I can declare collections in other programs like this:

```
DECLARE
   holiday_bonuses compensation_pkg.reward_tt;
```

Collections of records come in especially handy when you want to create in-memory collections that have the same structure (and, at least in part, data) as database tables. Why would I want to do this? Suppose that I am running a batch process on Sunday at 3:00 a.m. against tables that are modified only during the week. I need to do some intensive analysis that involves multiple passes against the tables' data. I could simply query the data repetitively from the database, but that is a relatively slow, intensive process.

Alternatively, I can copy the data from the table or tables into a collection and then move much more rapidly (and randomly) through my result set. I am, in essence, emulating bidirectional cursors in my PL/SQL code.

If you decide to copy data into collections and manipulate them within your program, you can choose between two basic approaches for implementing this logic:

- Embed all of the collection code in your main program.
- Create a separate package to encapsulate access to the data in the collection.

I generally choose the second approach for most situations. In other words, I find it useful to create separate, well-defined, and highly reusable APIs (application programming interfaces) to complex data structures and logic. Here is the package specification for my bidirectional cursor emulator:

```
/* File on web: bidir.pkg */
PACKAGE bidir
IS
   FUNCTION rowforid (id_in IN employees.employee_id%TYPE)
      RETURN employees%ROWTYPE;

   FUNCTION firstrow RETURN PLS_INTEGER;
   FUNCTION lastrow RETURN PLS_INTEGER;

   FUNCTION rowCount RETURN PLS_INTEGER;

   FUNCTION end_of_data RETURN BOOLEAN;

   PROCEDURE setrow (nth IN PLS_INTEGER);

   FUNCTION currrow RETURN employees%ROWTYPE;

   PROCEDURE nextrow;
   PROCEDURE prevrow;
END;
```

So how do you use this API? Here is an example of a program using this API to read through the result set for the employees table, first forward and then backward:

```
/* File on web: bidir.tst */
DECLARE
   l_employee   employees%ROWTYPE;
BEGIN
   LOOP
      EXIT WHEN bidir.end_of_data;
      l_employee := bidir.currrow;
      DBMS_OUTPUT.put_line (l_employee.last_name);
      bidir.nextrow;
   END LOOP;

   bidir.setrow (bidir.lastrow);
```

```
   LOOP
      EXIT WHEN bidir.end_of_data;
      l_employee := bidir.currrow;
      DBMS_OUTPUT.put_line (l_employee.last_name);
      bidir.prevrow;
   END LOOP;
END;
```

An astute reader will now be asking: when is the collection loaded up with the data? Or even better: where is the collection? There is no evidence of a collection anywhere in the code I have presented.

Let's take the second question first. The reason you don't see the collection is that I have hidden it behind my package specification. A user of the package never touches the collection and doesn't have to know anything about it. That is the whole point of the API. You just call one or another of the programs that will do all the work of traversing the collection (data set) for you.

Now, when and how is the collection loaded? This may seem a bit magical until you read about packages in Chapter 18. If you look in the package body, you will find that it has an initialization section as follows:

```
BEGIN -- Package initialization
   FOR rec IN  (SELECT * FROM employees)
   LOOP
      g_employees (rec.employee_id) := rec;
   END LOOP;
   g_currrow := firstrow;
END;
```

Note that g_currrow is defined in the package body and therefore was not listed in the preceding specification.

This means that the very first time I try to reference any element in the package specification, this code is run automatically, transferring the contents of the employees table to my g_employees collection. When does that happen in my sample program shown earlier? Inside my loop, when I call the bidir.end_of_data function to see if I am done looking through my data set!

I encourage you to examine the package implementation. The code is very basic and easy to understand, and the benefits of this approach can be dramatic.

Collections of objects and other complex types

You can use an object type, LOB, XML document, or virtually any valid PL/SQL type as the datatype of a collection TYPE statement. The syntax for defining these collections is the same, but the way you manipulate the contents of the collections can be complicated, depending on the underlying type.

For more information on Oracle object types, see Chapter 26.

Here is an example of working with a collection of objects:

```
/* File on web: object_collection.sql */
CREATE TYPE pet_t IS OBJECT (
   tag_no   INTEGER,
   name     VARCHAR2 (60),
   MEMBER FUNCTION set_tag_no (new_tag_no IN INTEGER) RETURN pet_t);
/

DECLARE
   TYPE pets_t IS TABLE OF pet_t;

   pets   pets_t :=
     pets_t (pet_t (1050, 'Sammy'), pet_t (1075, 'Mercury'));
BEGIN
   FOR indx IN pets.FIRST .. pets.LAST
   LOOP
      DBMS_OUTPUT.put_line (pets (indx).name);
   END LOOP;
END;
/
```

The output is:

```
Sammy
Mercury
```

Once I have my object type defined, I can declare a collection based on that type and then populate it with instances of those object types. You can just as easily declare collections of LOBs, XMLTypes, and so on. All the normal rules that apply to variables of those datatypes also apply to individual rows of a collection of that datatype.

Multilevel Collections

Oracle9*i* Database Release 2 introduced the ability to nest collections within collections, a feature that is also referred to as *multilevel collections*. Let's take a look at an example and then discuss how you can use this feature in your applications.

Suppose that I want to build a system to maintain information about my pets. Besides their standard information, such as breed, name, and so on, I would like to keep track of their visits to the veterinarian. So I create a vet visit object type:

```
CREATE TYPE vet_visit_t IS OBJECT (
   visit_date  DATE,
   reason      VARCHAR2 (100)
   );
/
```

Notice that objects instantiated from this type are not associated with a pet (i.e., a foreign key to a pet table or object). You will soon see why I don't need to do that. Now I create a nested table of vet visits (we are supposed to go at least once a year):

```
CREATE TYPE vet_visits_t IS TABLE OF vet_visit_t;
/
```

With these data structures defined, I now create my object type to maintain information about my pets:

```
CREATE TYPE pet_t IS OBJECT (
   tag_no   INTEGER,
   name     VARCHAR2 (60),
   petcare vet_visits_t,
   MEMBER FUNCTION set_tag_no (new_tag_no IN INTEGER) RETURN pet_t);
/
```

This object type has three attributes and one member method. Any object instantiated from this type will have associated with it a tag number, a name, and a list of visits to the vet. I can also modify the tag number for that pet by calling the set_tag_no program.

So I have now declared an object type that contains as an attribute a nested table. I don't need a separate database table to keep track of these veterinarian visits; they are a part of my object.

Now let's take advantage of the multilevel features of collections, as shown in the following example:

```
   /* File on web: multilevel_collections.sql */
 1 DECLARE
 2 TYPE bunch_of_pets_t
 3     IS
 4        TABLE OF pet_t INDEX BY PLS_INTEGER;
 5
 6     my_pets   bunch_of_pets_t;
 7  BEGIN
 8     my_pets (1) :=
 9        pet_t (
10           100
11         , 'Mercury'
12         , vet_visits_t (vet_visit_t ('01-Jan-2001', 'Clip wings')
13                       , vet_visit_t ('01-Apr-2002', 'Check cholesterol')
14                        )
15        );
16     DBMS_OUTPUT.put_line (my_pets (1).name);
17     DBMS_OUTPUT.put_line
18        (my_pets(1).petcare.LAST).reason);
```

```
19     DBMS_OUTPUT.put_line (my_pets.COUNT);
20     DBMS_OUTPUT.put_line (my_pets (1).petcare.LAST);
21  END;
```

The output from running this script is:

```
Mercury
Check cholesterol
1
2
```

The following table explains what's going on in the code.

Line(s)	Description
2–6	Declare a local associative array TYPE, in which each row contains a single pet object. I then declare a collection to keep track of my "bunch of pets."
8–15	Assign an object of type pet_t to index 1 in this associative array. As you can see, the syntax required when you're working with nested, complex objects of this sort can be quite intimidating. So let's parse the various steps. To instantiate an object of type pet_t, I must provide a tag number, a name, and a list of vet visits, which is a nested table. To provide a nested table of type vet_visits_t, I must call the associated constructor (of the same name). I can either provide a null or empty list, or initialize the nested table with some values. I do this in lines 8 and 9. Each row in the vet_visits_t collection is an object of type vet_visit_t, so again I must use the object constructor and pass in a value for each attribute (date and reason for visit).
16	Display the value of the name attribute of the pet object in row 1 of the my_pets associative array.
17-18	Display the value of the reason attribute of the vet visit object in row 2 of the nested table, which in turn resides in index 1 of the my_pets associative array. That's a mouthful, and it is a "lineful" of code.
19–21	Demonstrate use of the collection methods (in this case, COUNT and LAST) on both outer and nested collections.

In this example I have the good fortune to be working with collections that, at each level, actually have names: the my_pets associative array and the petcare nested table. This is not always the case, as is illustrated in the next example.

Unnamed multilevel collections: Emulation of multidimensional arrays

You can use nested multilevel collections to emulate multidimensional arrays within PL/SQL. Multidimensional collections are declared in stepwise fashion, adding a dimension at each step (quite different from the syntax used to declare an array in a 3GL).

I will start with a simple example and then step through the implementation of a generic three-dimensional array package. Suppose that I want to record temperatures within some three-dimensional space organized using some (X, Y, Z) coordinate system. The following block illustrates the sequential declarations necessary to accomplish this:

```
DECLARE
   SUBTYPE temperature IS NUMBER;
   SUBTYPE coordinate_axis IS PLS_INTEGER;

   TYPE temperature_x IS TABLE OF temperature INDEX BY coordinate_axis;
   TYPE temperature_xy IS TABLE OF temperature_x INDEX BY coordinate_axis;
```

```
   TYPE temperature_xyz IS TABLE OF temperature_xy INDEX BY coordinate_axis;

   temperature_3d temperature_xyz;
BEGIN
   temperature_3d (1) (2) (3) := 45;
END;
/
```

Here, the subtype and type names are used to provide clarity as to the usage of the contents of the actual collection (temperature_3d): the collection types (temperature_X, temperature_XY, temperature_XYZ) as well as the collection indexes (coordinate_axis).

Note that although my careful naming makes it clear what each of the collection types contains and is used for, I do not have corresponding clarity when it comes to referencing collection elements by subscript; in other words, in what order do I specify the dimensions? It is not obvious from my code whether the temperature 45 degrees is assigned to the point (X:1, Y:2, Z:3) or to (X:3, Y:2, Z:1).

Now let's move on to a more general treatment of a three-dimensional array structure.

The multdim package allows you to declare your own three-dimensional array, as well as set and retrieve values from individual cells. Here I create a simple package to encapsulate operations on a three-dimensional associative table storing VARCHAR2 elements indexed in all dimensions by PLS_INTEGER. The following declarations constitute some basic building blocks for the package:

```
/* Files on web: multdim.pkg, multdim.tst, multdim2.pkg */
CREATE OR REPLACE PACKAGE multdim
IS
   TYPE dim1_t IS TABLE OF VARCHAR2 (32767) INDEX BY PLS_INTEGER;
   TYPE dim2_t IS TABLE OF dim1_t INDEX BY PLS_INTEGER;
   TYPE dim3_t IS TABLE OF dim2_t INDEX BY PLS_INTEGER;

   PROCEDURE setcell (
      array_in   IN OUT   dim3_t,
      dim1_in             PLS_INTEGER,
      dim2_in             PLS_INTEGER,
      dim3_in             PLS_INTEGER,
      value_in   IN       VARCHAR2
   );

   FUNCTION getcell (
      array_in   IN   dim3_t,
      dim1_in         PLS_INTEGER,
      dim2_in         PLS_INTEGER,
      dim3_in         PLS_INTEGER
   )
      RETURN VARCHAR2;

   FUNCTION EXISTS (
      array_in   IN   dim3_t,
```

```
      dim1_in         PLS_INTEGER,
      dim2_in         PLS_INTEGER,
      dim3_in         PLS_INTEGER
   )
      RETURN BOOLEAN;
```

I have defined the three collection types progressively as before:

TYPE dim1_t
: A one-dimensional associative table of VARCHAR2 elements

TYPE dim2_t
: An associative table of Dim1_t elements

TYPE dim3_t
: An associative table of Dim2_t elements

Thus, three-dimensional space is modeled as cells in a collection of planes that are each modeled as a collection of lines. This is consistent with common understanding, which indicates a good model. Of course, my collections are sparse and finite while geometric three-dimensional space is considered to be dense and infinite, so the model has limitations. However, for my purposes, I am concerned only with a finite subset of points in three-dimensional space, and the model is adequate.

I've also equipped my three-dimensional collection type with a basic interface to get and set cell values, as well as the ability to test whether a specific cell value exists in a collection.

Exploring the multdim API

Let's look at the basic interface components. The procedure to set a cell value in a three-dimensional array given its coordinates could not be much simpler:

```
PROCEDURE setcell (
   array_in   IN OUT   dim3_t,
   dim1_in         PLS_INTEGER,
   dim2_in         PLS_INTEGER,
   dim3_in         PLS_INTEGER,
   value_in   IN   VARCHAR2
)
IS
BEGIN
   array_in(dim3_in )(dim2_in )(dim1_in) := value_in;
END;
```

Despite the simplicity of this code, there is significant added value in encapsulating the assignment statement, as it relieves me of having to remember the order of reference for the dimension indexes. It is not obvious when directly manipulating a dim3_t collection whether the third coordinate is the first index or the last. Whatever is not obvious in code will result in bugs sooner or later. The fact that all the collection indexes have

the same datatype complicates matters because mixed-up data assignments will not raise exceptions but rather just generate bad results somewhere down the line. If my testing is not thorough, these are the kinds of bugs that may make it to production code and wreak havoc on data and my reputation.

My function to return a cell value is likewise trivial but valuable:

```
FUNCTION getcell (
   array_in   IN   dim3_t,
   dim1_in         PLS_INTEGER,
   dim2_in         PLS_INTEGER,
   dim3_in         PLS_INTEGER
)
   RETURN VARCHAR2
IS
BEGIN
   RETURN array_in(dim3_in )(dim2_in )(dim1_in);
END;
```

If there is no cell in array_in corresponding to the supplied coordinates, then getcell will raise NO_DATA_FOUND. However, if any of the coordinates supplied are NULL, then the following, less friendly VALUE_ERROR exception is raised:

```
ORA-06502: PL/SQL: numeric or value error: NULL index table key value
```

In a more complete implementation, I should enhance the module to assert a precondition requiring all coordinate parameter values to be NOT NULL. At least the database's error message informs me that a null index value was responsible for the exception. It would be even better, though, if the database did not use the VALUE_ERROR exception for so many different error conditions.

With the EXISTS function, I get to some code that is a bit more interesting. EXISTS will return TRUE if the cell identified by the coordinates is contained in the collection and FALSE otherwise:

```
FUNCTION EXISTS (
   array_in   IN   dim3_t,
   dim1_in         PLS_INTEGER,
   dim2_in         PLS_INTEGER,
   dim3_in         PLS_INTEGER
 )
   RETURN BOOLEAN
IS
   l_value VARCHAR2(32767);
BEGIN
    l_value := array_in(dim3_in )(dim2_in )(dim1_in);
    RETURN TRUE;
EXCEPTION
   WHEN NO_DATA_FOUND THEN RETURN FALSE;
END;
```

This function traps the NO_DATA_FOUND exception raised when the assignment references a nonexistent cell and converts it to the appropriate Boolean. This is a very simple and direct method for obtaining my result, and illustrates a creative reliance on exception handling to handle the "conditional logic" of the function. You might think that you could and should use the EXISTS operator. You would, however, have to call EXISTS for each level of nested collections.

Here is a sample script that exercises this package:

```
/* File on web: multdim.tst */
DECLARE
   my_3d_array   multdim.dim3_t;
BEGIN
   multdim.setcell (my_3d_array, 1, 5, 800, 'def');
   multdim.setcell (my_3d_array, 1, 15, 800, 'def');
   multdim.setcell (my_3d_array, 5, 5, 800, 'def');
   multdim.setcell (my_3d_array, 5, 5, 805, 'def');

   DBMS_OUTPUT.PUT_LINE (multdim.getcell (my_3d_array, 1, 5, 800));
   /*
   Oracle 11g Release 2 allows me to call PUT_LINE with a Boolean input!
   */
   DBMS_OUTPUT.PUT_LINE (multdim.EXISTS (my_3d_array, 1, 5, 800));
   DBMS_OUTPUT.PUT_LINE (multdim.EXISTS (my_3d_array, 6000, 5, 800));
   DBMS_OUTPUT.PUT_LINE (multdim.EXISTS (my_3d_array, 6000, 5, 807));

   /*
   If you are not using Oracle 11g Release 2, then you can use this
   procedure created in bpl.sp:

   bpl (multdim.EXISTS (my_3d_array, 1, 5, 800));
   bpl (multdim.EXISTS (my_3d_array, 6000, 5, 800));
   bpl (multdim.EXISTS (my_3d_array, 6000, 5, 807));
   */

   DBMS_OUTPUT.PUT_LINE (my_3d_array.COUNT);
END;
```

The *multdim2.pkg* file on the book's website contains an enhanced version of the multdim package that implements support for "slicing" of that three-dimensional collection, in which I fix one dimension and isolate the two-dimensional plane determined by the fixed dimension. A slice from a temperature grid would give me, for example, the range of temperatures along a certain latitude or longitude.

Beyond the challenge of writing the code for slicing, an interesting question presents itself: will there be any differences between slicing out an XY plane, an XZ plane, or a YZ plane in this fashion from a symmetric cube of data? If there are significant differences, it could affect how you choose to organize your multidimensional collections.

I encourage you to explore these issues and the implementation of the *multdim2.pkg* package.

Extending string_tracker with multilevel collections

Let's look at another example of applying multilevel collections: extending the string_tracker package built in the string indexing section to support *multiple lists* of strings.

string_tracker is a handy utility, but it allows me to keep track of only one set of "used" strings at a time. What if I need to track multiple lists simultaneously? I can very easily do this with multilevel collections:

```
   /* File on web: string_tracker2.pks/pkb */
 1 PACKAGE BODY string_tracker
 2 IS
 3    SUBTYPE maxvarchar2_t IS VARCHAR2 (32767);
 4    SUBTYPE list_name_t IS maxvarchar2_t;
 5    SUBTYPE variable_name_t IS maxvarchar2_t;
 6
 7    TYPE used_aat IS TABLE OF BOOLEAN INDEX BY variable_name_t;
 8
 9    TYPE list_rt IS RECORD (
10       description      maxvarchar2_t
11     , list_of_values   used_aat
12    );
13
14    TYPE list_of_lists_aat IS TABLE OF list_rt INDEX BY list_name_t;
15
16    g_list_of_lists   list_of_lists_aat;
17
18    PROCEDURE create_list (
19       list_name_in     IN   list_name_t
20     , description_in   IN   VARCHAR2 DEFAULT NULL
21    )
22    IS
23    BEGIN
24       g_list_of_lists (list_name_in).description := description_in;
25    END create_list;
26
27    PROCEDURE mark_as_used (
28       list_name_in       IN   list_name_t
29     , variable_name_in   IN   variable_name_t
30    )
31    IS
32    BEGIN
33       g_list_of_lists (list_name_in)
34            .list_of_values (variable_name_in) := TRUE;
35    END mark_as_used;
36
37    FUNCTION string_in_use (
```

```
38        list_name_in       IN   list_name_t
39      , variable_name_in   IN   variable_name_t
40     )
41        RETURN BOOLEAN
42     IS
43     BEGIN
44        RETURN g_list_of_lists (list_name_in)
45                  .list_of_values.EXISTS (variable_name_in);
46     EXCEPTION
47        WHEN NO_DATA_FOUND
48        THEN
49           RETURN FALSE;
50     END string_in_use;
51  END string_tracker;
```

The following table explains the multilevel collection-related changes to this package.

Line(s)	Description
7	Once again, I have a collection type indexed by string to store the used strings.
9–12	Now I create a record to hold all the attributes of my list: the description and the list of used strings in that list. Notice that I do not have the list name as an attribute of my list. That may seem strange, except that the list name is the *index value* (see below).
14–16	Finally, I create a multilevel collection type: a list of lists, in which each element in this top-level collection contains a record, which in turn contains the collection of used strings.

33–34	Now the mark_as_used procedure uses both the list name and the variable name as the index values into their respective collections:
	`g_list_of_lists (list_name_in)`
	`.list_of_values(variable_name_in) := TRUE;`
	Notice that if I mark a variable name as used in a *new list*, the database creates a *new* element in the g_list_of_lists collection for that list. If I mark a variable name as used in a previously created list, it simply adds another element to the nested collection.

44–45	Now to check to see if a string is used, I look to see if the variable name is defined as an element *within* an element of the list of lists collection:
	`RETURN g_list_of_lists (list_name_in)`
	`.list_of_values.EXISTS (variable_name_in);`

Finally, notice that in this third implementation of string_tracker I was very careful to use named subtypes in each of my formal parameter declarations, and especially in the INDEX BY clauses of the collection type declarations. Using subtypes instead of hard-coded VARCHAR2 declarations makes the code much more self-documenting. If you do not do this, one day you will find yourself scratching your head and asking, "What am I using for the index of that collection?"

How deeply can I nest collections?

As I played around with two- and three-dimensional arrays, I found myself wondering how deeply I could nest these multilevel collections. So I decided to find out. I built a

small code generator that allows me to pass in the number of levels of nesting. It then constructs a procedure that declares *N* collection TYPEs, each one being a TABLE OF the previous table TYPE. Finally, it assigns a value to the string that is all the way at the heart of the nested collections.

I was able to create a collection of at least 250 nested collections before my computer ran into a memory error! I find it hard to believe that any PL/SQL developer will even come close to that level of complexity. If you would like to run this same experiment on your own system, check out the *gen_multcoll.sp* file available on the book's website.

Working with Collections in SQL

I've been working with Oracle's SQL for more than 22 years and PL/SQL for more than 18, but my brain has rarely turned as many cartwheels over SQL's semantics as it did when I first contemplated the *collection pseudofunctions* introduced in Oracle8 Database. These pseudofunctions exist to coerce database tables into acting like collections, and vice versa. Because there are some manipulations that work best when data is in one form rather than the other, these functions give application programmers access to a rich and interesting set of structures and operations.

The collection pseudofunctions are not available in PL/SQL proper, only in SQL. You can, however, employ these operators in SQL statements that appear in your PL/SQL code, and it is extremely useful to understand how and when to do so. You'll see examples in the following sections.

The four collection pseudofunctions are as follows:

CAST
: Maps a collection of one type to a collection of another type. This can encompass mapping a VARRAY to a nested table.

COLLECT
: Aggregates data into a collection in SQL. First introduced in Oracle Database 10*g*, this function was enhanced in 11.2 to support ordering and deduplication of data.

MULTISET
: Maps a database table to a collection. With MULTISET and CAST, you can actually retrieve a row from a database table as a collection-typed column.

TABLE
: Maps a collection to a database table. This is the inverse of MULTISET: it returns a single value that contains the mapped table.

Oracle introduced these pseudofunctions to manipulate collections that live in the database. They are important to your PL/SQL programs for several reasons, not the least

of which is that they provide an incredibly efficient way to move data between the database and the application.

Yes, these pseudofunctions can be puzzling. But if you're the kind of person who gets truly excited by arcane code, these SQL extensions will make you jumping-up-and-down silly.

The CAST pseudofunction

The CAST operator can be used in a SQL statement to convert from one built-in datatype or collection type to another built-in datatype or collection type. In other words, within SQL you can use CAST in place of TO_CHAR to convert from number to string.

Another very handy use of CAST is to convert between types of collections. Here is an example of casting a named collection. Suppose that I have created the color_ models table based on a VARRAY type as follows:

```
TYPE color_nt AS TABLE OF VARCHAR2(30)

TYPE color_vat AS VARRAY(16) OF VARCHAR2(30)

TABLE color_models (
   model_type VARCHAR2(12),
   colors color_vat);
```

I can CAST the VARRAY colors column as a nested table and apply the pseudofunction TABLE (explained shortly) to the result. An example is shown here. COLUMN_VALUE is the name that the database gives to the column in the resulting one-column virtual table. You can change it to whatever you want with a column alias:

```
SELECT COLUMN_VALUE my_colors
  FROM  TABLE (SELECT CAST(colors AS color_nt)
                 FROM color_models
                WHERE model_type = 'RGB')
```

CAST performs an on-the-fly conversion of the color_vat collection type to the color_nt collection type. CAST cannot serve as the target of an INSERT, UPDATE, or DELETE statement.

Starting with Oracle Database 10*g*, you do not need to explicitly CAST a collection inside the TABLE operator. Instead, the database automatically determines the correct type.

It is also possible to cast a "bunch of table rows"—such as the result of a subquery—as a particular collection type. Doing so requires the MULTISET function, covered in the next section.

The COLLECT pseudofunction

The COLLECT aggregate function, introduced in Oracle 10*g*, enables aggregation of data from within a SQL statement into a collection. In 11.2, Oracle enhanced this function so that you can specify an order for the aggregated results and also remove duplicates during the aggregation process.

Here is an example of calling COLLECT and ordering the results (I include only the first three rows):

```
SQL> CREATE OR REPLACE TYPE strings_nt IS TABLE OF VARCHAR2 (100)
  2  /

SQL>   SELECT department_id,
  2           CAST (COLLECT (last_name ORDER BY hire_date) AS strings_nt)
  3              AS by_hire_date
  4      FROM employees
  5  GROUP BY department_id
  6  /

DEPARTMENT_ID BY_HIRE_DATE
------------- ------------------------------------------------------------
           10  STRINGS_NT('Whalen')
           20  STRINGS_NT('Hartstein', 'Fay')
           30  STRINGS_NT('Raphaely', 'Khoo', 'Tobias', 'Baida', 'Colmenares')
```

And here is an example that demonstrates how to remove duplicates in the aggregation process:

```
SQL>   SELECT department_id,
  2           CAST (COLLECT (DISTINCT job_id) AS strings_nt)
  3              AS unique_jobs
  4      FROM employees
  5  GROUP BY department_id
  6  /

DEPARTMENT_ID UNIQUE_JOBS
------------- ------------------------------------
           10 STRINGS_NT('AD_ASST')
           20 STRINGS_NT('MK_MAN', 'MK_REP')
           30 STRINGS_NT('PU_CLERK', 'PU_MAN')
```

You will find an excellent article on COLLECT at oracle-developer.net.

The MULTISET pseudofunction

The MULTISET function exists only for use within CASTs. MULTISET allows you to retrieve a set of data and convert it on the fly to a collection type. (Note that the SQL MULTISET function is distinct from the PL/SQL MULTISET operators for nested tables, discussed in "Nested Table Multiset Operations" on page 406.)

The simplest form of MULTISET is this:

```
SELECT CAST (MULTISET (SELECT field FROM table) AS collection-type)
   FROM DUAL;
```

You can also use MULTISET in a correlated subquery in the select list:

```
SELECT outerfield,
  CAST(MULTISET(SELECT field FROM whateverTable
                WHERE correlationCriteria)
    AS collectionTypeName)
  FROM outerTable
```

This technique is useful for making joins look as if they include a collection. For example, suppose I have a detail table that lists, for each bird in my table, the countries where that species lives:

```
CREATE TABLE birds (
   genus VARCHAR2(128),
   species VARCHAR2(128),
   colors color_tab_t,
   PRIMARY KEY (genus, species)
);

CREATE TABLE bird_habitats (
   genus VARCHAR2(128),
   species VARCHAR2(128),
   country VARCHAR2(60),
   FOREIGN KEY (genus, species) REFERENCES birds (genus, species)
);

CREATE TYPE country_tab_t AS TABLE OF VARCHAR2(60);
```

I should then be able to smush the master and detail tables together in a single SELECT that converts the detail records into a collection type. This feature has enormous significance for client/server programs because the number of roundtrips can be cut down without incurring the overhead of duplicating the master records with each and every detail record:

```
DECLARE
   CURSOR bird_curs IS
      SELECT b.genus, b.species,
         CAST(MULTISET(SELECT bh.country FROM bird_habitats bh
                       WHERE bh.genus = b.genus
                         AND bh.species = b.species)
           AS country_tab_t)
        FROM birds b;
   bird_row bird_curs%ROWTYPE;
BEGIN
   OPEN bird_curs;
   FETCH bird_curs into bird_row;
   CLOSE bird_curs;
END;
```

Like the CAST pseudofunction, MULTISET cannot serve as the target of an INSERT, UPDATE, or DELETE statement.

The TABLE pseudofunction

The TABLE operator casts or converts a collection into something you can SELECT from. It sounds complicated, but this section presents an example that's not too hard to follow.

Looking at it another way, let's say that you have a database table with a column of a collection type. How can you figure out which rows in the table contain a collection that meets certain criteria? That is, how can you select from the database table, putting a WHERE clause on the collection's contents? Wouldn't it be nice if you could just say:

```
SELECT *
  FROM table_name
 WHERE collection_column
       HAS CONTENTS 'whatever';   -- INVALID! Imaginary syntax!
```

Logically, that's exactly what you can do with the TABLE function. Going back to my color_models database table, how could I get a listing of all color models that contain the color RED? Here's the real way to do it:

```
SELECT *
  FROM color_models c
 WHERE 'RED' IN
       (SELECT * FROM TABLE (c.colors));
```

which, in SQL*Plus, returns:

```
MODEL_TYPE   COLORS
------------ -----------------------------------
RGB          COLOR_TAB_T('RED', 'GREEN', 'BLUE')
```

The query means "go through the color_models table and return all rows whose list of colors contains at least one RED element." Had there been more rows with a RED element in their colors column, these rows too would have appeared in my SQL*Plus result set.

As shown previously, TABLE accepts a collection as its only argument, which can be an alias-qualified collection column or even a PL/SQL variable:

```
TABLE(scope_name.collection_name)
```

TABLE returns the contents of this collection coerced into a virtual database table. You can then SELECT from it, and from that point on it can be used like any other SELECT: you can join it to other data sets, perform set operations (UNION, INTERSECT, MINUS), etc. In the previous example, it is used in a subquery. Here's an example of using TABLE with a local PL/SQL variable:

```
/* File on web: nested_table_example.sql */

/* Create a schema-level type. */

CREATE OR REPLACE TYPE list_of_names_t
   IS TABLE OF VARCHAR2 (100);
/

/* Populate the collection, then use a cursor FOR loop
   to select all elements and display them. */

DECLARE
   happyfamily   list_of_names_t := list_of_names_t ();
BEGIN
   happyfamily.EXTEND (6);
   happyfamily (1) := 'Eli';
   happyfamily (2) := 'Steven';
   happyfamily (3) := 'Chris';
   happyfamily (4) := 'Veva';
   happyfamily (5) := 'Lauren';
   happyfamily (6) := 'Loey';

   FOR rec IN (  SELECT COLUMN_VALUE family_name
                   FROM TABLE (happyfamily)
               ORDER BY family_name)
   LOOP
      DBMS_OUTPUT.put_line (rec.family_name);
   END LOOP;
END;
/
```

Prior to Oracle Database 12*c*, you could only use the TABLE pseudofunction with nested tables and VARRAYs, and only if their types were defined at the schema level with CREATE OR REPLACE TYPE (there was one exception to this rule: pipelined table functions could work with types defined in package specifications). Starting with Oracle Database 12*c*, however, you can also use the TABLE pseudofunction with nested tables, VARRAYs, and integer-indexed associative arrays, as long as their types are defined in a package specification:

```
/* File on web: 12c_table_pf_with_aa.sql */

/* Create a package-based type. */

CREATE OR REPLACE PACKAGE aa_pkg
IS
   TYPE strings_t IS TABLE OF VARCHAR2 (100)
      INDEX BY PLS_INTEGER;
END;
/

/* Populate the collection, then use a cursor FOR loop
   to select all elements and display them. */
```

```
DECLARE
   happyfamily   aa_pkg.strings_t;
BEGIN
   happyfamily (1) := 'Me';
   happyfamily (2) := 'You';

   FOR rec IN (  SELECT COLUMN_VALUE family_name
                   FROM TABLE (happyfamily)
               ORDER BY family_name)
   LOOP
      DBMS_OUTPUT.put_line (rec.family_name);
   END LOOP;
END;
/
```

You cannot, however, use TABLE with a locally declared collection type, as you can see here (note also that the error message has not yet been changed to reflect the wider use of TABLE):

```
/* File on web: 12c_table_pf_with_aa.sql */
DECLARE
   TYPE strings_t IS TABLE OF VARCHAR2 (100)
      INDEX BY PLS_INTEGER;

   happyfamily   strings_t;
BEGIN
   happyfamily (1) := 'Me';
   happyfamily (2) := 'You';


   FOR rec IN (  SELECT COLUMN_VALUE family_name
                   FROM TABLE (happyfamily)
               ORDER BY family_name)
   LOOP
      DBMS_OUTPUT.put_line (rec.family_name);
   END LOOP;
END;
/

ERROR at line 12:
ORA-06550: line 12, column 32:
PLS-00382: expression is of wrong type
ORA-06550: line 12, column 25:
PL/SQL: ORA-22905: cannot access rows from a non-nested table item
```

To repeat an earlier admonition, none of the collection pseudofunctions is available from within PL/SQL, but PL/SQL programmers will certainly want to know how to use these gizmos in their SQL statements!

You will also find the pseudofunctions, particularly TABLE, very handy when you are taking advantage of the *table function capability* introduced in Oracle9*i* Database. A

table function is a function that returns a collection, and it can be used in the FROM clause of a query. This functionality is explored in Chapter 17.

Sorting contents of collections

One of the wonderful aspects of pseudofunctions is that they allow you to apply SQL operations against the contents of PL/SQL data structures (nested tables and VARRAYs, at least). You can, for example, use ORDER BY to select information from the nested table in the order you desire. Here, I populate a database table with some of my favorite authors:

```
CREATE TYPE authors_t AS TABLE OF VARCHAR2 (100);
CREATE TABLE favorite_authors (name varchar2(200));

BEGIN
   INSERT INTO favorite_authors VALUES ('Robert Harris');
   INSERT INTO favorite_authors VALUES ('Tom Segev');
   INSERT INTO favorite_authors VALUES ('Toni Morrison');
END;
```

Now I would like to blend this information with data from my PL/SQL program:

```
DECLARE
   scifi_favorites   authors_t
      := authors_t ('Sheri S. Tepper', 'Orson Scott Card', 'Gene Wolfe');
BEGIN
   DBMS_OUTPUT.put_line ('I recommend that you read books by:');

   FOR rec IN  (SELECT COLUMN_VALUE favs
                  FROM TABLE (CAST (scifi_favorites AS authors_t))
                UNION
                SELECT NAME
                  FROM favorite_authors
                ORDER BY favs)
   LOOP
      DBMS_OUTPUT.put_line (rec.favs);
   END LOOP;
END;
```

Notice that I can use UNION to combine data from my database table and collection. I can also apply this technique only to PL/SQL data to sort the contents being retrieved:

```
DECLARE
   scifi_favorites   authors_t
      := authors_t ('Sheri S. Tepper', 'Orson Scott Card', 'Gene Wolfe');
BEGIN
   DBMS_OUTPUT.put_line ('I recommend that you read books by:');

   FOR rec IN  (SELECT COLUMN_VALUE Favs
                  FROM TABLE (CAST (scifi_favorites AS authors_t))
                 ORDER BY COLUMN_VALUE)
   LOOP
```

```
      DBMS_OUTPUT.put_line (rec.favs);
   END LOOP;
END;
```

COLUMN_VALUE in the preceding query is the system-defined name of the column created with the TABLE operator, when only a single column is being fetched. If more than one column is being queried, the names of the associated object type attributes will be used in place of COLUMN_VALUE.

Nested Table Multiset Operations

The essential advance made in collections starting with Oracle Database 10*g* is that the database treats nested tables more like the multisets that they actually are. The database provides high-level set operations that can be applied to nested tables (and, for the time being, *only* to nested tables). This table offers a brief summary of these set-level capabilities.

Operation	Return value	Description
=	BOOLEAN	Compares two nested tables, and returns TRUE if they have the same named type and cardinality and if the elements are equal.
<> or !=	BOOLEAN	Compares two nested tables, and returns FALSE if they differ in named type, cardinality, or equality of elements.
[NOT] IN ()	BOOLEAN	Returns TRUE [FALSE] if the nested table to the left of IN exists in the list of nested tables in the parentheses.
x MULTISET EXCEPT [DISTINCT] *y*	NESTED TABLE	Performs a MINUS set operation on nested tables *x* and *y*, returning a nested table whose elements are in *x*, but not in *y*. *x*, *y*, and the returned nested table must all be of the same type. The DISTINCT keyword instructs Oracle to eliminate duplicates in the resulting nested table.
x MULTISET INTERSECT [DISTINCT] *y*	NESTED TABLE	Performs an INTERSECT set operation on nested tables *x* and *y*, returning a nested table whose elements are in both *x* and *y*. *x*, *y*, and the returned nested table must all be of the same type. The DISTINCT keyword forces the elimination of duplicates from the returned nested table, including duplicates of NULL, if they exist.
x MULTISET UNION [DISTINCT] *y*	NESTED TABLE	Performs a UNION set operation on nested tables *x* and *y*, returning a nested table whose elements include all those in *x* as well as those in *y*. *x*, *y*, and the returned nested table must all be of the same type. The DISTINCT keyword forces the elimination of duplicates from the returned nested table, including duplicates of NULL, if they exist.
SET(*x*)	NESTED TABLE	Returns nested table *x* without duplicate elements.
x IS [NOT] A SET	BOOLEAN	Returns TRUE [FALSE] if the nested table *x* is composed of unique elements.
x IS [NOT] EMPTY	BOOLEAN	Returns TRUE [FALSE] if the nested table *x* is empty.

Operation	Return value	Description
e [NOT] MEMBER [OF] *x*	BOOLEAN	Returns TRUE [FALSE] if the expression *e* is a member of the nested table *x*. Warning: MEMBER inside SQL statements is very inefficient, while in PL/SQL performance is much better.
y [NOT] SUBMULTISET [OF] *x*	BOOLEAN	Returns TRUE [FALSE] if for every element of *y* there is a matching element in *x*.

In the following sections, I will take a closer look at many of these features. As I do so, I'll make frequent references to this nested table type:

```
/* File on web: 10g_strings_nt.sql */
TYPE strings_nt IS TABLE OF VARCHAR2(100);
```

I'll also make repeated use of the following package:

```
/* File on web: 10g_authors.pkg */
CREATE OR REPLACE PACKAGE authors_pkg
IS
   steven_authors    strings_nt
      := strings_nt ('ROBIN HOBB'
                   , 'ROBERT HARRIS'
                   , 'DAVID BRIN'
                   , 'SHERI S. TEPPER'
                   , 'CHRISTOPHER ALEXANDER'
                    );
   veva_authors   strings_nt
      := strings_nt ('ROBIN HOBB'
                   , 'SHERI S. TEPPER'
                   , 'ANNE MCCAFFREY'
                   );

   eli_authors   strings_nt
      := strings_nt ( 'SHERI S. TEPPER'
                    , 'DAVID BRIN'
                    );

   PROCEDURE show_authors (
      title_in   IN   VARCHAR2
    , authors_in    IN   strings_nt
   );
END;
/

CREATE OR REPLACE PACKAGE BODY authors_pkg
IS
   PROCEDURE show_authors (
      title_in   IN   VARCHAR2
    , authors_in    IN   strings_nt
   )
   IS
```

```
   BEGIN
      DBMS_OUTPUT.put_line (title_in);

      FOR indx IN authors_in.FIRST .. authors_in.LAST
      LOOP
         DBMS_OUTPUT.put_line (indx || ' = ' || authors_in (indx));
      END LOOP;

      DBMS_OUTPUT.put_line ('_');
   END show_authors;
END;
/
```

Testing Equality and Membership of Nested Tables

Prior to Oracle Database 10*g*, the only way to tell if two collections were identical (i.e., had the same contents) was to compare the values of each row for equality (and if the collection contained records, you would have to compare each field of each record); see the example in *10g_coll_compare_old.sql* for an example of this code. From Oracle Database 10*g* onward, with nested tables, you only need to use the standard = and != operators, as shown in the following example:

```
/* File on web: 10g_coll_compare.sql */
DECLARE
   TYPE clientele IS TABLE OF VARCHAR2 (64);

   group1   clientele := clientele ('Customer 1', 'Customer 2');
   group2   clientele := clientele ('Customer 1', 'Customer 3');
   group3   clientele := clientele ('Customer 3', 'Customer 1');
BEGIN
   IF group1 = group2
   THEN
      DBMS_OUTPUT.put_line ('Group 1 = Group 2');
   ELSE
      DBMS_OUTPUT.put_line ('Group 1 != Group 2');
   END IF;

   IF group2 != group3
   THEN
      DBMS_OUTPUT.put_line ('Group 2 != Group 3');
   ELSE
      DBMS_OUTPUT.put_line ('Group 2 = Group 3');
   END IF;
END;
```

Note that the equality check implemented for nested tables treats NULLs consistently with other operators. It considers NULL to be "unknowable." Thus, one NULL is never equal to another NULL. As a consequence, if both of the nested tables you are comparing contain a NULL value at the same row, they will *not* be considered equal.

Checking for Membership of an Element in a Nested Table

In a variation on that theme, you can use the MEMBER operator to determine if a particular element is in a nested table. Use SUBMULTISET to determine if an *entire* nested table is contained in another nested table. Here is an example:

```
/* File on web: 10g_submultiset.sql */
BEGIN
   bpl (authors_pkg.steven_authors
            SUBMULTISET OF authors_pkg.eli_authors
      , 'Father follows son?');
   bpl (authors_pkg.eli_authors
            SUBMULTISET OF authors_pkg.steven_authors
      , 'Son follows father?');

   bpl (authors_pkg.steven_authors
            NOT SUBMULTISET OF authors_pkg.eli_authors
      , 'Father doesn''t follow son?');
   bpl (authors_pkg.eli_authors
            NOT SUBMULTISET OF authors_pkg.steven_authors
      , 'Son doesn''t follow father?');
END;
/
```

Here are the results of running this code:

```
SQL> @10g_submultiset
Father follows son? - FALSE
Son follows father? - TRUE
Father doesn't follow son? - TRUE
Son doesn't follow father? - FALSE
```

Performing High-Level Set Operations

Set operations like UNION, INTERSECT, and MINUS are extremely powerful and helpful, precisely because they are such simple, high-level concepts. You can write a very small amount of code to achieve great effects. Consider the following code, which shows a variety of set operators at work:

```
   /* File on web: 10g_union.sql */
 1 DECLARE
 2    our_authors strings_nt := strings_nt();
 3 BEGIN
 4     our_authors := authors_pkg.steven_authors
 5                    MULTISET UNION authors_pkg.veva_authors;
 6
 7     authors_pkg.show_authors ('MINE then VEVA', our_authors);
 8
 9     our_authors := authors_pkg.veva_authors
10                    MULTISET UNION authors_pkg.steven_authors;
11
12     authors_pkg.show_authors ('VEVA then MINE', our_authors);
```

```
13
14      our_authors := authors_pkg.steven_authors
15                     MULTISET UNION DISTINCT authors_pkg.veva_authors;
16
17      authors_pkg.show_authors ('MINE then VEVA with DISTINCT', our_authors);
18
19      our_authors := authors_pkg.steven_authors
20                     MULTISET INTERSECT authors_pkg.veva_authors;
21
22      authors_pkg.show_authors ('IN COMMON', our_authors);
23
24      our_authors := authors_pkg.veva_authors
25                     MULTISET EXCEPT authors_pkg.steven_authors;
26
27      authors_pkg.show_authors (q'[ONLY VEVA'S]', our_authors);
28  END;
```

Here is the output from running this script:

```
SQL>  @10g_union
MINE then VEVA
1 = ROBIN HOBB
2 = ROBERT HARRIS
3 = DAVID BRIN
4 = SHERI S. TEPPER
5 = CHRISTOPHER ALEXANDER
6 = ROBIN HOBB
7 = SHERI S. TEPPER
8 = ANNE MCCAFFREY

_
VEVA then MINE
1 = ROBIN HOBB
2 = SHERI S. TEPPER
3 = ANNE MCCAFFREY
4 = ROBIN HOBB
5 = ROBERT HARRIS
6 = DAVID BRIN
7 = SHERI S. TEPPER
8 = CHRISTOPHER ALEXANDER

_
MINE then VEVA with DISTINCT
1 = ROBIN HOBB
2 = ROBERT HARRIS
3 = DAVID BRIN
4 = SHERI S. TEPPER
5 = CHRISTOPHER ALEXANDER
6 = ANNE MCCAFFREY

_
IN COMMON
1 = ROBIN HOBB
2 = SHERI S. TEPPER

_
```

```
ONLY VEVA'S
1 = ANNE MCCAFFREY
```

Note that MULTISET UNION does not act precisely the same as the SQL UNION. It does not reorder the data, and it does not remove duplicate values. Duplicates are perfectly acceptable and, indeed, are significant in a multiset. If, however, you want to remove duplicates, use MULTISET UNION DISTINCT.

Handling Duplicates in a Nested Table

So, a nested table can have duplicates (the same value stored more than once), and those duplicates will persist even beyond a MULTISET UNION operation. Sometimes this is what you want; sometimes you would much rather have a distinct set of values with which to work. Oracle provides the following operators:

SET operator
: Helps you transform a nondistinct set of elements in a nested table into a distinct set. You can think of it as a "SELECT DISTINCT" for nested tables.

IS A SET and IS [NOT] A SET operators
: Helps you answers questions like "Does this nested table contain any duplicate entries?"

The following script exercises these features of Oracle Database 10*g* and later:

```
/* Files on web: 10g_set.sql, bpl2.sp */
BEGIN
   -- Add a duplicate author to Steven's list
   authors_pkg.steven_authors.EXTEND;
   authors_pkg.steven_authors(
   authors_pkg.steven_authors.LAST) := 'ROBERT HARRIS';

   distinct_authors :=
      SET (authors_pkg.steven_authors);

   authors_pkg.show_authors (
      'FULL SET', authors_pkg.steven_authors);

   bpl (authors_pkg.steven_authors IS A SET, 'My authors distinct?');
   bpl (authors_pkg.steven_authors IS NOT A SET, 'My authors NOT distinct?');
   DBMS_OUTPUT.PUT_LINE ('');

   authors_pkg.show_authors (
      'DISTINCT SET', distinct_authors);

   bpl (distinct_authors IS A SET, 'SET of authors distinct?');
   bpl (distinct_authors IS NOT A SET, 'SET of authors NOT distinct?');
   DBMS_OUTPUT.PUT_LINE ('');

END;
/
```

And here are the results of this script:

```
SQL> @10g_set
FULL SET
1 = ROBIN HOBB
2 = ROBERT HARRIS
3 = DAVID BRIN
4 = SHERI S. TEPPER
5 = CHRISTOPHER ALEXANDER
6 = ROBERT HARRIS

_
My authors distinct? - FALSE
My authors NOT distinct? - TRUE

DISTINCT SET
1 = ROBIN HOBB
2 = ROBERT HARRIS
3 = DAVID BRIN
4 = SHERI S. TEPPER
5 = CHRISTOPHER ALEXANDER

_
SET of authors distinct? - TRUE
SET of authors NOT distinct? - FALSE
```

Maintaining Schema-Level Collections

Here are some not-so-obvious bits of information that will assist you in using nested tables and VARRAYs. This kind of housekeeping is not necessary or relevant when you're working with associative arrays.

Necessary Privileges

When they live in the database, collection datatypes can be shared by more than one database user (schema). As you can imagine, privileges are involved. Fortunately, it's not complicated; only one Oracle privilege—EXECUTE—applies to collection types.

If you are Scott, and you want to grant Joe permission to use color_tab_t in his programs, all you need to do is grant the EXECUTE privilege to him:

```
GRANT EXECUTE on color_tab_t TO JOE;
```

Joe can then refer to the type using *schema.type* notation. For example:

```
CREATE TABLE my_stuff_to_paint (
   which_stuff VARCHAR2(512),
   paint_mixture SCOTT.color_tab_t
)
NESTED TABLE paint_mixture STORE AS paint_mixture_st;
```

EXECUTE privileges are also required by users who need to run PL/SQL anonymous blocks that use the object type. That's one of several reasons that named PL/SQL

modules—packages, procedures, functions—are generally preferred. Granting EXECUTE privileges on the module confers the grantor's privileges to the grantee while that user is executing the module.

For tables that include collection columns, the traditional SELECT, INSERT, UPDATE, and DELETE privileges still have meaning, as long as there is no requirement to build a collection for any columns. However, if a user is going to INSERT or UPDATE the contents of a collection column, that user must have the EXECUTE privilege on the type because that is the only way to use the default constructor.

Collections and the Data Dictionary

The Oracle database offers several data dictionary views that provide information about your nested table and VARRAY collection types (see Table 12-4). The shorthand dictionary term for user-defined types is simply TYPE.

Table 12-4. Data dictionary entries for collection types

To answer the question...	...use this view	...as in

What collection types have I created?	ALL_TYPES	`SELECT type_name`
		`  FROM all_types`
		`WHERE owner = USER`
		`  AND typecode ='COLLECTION';`

What was the original type definition of collection Foo_t?	ALL_SOURCE	`SELECT text`
		`  FROM all_source`
		`WHERE owner = USER`
		`  AND name = 'FOO_T'`
		`  AND type = 'TYPE'`
		`ORDER BY line;`

What columns implement Foo_t?	ALL_TAB_ COLUMNS	`SELECT table_name,column_name`
		`  FROM all_tab_columns`
		`WHERE owner = USER`
		`  AND data_type = 'FOO_T';`

What database objects are dependent on Foo_t?	ALL_DEPENDENCIES	`SELECT name, type`
		`  FROM all_dependencies`
		`WHERE owner = USER`
		`  AND referenced_name='FOO_T';`