[Skip Headers](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm#BEGIN)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **PL/SQL User's Guide and Reference Release 2 (9.2)** Part Number A96624-01 | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | [[Go To Documentation Library](http://docs.oracle.com/cd/B10500_01/index.htm) Home](http://docs.oracle.com/cd/B10500_01/index.htm) | [[Go To Product List](http://docs.oracle.com/cd/B10500_01/nav/docindex.htm) Book List](http://docs.oracle.com/cd/B10500_01/nav/docindex.htm) | [[Go To Table Of Contents](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/toc.htm) Contents](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/toc.htm) | [[Go To Index](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/index.htm) Index](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/index.htm) | [[http://docs.oracle.com/cd/B10500_01/dcommon/gifs/mix.gif](http://docs.oracle.com/cd/B10500_01/mix.920/a96625/toc.htm) Master Index](http://docs.oracle.com/cd/B10500_01/mix.920/a96625/toc.htm) | [[http://docs.oracle.com/cd/B10500_01/dcommon/gifs/feedback.gif](http://docs.oracle.com/cd/B10500_01/dcommon/html/feedback.htm) Feedback](http://docs.oracle.com/cd/B10500_01/dcommon/html/feedback.htm) | |

[Go to previous page](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/04_struc.htm)[Go to next page](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/06_ora.htm)

**5****PL/SQL Collections and Records**

*Knowledge is that area of ignorance that we arrange and classify. --*Ambrose Bierce

Many programming techniques use collection types such as arrays, bags, lists, nested tables, sets, and trees. To support these techniques in database applications, PL/SQL provides the datatypes TABLE and VARRAY, which allow you to declare index-by tables, nested tables and variable-size arrays. In this chapter, you learn how those types let you reference and manipulate collections of data as whole objects. You also learn how the datatype RECORD lets you treat related but dissimilar data as a logical unit.

This chapter discusses the following topics:

["What Is a Collection?"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm" \l "20425)

["Choosing Which PL/SQL Collection Types to Use"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm" \l "34607)

["Defining Collection Types"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm" \l "19661)

["Declaring PL/SQL Collection Variables"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm" \l "35385)

["Initializing and Referencing Collections"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm" \l "19834)

["Assigning Collections"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm" \l "20986)

["Using PL/SQL Collections with SQL Statements"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm" \l "20023)

["Using Collection Methods"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm" \l "14165)

["Avoiding Collection Exceptions"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm" \l "14332)

["Reducing Loop Overhead for Collections with Bulk Binds"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm" \l "23723)

["What Is a Record?"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm" \l "7658)

["Defining and Declaring Records"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm" \l "7543)

["Initializing Records"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm" \l "7581)

["Assigning Records"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm" \l "7600)

["Manipulating Records"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm" \l "28087)

**What Is a Collection?**

A *collection* is an ordered group of elements, all of the same type. It is a general concept that encompasses lists, arrays, and other familiar datatypes. Each element has a unique subscript that determines its position in the collection.

PL/SQL offers these collection types:

* **Index-by tables**, also known as **associative arrays**, let you look up elements using arbitrary numbers and strings for subscript values. (They are similar to *hash tables* in other programming languages.)
* **Nested tables** hold an arbitrary number of elements. They use sequential numbers as subscripts. You can define equivalent SQL types, allowing nested tables to be stored in database tables and manipulated through SQL.
* **Varrays** (short for variable-size arrays) hold a fixed number of elements (although you can change the number of elements at runtime). They use sequential numbers as subscripts. You can define equivalent SQL types, allowing varrays to be stored in database tables. They can be stored and retrieved through SQL, but with less flexibility than nested tables.

Although collections can have only one dimension, you can model multi-dimensional arrays by creating collections whose elements are also collections.

To use collections in an application, you define one or more PL/SQL types, then define variables of those types. You can define collection types in a procedure, function, or package. You can pass collection variables as parameters, to move data between client-side applications and stored subprograms.

To look up data that is more complex than single values, you can store PL/SQL records or SQL object types in collections. Nested tables and varrays can also be attributes of object types.

**Understanding Nested Tables**

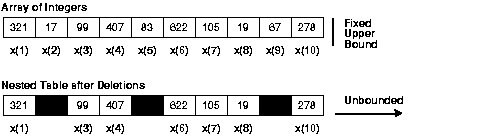
Within the database, nested tables can be considered one-column database tables. Oracle stores the rows of a nested table in no particular order. But, when you retrieve the nested table into a PL/SQL variable, the rows are given consecutive subscripts starting at 1. That gives you array-like access to individual rows.

PL/SQL nested tables are like one-dimensional arrays. You can model multi-dimensional arrays by creating nested tables whose elements are also nested tables.

Nested tables differ from arrays in two important ways:

1. Arrays have a fixed upper bound, but nested tables are unbounded (see [Figure 5-1](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm#24240)). So, the size of a nested table can increase dynamically.

***Figure 5-1 Array versus Nested Table***

 [Text description of the illustration pls81016\_array\_versus\_nested\_table.gif](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/img_text/pls81016_array_versus_nested_table.htm)

1. Arrays must be *dense* (have consecutive subscripts). So, you cannot delete individual elements from an array. Initially, nested tables are dense, but they can become *sparse* (have nonconsecutive subscripts). So, you can delete elements from a nested table using the built-in procedure DELETE. That might leave gaps in the index, but the built-in function NEXT lets you iterate over any series of subscripts.

**Understanding Varrays**

Items of type VARRAY are called *varrays*. They allow you to associate a single identifier with an entire collection. This association lets you manipulate the collection as a whole and reference individual elements easily. To reference an element, you use standard subscripting syntax (see [Figure 5-2](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm#19471)). For example, Grade(3) references the third element in varray Grades.

***Figure 5-2 Varray of Size 10***

Text description of pls81017_varray_of_size_10.gif follows [Text description of the illustration pls81017\_varray\_of\_size\_10.gif](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/img_text/pls81017_varray_of_size_10.htm)

A varray has a maximum size, which you must specify in its type definition. Its index has a fixed lower bound of 1 and an extensible upper bound. For example, the current upper bound for varray Grades is 7, but you can extend it to 8, 9, 10, and so on. Thus, a varray can contain a varying number of elements, from zero (when empty) to the maximum specified in its type definition.

**Understanding Associative Arrays (Index-By Tables)**

Associative arrays are sets of key-value pairs, where each key is unique and is used to locate a corresponding value in the array. The key can be an integer or a string.

Assigning a value using a key for the first time adds that key to the associative array. Subsequent assignments using the same key update the same entry. It is important to choose a key that is unique, either by using the primary key from a SQL table, or by concatenating strings together to form a unique value.

For example, here is the declaration of an associative array type, and two arrays of that type, using keys that are strings:

DECLARE

TYPE population\_type IS TABLE OF NUMBER INDEX BY VARCHAR2(64);

country\_population population\_type;

continent\_population population\_type;

howmany NUMBER;

which VARCHAR2(64)

BEGIN

country\_population('Greenland') := 100000;

country\_population('Iceland') := 750000;

howmany := country\_population('Greenland');

continent\_population('Australia') := 30000000;

continent\_population('Antarctica') := 1000; -- Creates new entry

continent\_population('Antarctica') := 1001; -- Replaces previous

value

which := continent\_population.FIRST; -- Returns 'Antarctica'

-- as that comes first alphabetically.

which := continent\_population.LAST; -- Returns 'Australia'

howmany := continent\_population(continent\_population.LAST);

-- Returns the value corresponding to the last key, in this

-- case the population of Australia.

END;

/

Associative arrays help you represent data sets of arbitrary size, with fast lookup for an individual element without knowing its position within the array and without having to loop through all the array elements. It is like a simple version of a SQL table where you can retrieve values based on the primary key. For simple temporary storage of lookup data, associative arrays let you avoid using the disk space and network operations required for SQL tables.

Because associative arrays are intended for temporary data rather than storing persistent data, you cannot use them with SQL statements such as INSERT and SELECT INTO. You can make them persistent for the life of a database session by declaring the type in a package and assigning the values in a package body.

**How Globalization Settings Affect VARCHAR2 Keys for Associative Arrays**

If settings for national language or globalization change during a session that uses associative arrays with VARCHAR2 key values, the program might encounter a runtime error. For example, changing the NLS\_COMP or NLS\_SORT initialization parameters within a session might cause methods such as NEXT and PRIOR to raise exceptions. If you need to change these settings during the session, make sure to set them back to their original values before performing further operations with these kinds of associative arrays.

When you declare an associative array using a string as the key, the declaration must use a VARCHAR2, STRING, or LONG type. You can use a different type, such as NCHAR or NVARCHAR2, as the key value to reference an associative array. You can even use a type such as DATE, as long as it can be converted to VARCHAR2 by the TO\_CHAR function.

However, you must be careful when using other types that the values used as keys are consistent and unique. For example, the string value of SYSDATE might change if the NLS\_DATE\_FORMAT initialization parameter changes, so that array\_element(SYSDATE) does not produce the same result as before. Two different NVARCHAR2 values might turn into the same VARCHAR2 value (containing question marks instead of certain national characters). In that case, array\_element(national\_string1) and array\_element(national\_string2) might refer to the same element.

When you pass an associative array as a parameter to a remote database using a database link, the two databases can have different globalization settings. When the remote database performs operations such as FIRST and NEXT, it uses its own character order even if that is different from the order where the collection originated. If character set differences mean that two keys that were unique are not unique on the remote database, the program receives a VALUE\_ERROR exception.

**Choosing Which PL/SQL Collection Types to Use**

If you already have code or business logic that uses some other language, you can usually translate that language's array and set types directly to PL/SQL collection types.

* + Arrays in other languages become VARRAYs in PL/SQL.
  + Sets and bags in other languages become nested tables in PL/SQL.
  + Hash tables and other kinds of unordered lookup tables in other languages become associative arrays in PL/SQL.

When you are writing original code or designing the business logic from the start, you should consider the strengths of each collection type to decide which is appropriate for each situation.

**Choosing Between Nested Tables and Associative Arrays**

Both nested tables and associative arrays (formerly known as index-by tables) use similar subscript notation, but they have different characteristics when it comes to persistence and ease of parameter passing.

Nested tables can be stored in a database column, but associative arrays cannot. Nested tables are appropriate for important data relationships that must be stored persistently.

Associative arrays are appropriate for relatively small lookup tables where the collection can be constructed in memory each time a procedure is called or a package is initialized. They are good for collecting information whose volume is unknown beforehand, because there is no fixed limit on their size. Their index values are more flexible, because associative array subscripts can be negative, can be nonsequential, and can use string values instead of numbers when appropriate.

PL/SQL automatically converts between host arrays and associative arrays that use numeric key values. The most efficient way to pass collections to and from the database server is to use anonymous PL/SQL blocks to bulk-bind input and output host arrays to associative arrays.

**Choosing Between Nested Tables and Varrays**

Varrays are a good choice when the number of elements is known in advance, and when the elements are usually all accessed in sequence. When stored in the database, varrays retain their ordering and subscripts.

Each varray is stored as a single object, either inside the table of which it is a column (if the varray is less than 4KB) or outside the table but still in the same tablespace (if the varray is greater than 4KB). You must update or retrieve all elements of the varray at the same time, which is most appropriate when performing some operation on all the elements at once. But you might find it impractical to store and retrieve large numbers of elements this way.

Nested tables can be sparse: you can delete arbitrary elements, rather than just removing an item from the end. Nested table data is stored out-of-line in a **store table**, a system-generated database table associated with the nested table. This makes nested tables suitable for queries and updates that only affect some elements of the collection. You cannot rely on the order and subscripts of a nested table remaining stable as the table is stored and retrieved, because the order and subscripts are not preserved when a nested table is stored in the database.

**Defining Collection Types**

To create collections, you define a collection type, then declare variables of that type. You can define TABLE and VARRAY types in the declarative part of any PL/SQL block, subprogram, or package.

Collections follow the same scoping and instantiation rules as other types and variables. In a block or subprogram, collections are instantiated when you enter the block or subprogram and cease to exist when you exit. In a package, collections are instantiated when you first reference the package and cease to exist when you end the database session.

**Nested Tables**

For nested tables, use the syntax:

TYPE *type\_name* IS TABLE OF *element\_type* [NOT NULL];

*type\_name* is a type specifier used later to declare collections. For nested tables declared within PL/SQL, *element\_type* is any PL/SQL datatype except:

REF CURSOR

Nested tables declared globally in SQL have additional restrictions on the element type. They cannot use the following element types:

BINARY\_INTEGER, PLS\_INTEGER

BOOLEAN

LONG, LONG RAW

NATURAL, NATURALN

POSITIVE, POSITIVEN

REF CURSOR

SIGNTYPE

STRING

**Varrays**

For varrays, use the syntax:

TYPE *type\_name* IS {VARRAY | VARYING ARRAY} (*size\_limit*)

OF *element\_type* [NOT NULL];

The meanings of *type\_name* and *element\_type* are the same as for nested tables.

*size\_limit* is a positive integer literal representing the maximum number of elements in the array. When defining a VARRAY type, you must specify its maximum size. In the following example, you define a type that stores up to 366 dates:

DECLARE

TYPE Calendar IS VARRAY(366) OF DATE;

**Associative Arrays**

For associative arrays (also known as index-by tables), use the syntax:

TYPE *type\_name* IS TABLE OF *element\_type* [NOT NULL]

INDEX BY [BINARY\_INTEGER | PLS\_INTEGER | VARCHAR2(*size\_limit*)];

INDEX BY *key\_type*;

The key\_type can be numeric, either BINARY\_INTEGER or PLS\_INTEGER. It can also be VARCHAR2 or one of its subtypes VARCHAR, STRING, or LONG. You must specify the length of a VARCHAR2-based key, except for LONG which is equivalent to declaring a key type of VARCHAR2(32760). The types RAW, LONG RAW, ROWID, CHAR, and CHARACTER are not allowed as keys for an associative array.

An initialization clause is not required (or allowed).

When you reference an element of an associative array that uses a VARCHAR2-based key, you can use other types, such as DATE or TIMESTAMP, as long as they can be converted to VARCHAR2 with the TO\_CHAR function.

Index-by tables can store data using a primary key value as the index, where the key values are not sequential. In the example below, you store a single record in the index-by table, and its subscript is 7468 rather than 1.

DECLARE

TYPE EmpTabTyp IS TABLE OF emp%ROWTYPE

INDEX BY BINARY\_INTEGER;

emp\_tab EmpTabTyp;

BEGIN

/\* Retrieve employee record. \*/

SELECT \* INTO emp\_tab(7468) FROM emp WHERE empno = 7468;

END;

**Defining SQL Types Equivalent to PL/SQL Collection Types**

To store nested tables and varrays inside database tables, you must also declare SQL types using the CREATE TYPE statement. The SQL types can be used as columns or as attributes of SQL object types.

You can declare equivalent types within PL/SQL, or use the SQL type name in a PL/SQL variable declaration.

**Nested Table Example**

The following SQL\*Plus script shows how you might declare a nested table in SQL, and use it as an attribute of an object type:

CREATE TYPE CourseList AS TABLE OF VARCHAR2(10) -- define type

/

CREATE TYPE Student AS OBJECT ( -- create object

id\_num INTEGER(4),

name VARCHAR2(25),

address VARCHAR2(35),

status CHAR(2),

courses CourseList) -- declare nested table as attribute

/

The identifier courses represents an entire nested table. Each element of courses will store the code name of a college course such as 'Math 1020'.

**Varray Example**

The script below creates a database column that stores varrays. Each varray element contains a VARCHAR2.

-- Each project has a 16-character code name.

-- We will store up to 50 projects at a time in a database column.

CREATE TYPE ProjectList AS VARRAY(50) OF VARCHAR2(16);

/

CREATE TABLE department ( -- create database table

dept\_id NUMBER(2),

name VARCHAR2(15),

budget NUMBER(11,2),

-- Each department can have up to 50 projects.

projects ProjectList)

/

**Declaring PL/SQL Collection Variables**

Once you define a collection type, you can declare variables of that type. You use the new type name in the declaration, the same as with predefined types such as NUMBER and INTEGER.

**Example: Declaring Nested Tables, Varrays, and Associative Arrays**

DECLARE

TYPE nested\_type IS TABLE OF VARCHAR2(20);

TYPE varray\_type IS VARRAY(50) OF INTEGER;

TYPE associative\_array\_type IS TABLE OF NUMBER

INDEXED BY BINARY\_INTEGER;

v1 nested\_type;

v2 varray\_type;

v3 associative\_array\_type;

**%TYPE Example**

You can use %TYPE to specify the datatype of a previously declared collection, so that changing the definition of the collection automatically updates other variables that depend on the number of elements or the element type:

DECLARE

TYPE Platoon IS VARRAY(20) OF Soldier;

p1 Platoon;

-- If we change the number of soldiers in a platoon, p2 will

-- reflect that change when this block is recompiled.

p2 p1%TYPE;

**Example: Declaring a Procedure Parameter as a Nested Table**

You can declare collections as the formal parameters of functions and procedures. That way, you can pass collections to stored subprograms and from one subprogram to another. The following example declares a nested table as a parameter of a packaged procedure:

CREATE PACKAGE personnel AS

TYPE Staff IS TABLE OF Employee;

...

PROCEDURE award\_bonuses (members IN Staff);

END personnel;

To call PERSONNEL.AWARD\_BONUSES from outside the package, you declare a variable of type PERSONNEL.STAFF and pass that variable as the parameter.

You can also specify a collection type in the RETURN clause of a function specification:

DECLARE

TYPE SalesForce IS VARRAY(25) OF Salesperson;

FUNCTION top\_performers (n INTEGER) RETURN SalesForce IS ...

**Example: Specifying Collection Element Types with %TYPE and %ROWTYPE**

To specify the element type, you can use %TYPE, which provides the datatype of a variable or database column. Also, you can use %ROWTYPE, which provides the rowtype of a cursor or database table. Two examples follow:

DECLARE

TYPE EmpList IS TABLE OF emp.ename%TYPE; -- based on column

CURSOR c1 IS SELECT \* FROM dept;

TYPE DeptFile IS VARRAY(20) OF c1%ROWTYPE; -- based on cursor

**Example: VARRAY of Records**

In the next example, you use a RECORD type to specify the element type:

DECLARE

TYPE AnEntry IS RECORD (

term VARCHAR2(20),

meaning VARCHAR2(200));

TYPE Glossary IS VARRAY(250) OF AnEntry;

**Example: NOT NULL Constraint on Collection Elements**

You can also impose a NOT NULL constraint on the element type:

DECLARE

TYPE EmpList IS TABLE OF emp.empno%TYPE NOT NULL;

**Initializing and Referencing Collections**

Until you initialize it, a nested table or varray is atomically null: the collection itself is null, not its elements. To initialize a nested table or varray, you use a *constructor*, a system-defined function with the same name as the collection type. This function "constructs" collections from the elements passed to it.

You must explicitly call a constructor for each varray and nested table variable. (Associative arrays, the third kind of collection, do not use constructors.) Constructor calls are allowed wherever function calls are allowed.

**Example: Constructor for a Nested Table**

In the following example, you pass multiple elements to the constructor CourseList(), which returns a nested table containing those elements:

DECLARE

TYPE CourseList IS TABLE OF VARCHAR2(16);

my\_courses CourseList;

BEGIN

my\_courses :=

CourseList('Econ 2010', 'Acct 3401', 'Mgmt 3100');

END;

Because a nested table does not have a declared maximum size, you can put as many elements in the constructor as necessary.

**Example: Constructor for a Varray**

In the next example, you pass three objects to constructor ProjectList(), which returns a varray containing those objects:

DECLARE

TYPE ProjectList IS VARRAY(50) OF VARCHAR2(16);

accounting\_projects ProjectList;

BEGIN

accounting\_projects :=

ProjectList('Expense Report', 'Outsourcing', 'Auditing');

END;

You need not initialize the whole varray. For example, if a varray has a maximum size of 50, you can pass fewer than 50 elements to its constructor.

**Example: Collection Constructor Including Null Elements**

Unless you impose the NOT NULL constraint, you can pass null elements to a constructor. An example follows:

BEGIN

my\_courses := CourseList('Math 3010', NULL, 'Stat 3202');

**Example: Combining Collection Declaration and Constructor**

You can initialize a collection in its declaration, which is a good programming practice:

DECLARE

TYPE CourseList IS TABLE OF VARCHAR2(16);

my\_courses CourseList :=

CourseList('Art 1111', 'Hist 3100', 'Engl 2005');

**Example: Empty Varray Constructor**

If you call a constructor without arguments, you get an empty but non-null collection:

DECLARE

TYPE Clientele IS VARRAY(100) OF Customer;

vips Clientele := Clientele(); -- initialize empty varray

BEGIN

IF vips IS NOT NULL THEN -- condition yields TRUE

...

END IF;

END;

In this case, you can call the collection's EXTEND method to add elements later.

**Example: Nested Table Constructor Within a SQL Statement**

In this example, you insert several scalar values and a CourseList nested table into the SOPHOMORES table.

BEGIN

INSERT INTO sophomores

VALUES (5035, 'Janet Alvarez', '122 Broad St', 'FT',

CourseList('Econ 2010', 'Acct 3401', 'Mgmt 3100'));

**Example: Varray Constructor Within a SQL Statement**

In this example, you insert a row into database table DEPARTMENT. The varray constructor ProjectList() provides a value for column PROJECTS.

BEGIN

INSERT INTO department

VALUES(60, 'Security', 750400,

ProjectList('New Badges', 'Track Computers', 'Check Exits'));

**Referencing Collection Elements**

Every reference to an element includes a collection name and a subscript enclosed in parentheses. The subscript determines which element is processed. To reference an element, you specify its subscript using the syntax

*collection\_name*(*subscript*)

where *subscript* is an expression that yields an integer in most cases, or a VARCHAR2 for associative arrays declared with strings as keys.

The allowed subscript ranges are:

* + For nested tables, 1 .. 2\*\*31.
  + For varrays, 1 .. *size\_limit*, where you specify the limit in the declaration.
  + For associative arrays with a numeric key, -2\*\*31 .. 2\*\*31.
  + For associative arrays with a string key, the length of the key and number of possible values depends on the VARCHAR2 length limit in the type declaration, and the database character set.

**Example: Referencing a Nested Table Element By Subscript**

This example shows how to reference an element in the nested table NAMES:

DECLARE

TYPE Roster IS TABLE OF VARCHAR2(15);

names Roster := Roster('J Hamil', 'D Caruso', 'R Singh');

BEGIN

FOR i IN names.FIRST .. names.LAST

LOOP

IF names(i) = 'J Hamil' THEN

NULL;

END IF;

END LOOP;

END;

**Example: Passing a Nested Table Element as a Parameter**

This example shows that you can reference the elements of a collection in subprogram calls:

DECLARE

TYPE Roster IS TABLE OF VARCHAR2(15);

names Roster := Roster('J Hamil', 'D Piro', 'R Singh');

i BINARY\_INTEGER := 2;

BEGIN

verify\_name(names(i)); -- call procedure

END;

**Assigning Collections**

One collection can be assigned to another by an INSERT, UPDATE, FETCH, or SELECT statement, an assignment statement, or a subprogram call.

You can assign the value of an expression to a specific element in a collection using the syntax:

*collection\_name*(*subscript*) := *expression*;

where *expression* yields a value of the type specified for elements in the collection type definition.

**Example: Datatype Compatibility**

This example shows that collections must have the same datatype for an assignment to work. Having the same element type is not enough.

DECLARE

TYPE Clientele IS VARRAY(100) OF Customer;

TYPE Vips IS VARRAY(100) OF Customer;

-- These first two variables have the same datatype.

group1 Clientele := Clientele(...);

group2 Clientele := Clientele(...);

-- This third variable has a similar declaration,

-- but is not the same type.

group3 Vips := Vips(...);

BEGIN

-- Allowed because they have the same datatype

group2 := group1;

-- Not allowed because they have different datatypes

group3 := group2;

END;

**Example: Assigning a Null Value to a Nested Table**

You assign an atomically null nested table or varray to a second nested table or varray. In this case, the second collection must be reinitialized:

DECLARE

TYPE Clientele IS TABLE OF VARCHAR2(64);

-- This nested table has some values.

group1 Clientele := Clientele('Customer 1','Customer 2');

-- This nested table is not initialized ("atomically null").

group2 Clientele;

BEGIN

-- At first, the test IF group1 IS NULL yields FALSE.

-- Then we assign a null nested table to group1.

group1 := group2;

-- Now the test IF group1 IS NULL yields TRUE.

-- We must use another constructor to give it some values.

END;

In the same way, assigning the value NULL to a collection makes it atomically null.

**Example: Possible Exceptions for Collection Assignments**

Assigning a value to a collection element can cause various exceptions:

* + If the subscript is null or is not convertible to the right datatype, PL/SQL raises the predefined exception VALUE\_ERROR. Usually, the subscript must be an integer. Associative arrays can also be declared to have VARCHAR2 subscripts.
  + If the subscript refers to an uninitialized element, PL/SQL raises SUBSCRIPT\_BEYOND\_COUNT.
  + If the collection is atomically null, PL/SQL raises COLLECTION\_IS\_NULL.
  + DECLARE
  + TYPE WordList IS TABLE OF VARCHAR2(5);
  + words WordList;
  + BEGIN
  + /\* Assume execution continues despite the raised exceptions. \*/
  + -- Raises COLLECTION\_IS\_NULL. We haven't used a constructor yet.
  + -- This exception applies to varrays and nested tables, but not
  + -- associative arrays which don't need a constructor.
  + words(1) := 10;
  + -- After using a constructor, we can assign values to the elements.
  + words := WordList(10,20,30);
  + -- Any expression that returns a VARCHAR2(5) is OK.
  + words(1) := 'yes';
  + words(2) := words(1) || 'no';
  + -- Raises VALUE\_ERROR because the assigned value is too long.
  + words(3) := 'longer than 5 characters';
  + -- Raises VALUE\_ERROR because the subscript of a nested table must
  + -- be an integer.
  + words('B') := 'dunno';
  + -- Raises SUBSCRIPT\_BEYOND\_COUNT because we only made 3 elements
  + -- in the constructor. To add new ones, we must call the EXTEND
  + -- method first.
  + words(4) := 'maybe';
  + END;

**Comparing Collections**

You can check whether a collection is null, but not test whether two collections are the same. Conditions such as greater than, less than, and so on are also not allowed.

**Example: Checking if a Collection Is Null**

Nested tables and varrays can be atomically null, so they can be tested for nullity:

DECLARE

TYPE Staff IS TABLE OF Employee;

members Staff;

BEGIN

-- Condition yields TRUE because we haven't used a constructor.

IF members IS NULL THEN ...

END;

**Example: Comparing Two Collections**

Collections cannot be directly compared for equality or inequality. For instance, the following IF condition is not allowed:

DECLARE

TYPE Clientele IS TABLE OF VARCHAR2(64);

group1 Clientele := Clientele('Customer 1', 'Customer 2');

group2 Clientele := Clientele('Customer 1', 'Customer 3');

BEGIN

-- Equality test causes compilation error.

IF group1 = group2 THEN

...

END IF;

END;

This restriction also applies to implicit comparisons. For example, collections cannot appear in a DISTINCT, GROUP BY, or ORDER BY list.

If you want to do such comparison operations, you must define your own notion of what it means for collections to be equal or greater than, less than, and so on, and write one or more functions to examine the collections and their elements and return a true or false value.

**Using PL/SQL Collections with SQL Statements**

Collections let you manipulate complex datatypes within PL/SQL. Your program can compute subscripts to process specific elements in memory, and use SQL to store the results in database tables.

**Example: Creating a SQL Type Corresponding to a PL/SQL Nested Table**

In SQL\*Plus, you can create SQL types whose definitions correspond to PL/SQL nested tables and varrays:

SQL> CREATE TYPE CourseList AS TABLE OF VARCHAR2(64);

You can use these SQL types as columns in database tables:

SQL> CREATE TABLE department (

2 name VARCHAR2(20),

3 director VARCHAR2(20),

4 office VARCHAR2(20),

5 courses CourseList)

6 NESTED TABLE courses STORE AS courses\_tab;

Each item in column COURSES is a nested table that will store the courses offered by a given department. The NESTED TABLE clause is required whenever a database table has a nested table column. The clause identifies the nested table and names a system-generated store table, in which Oracle stores the nested table data.

**Example: Inserting a Nested Table into a Database Table**

Now, you can populate the database table. The table constructor provides values that all go into the single column COURSES:

BEGIN

INSERT INTO department

VALUES('English', 'Lynn Saunders', 'Breakstone Hall 205',

CourseList('Expository Writing',

'Film and Literature',

'Modern Science Fiction',

'Discursive Writing',

'Modern English Grammar',

'Introduction to Shakespeare',

'Modern Drama',

'The Short Story',

'The American Novel'));

END;

**Example: Retrieving a PL/SQL Nested Table from a Database Table**

You can retrieve all the courses offered by the English department into a PL/SQL nested table:

DECLARE

english\_courses CourseList;

BEGIN

SELECT courses INTO english\_courses FROM department

WHERE name = 'English';

END;

Within PL/SQL, you can manipulate the nested table by looping through its elements, using methods such as TRIM or EXTEND, and updating some or all of the elements. Afterwards, you can store the updated table in the database again.

**Example: Updating a Nested Table within a Database Table**

You can revise the list of courses offered by the English Department:

DECLARE

new\_courses CourseList :=

CourseList('Expository Writing',

'Film and Literature',

'Discursive Writing',

'Modern English Grammar',

'Realism and Naturalism',

'Introduction to Shakespeare',

'Modern Drama',

'The Short Story',

'The American Novel',

'20th-Century Poetry',

'Advanced Workshop in Poetry');

BEGIN

UPDATE department

SET courses = new\_courses WHERE name = 'English';

END;

**Some Varray Examples**

In SQL\*Plus, suppose you define object type Project, as follows:

SQL> CREATE TYPE Project AS OBJECT (

2 project\_no NUMBER(2),

3 title VARCHAR2(35),

4 cost NUMBER(7,2));

Next, you define VARRAY type ProjectList, which stores Project objects:

SQL> CREATE TYPE ProjectList AS VARRAY(50) OF Project;

Finally, you create relational table department, which has a column of type ProjectList, as follows:

SQL> CREATE TABLE department (

2 dept\_id NUMBER(2),

3 name VARCHAR2(15),

4 budget NUMBER(11,2),

5 projects ProjectList);

Each item in column projects is a varray that will store the projects scheduled for a given department.

Now, you are ready to populate relational table department. In the following example, notice how varray constructor ProjectList() provides values for column projects:

BEGIN

INSERT INTO department

VALUES(30, 'Accounting', 1205700,

ProjectList(Project(1, 'Design New Expense Report', 3250),

Project(2, 'Outsource Payroll', 12350),

Project(3, 'Evaluate Merger Proposal', 2750),

Project(4, 'Audit Accounts Payable', 1425)));

INSERT INTO department

VALUES(50, 'Maintenance', 925300,

ProjectList(Project(1, 'Repair Leak in Roof', 2850),

Project(2, 'Install New Door Locks', 1700),

Project(3, 'Wash Front Windows', 975),

Project(4, 'Repair Faulty Wiring', 1350),

Project(5, 'Winterize Cooling System', 1125)));

INSERT INTO department

VALUES(60, 'Security', 750400,

ProjectList(Project(1, 'Issue New Employee Badges', 13500),

Project(2, 'Find Missing IC Chips', 2750),

Project(3, 'Upgrade Alarm System', 3350),

Project(4, 'Inspect Emergency Exits', 1900)));

END;

In the following example, you update the list of projects assigned to the Security Department:

DECLARE

new\_projects ProjectList :=

ProjectList(Project(1, 'Issue New Employee Badges', 13500),

Project(2, 'Develop New Patrol Plan', 1250),

Project(3, 'Inspect Emergency Exits', 1900),

Project(4, 'Upgrade Alarm System', 3350),

Project(5, 'Analyze Local Crime Stats', 825));

BEGIN

UPDATE department

SET projects = new\_projects WHERE dept\_id = 60;

END;

In the next example, you retrieve all the projects for the Accounting Department into a local varray:

DECLARE

my\_projects ProjectList;

BEGIN

SELECT projects INTO my\_projects FROM department

WHERE dept\_id = 30;

END;

In the final example, you delete the Accounting Department and its project list from table department:

BEGIN

DELETE FROM department WHERE dept\_id = 30;

END;

**Manipulating Individual Collection Elements with SQL**

By default, SQL operations store and retrieve whole collections rather than individual elements. To manipulate the individual elements of a collection with SQL, use the TABLE operator. The TABLE operator uses a subquery to extract the varray or nested table, so that the INSERT, UPDATE, or DELETE statement applies to the nested table rather than the top-level table.

**Example: Inserting an Element into a Nested Table with SQL**

In the following example, you add a row to the History Department nested table stored in column COURSES:

BEGIN

-- The TABLE operator makes the statement apply to the nested

-- table from the 'History' row of the DEPARTMENT table.

INSERT INTO

TABLE(SELECT courses FROM department WHERE name = 'History')

VALUES('Modern China');

END;

**Example: Updating Elements Inside a Nested Table with SQL**

In the next example, you abbreviate the names for some courses offered by the Psychology Department:

BEGIN

UPDATE TABLE(SELECT courses FROM department

WHERE name = 'Psychology')

SET credits = credits + adjustment

WHERE course\_no IN (2200, 3540);

END;

**Example: Retrieving a Single Element from a Nested Table with SQL**

In the following example, you retrieve the title of a specific course offered by the History Department:

DECLARE

my\_title VARCHAR2(64);

BEGIN

-- We know that there is one history course with 'Etruscan'

-- in the title. This query retrieves the complete title

-- from the nested table of courses for the History department.

SELECT title INTO my\_title

FROM

TABLE(SELECT courses FROM department WHERE name = 'History')

WHERE name LIKE '%Etruscan%';

END;

**Example: Deleting Elements from a Nested Table with SQL**

In the next example, you delete all 5-credit courses offered by the English Department:

BEGIN

DELETE TABLE(SELECT courses FROM department

WHERE name = 'English')

WHERE credits = 5;

END;

**Example: Retrieving Elements from a Varray with SQL**

In the following example, you retrieve the title and cost of the Maintenance Department's fourth project from the varray column projects:

DECLARE

my\_cost NUMBER(7,2);

my\_title VARCHAR2(35);

BEGIN

SELECT cost, title INTO my\_cost, my\_title

FROM TABLE(SELECT projects FROM department

WHERE dept\_id = 50)

WHERE project\_no = 4;

...

END;

**Example: Performing INSERT, UPDATE, and DELETE Operations on a Varray with SQL**

Currently, you cannot reference the individual elements of a varray in an INSERT, UPDATE, or DELETE statement. You must retrieve the entire varray, use PL/SQL procedural statements to add, delete, or update its elements, and then store the changed varray back in the database table.

In the following example, stored procedure ADD\_PROJECT inserts a new project into a department's project list at a given position:

CREATE PROCEDURE add\_project (

dept\_no IN NUMBER,

new\_project IN Project,

position IN NUMBER) AS

my\_projects ProjectList;

BEGIN

SELECT projects INTO my\_projects FROM department

WHERE dept\_no = dept\_id FOR UPDATE OF projects;

my\_projects.EXTEND; -- make room for new project

/\* Move varray elements forward. \*/

FOR i IN REVERSE position..my\_projects.LAST - 1 LOOP

my\_projects(i + 1) := my\_projects(i);

END LOOP;

my\_projects(position) := new\_project; -- add new project

UPDATE department SET projects = my\_projects

WHERE dept\_no = dept\_id;

END add\_project;

The following stored procedure updates a given project:

CREATE PROCEDURE update\_project (

dept\_no IN NUMBER,

proj\_no IN NUMBER,

new\_title IN VARCHAR2 DEFAULT NULL,

new\_cost IN NUMBER DEFAULT NULL) AS

my\_projects ProjectList;

BEGIN

SELECT projects INTO my\_projects FROM department

WHERE dept\_no = dept\_id FOR UPDATE OF projects;

/\* Find project, update it, then exit loop immediately. \*/

FOR i IN my\_projects.FIRST..my\_projects.LAST LOOP

IF my\_projects(i).project\_no = proj\_no THEN

IF new\_title IS NOT NULL THEN

my\_projects(i).title := new\_title;

END IF;

IF new\_cost IS NOT NULL THEN

my\_projects(i).cost := new\_cost;

END IF;

EXIT;

END IF;

END LOOP;

UPDATE department SET projects = my\_projects

WHERE dept\_no = dept\_id;

END update\_project;

**Example: Performing INSERT, UPDATE, and DELETE Operations on PL/SQL Nested Tables**

To perform DML operations on a PL/SQL nested table, use the operators TABLE and CAST. This way, you can do set operations on nested tables using SQL notation, without actually storing the nested tables in the database.

The operands of CAST are PL/SQL collection variable and a SQL collection type (created by the CREATE TYPE statement). CAST converts the PL/SQL collection to the SQL type.

The following example counts the number of differences between a revised course list and the original (notice that the number of credits for course 3720 changed from 4 to 3):

DECLARE

revised CourseList :=

CourseList(Course(1002, 'Expository Writing', 3),

Course(2020, 'Film and Literature', 4),

Course(2810, 'Discursive Writing', 4),

Course(3010, 'Modern English Grammar ', 3),

Course(3550, 'Realism and Naturalism', 4),

Course(3720, 'Introduction to Shakespeare', 3),

Course(3760, 'Modern Drama', 4),

Course(3822, 'The Short Story', 4),

Course(3870, 'The American Novel', 5),

Course(4210, '20th-Century Poetry', 4),

Course(4725, 'Advanced Workshop in Poetry', 5));

num\_changed INTEGER;

BEGIN

SELECT COUNT(\*) INTO num\_changed

FROM TABLE(CAST(revised AS CourseList)) new,

TABLE(SELECT courses FROM department

WHERE name = 'English') AS old

WHERE new.course\_no = old.course\_no AND

(new.title != old.title OR new.credits != old.credits);

dbms\_output.put\_line(num\_changed);

END;

**Using Multilevel Collections**

In addition to collections of scalar or object types, you can also create collections whose elements are collections. For example, you can create a nested table of varrays, a varray of varrays, a varray of nested tables, and so on.

When creating a nested table of nested tables as a column in SQL, check the syntax of the CREATE TABLE statement to see how to define the storage table.

Here are some examples showing the syntax and possibilities for multilevel collections.

**Multilevel VARRAY Example**

declare

type t1 is varray(10) of integer;

type nt1 is varray(10) of t1; -- multilevel varray type

va t1 := t1(2,3,5);

-- initialize multilevel varray

nva nt1 := nt1(va, t1(55,6,73), t1(2,4), va);

i integer;

va1 t1;

begin

-- multilevel access

i := nva(2)(3); -- i will get value 73

dbms\_output.put\_line(i);

-- add a new varray element to nva

nva.extend;

nva(5) := t1(56, 32);

-- replace an inner varray element

nva(4) := t1(45,43,67,43345);

-- replace an inner integer element

nva(4)(4) := 1; -- replaces 43345 with 1

-- add a new element to the 4th varray element

-- and store integer 89 into it.

nva(4).extend;

nva(4)(5) := 89;

end;

/

**Multilevel Nested Table Example**

declare

type tb1 is table of varchar2(20);

type ntb1 is table of tb1; -- table of table elements

type tv1 is varray(10) of integer;

type ntb2 is table of tv1; -- table of varray elements

vtb1 tb1 := tb1('one', 'three');

vntb1 ntb1 := ntb1(vtb1);

vntb2 ntb2 := ntb2(tv1(3,5), tv1(5,7,3)); -- table of varray

elements

begin

vntb1.extend;

vntb1(2) := vntb1(1);

-- delete the first element in vntb1

vntb1.delete(1);

-- delete the first string from the second table in the nested

table

vntb1(2).delete(1);

end;

/

**Multilevel Associative Array Example**

declare

type tb1 is table of integer index by binary\_integer;

-- the following is index-by table of index-by tables

type ntb1 is table of tb1 index by binary\_integer;

type va1 is varray(10) of varchar2(20);

-- the following is index-by table of varray elements

type ntb2 is table of va1 index by binary\_integer;

v1 va1 := va1('hello', 'world');

v2 ntb1;

v3 ntb2;

v4 tb1;

v5 tb1; -- empty table

begin

v4(1) := 34;

v4(2) := 46456;

v4(456) := 343;

v2(23) := v4;

v3(34) := va1(33, 456, 656, 343);

-- assign an empty table to v2(35) and try again

v2(35) := v5;

v2(35)(2) := 78; -- it works now

end;

/

**Example of Multilevel Collections and Bulk SQL**

create type t1 is varray(10) of integer;

/

create table tab1 (c1 t1);

insert into tab1 values (t1(2,3,5));

insert into tab1 values (t1(9345, 5634, 432453));

declare

type t2 is table of t1;

v2 t2;

begin

select c1 BULK COLLECT INTO v2 from tab1;

dbms\_output.put\_line(v2.count); -- prints 2

end;

/

**Using Collection Methods**

The following collection methods help generalize code, make collections easier to use, and make your applications easier to maintain:

EXISTS

COUNT

LIMIT

FIRST and LAST

PRIOR and NEXT

EXTEND

TRIM

DELETE

A **collection method** is a built-in function or procedure that operates on collections and is called using dot notation. The syntax follows:

collection\_name.method\_name[(parameters)]

Collection methods cannot be called from SQL statements. Also, EXTEND and TRIM cannot be used with associative arrays. EXISTS, COUNT, LIMIT, FIRST, LAST, PRIOR, and NEXT are functions; EXTEND, TRIM, and DELETE are procedures. EXISTS, PRIOR, NEXT, TRIM, EXTEND, and DELETE take parameters corresponding to collection subscripts, which are usually integers but can also be strings for associative arrays.

Only EXISTS can be applied to atomically null collections. If you apply another method to such collections, PL/SQL raises COLLECTION\_IS\_NULL.

**Checking If a Collection Element Exists (EXISTS Method)**

EXISTS(n) returns TRUE if the *n*th element in a collection exists. Otherwise, EXISTS(n) returns FALSE. Mainly, you use EXISTS with DELETE to maintain sparse nested tables. You can also use EXISTS to avoid raising an exception when you reference a nonexistent element. In the following example, PL/SQL executes the assignment statement only if element i exists:

IF courses.EXISTS(i) THEN courses(i) := new\_course; END IF;

When passed an out-of-range subscript, EXISTS returns FALSE instead of raising SUBSCRIPT\_OUTSIDE\_LIMIT.

**Counting the Elements in a Collection (COUNT Method)**

COUNT returns the number of elements that a collection currently contains. For instance, if varray projects contains 25 elements, the following IF condition is true:

IF projects.COUNT = 25 THEN ...

COUNT is useful because the current size of a collection is not always known. For example, if you fetch a column of Oracle data into a nested table, how many elements does the table contain? COUNT gives you the answer.

You can use COUNT wherever an integer expression is allowed. In the next example, you use COUNT to specify the upper bound of a loop range:

FOR i IN 1..courses.COUNT LOOP ...

For varrays, COUNT always equals LAST. For nested tables, COUNT normally equals LAST. But, if you delete elements from the middle of a nested table, COUNT becomes smaller than LAST.

When tallying elements, COUNT ignores deleted elements.

**Checking the Maximum Size of a Collection (LIMIT Method)**

For nested tables and associative arrays, which have no maximum size, LIMIT returns NULL. For varrays, LIMIT returns the maximum number of elements that a varray can contain (which you must specify in its type definition, and can change later with the TRIM and EXTEND methods). For instance, if the maximum size of varray PROJECTS is 25 elements, the following IF condition is true:

IF projects.LIMIT = 25 THEN ...

You can use LIMIT wherever an integer expression is allowed. In the following example, you use LIMIT to determine if you can add 15 more elements to varray projects:

IF (projects.COUNT + 15) < projects.LIMIT THEN ...

**Finding the First or Last Collection Element (FIRST and LAST Methods)**

FIRST and LAST return the first and last (smallest and largest) index numbers in a collection. For an associative array with VARCHAR2 key values, the lowest and highest key values are returned; ordering is based on the binary values of the characters in the string, unless the NLS\_COMP initialization parameter is set to ANSI, in which case the ordering is based on the locale-specific sort order specified by the NLS\_SORT initialization parameter.

If the collection is empty, FIRST and LAST return NULL.

If the collection contains only one element, FIRST and LAST return the same index value:

IF courses.FIRST = courses.LAST THEN ... -- only one element

The next example shows that you can use FIRST and LAST to specify the lower and upper bounds of a loop range provided each element in that range exists:

FOR i IN courses.FIRST..courses.LAST LOOP ...

In fact, you can use FIRST or LAST wherever an integer expression is allowed. In the following example, you use FIRST to initialize a loop counter:

i := courses.FIRST;

WHILE i IS NOT NULL LOOP ...

For varrays, FIRST always returns 1 and LAST always equals COUNT. For nested tables, FIRST normally returns 1. But, if you delete elements from the beginning of a nested table, FIRST returns a number larger than 1. Also for nested tables, LAST normally equals COUNT. But, if you delete elements from the middle of a nested table, LAST becomes larger than COUNT.

When scanning elements, FIRST and LAST ignore deleted elements.

**Looping Through Collection Elements (PRIOR and NEXT Methods)**

PRIOR(n) returns the index number that precedes index n in a collection. NEXT(n) returns the index number that succeeds index n. If n has no predecessor, PRIOR(n) returns NULL. Likewise, if n has no successor, NEXT(n) returns NULL.

For associative arrays with VARCHAR2 keys, these methods return the appropriate key value; ordering is based on the binary values of the characters in the string, unless the NLS\_COMP initialization parameter is set to ANSI, in which case the ordering is based on the locale-specific sort order specified by the NLS\_SORT initialization parameter.

These methods are more reliable than looping through a fixed set of subscript values, because elements might be inserted or deleted from the collection during the loop. This is especially true for associative arrays, where the subscripts might not be in consecutive order and so the sequence of subscripts might be (1,2,4,8,16) or ('A','E','I','O','U').

PRIOR and NEXT do not wrap from one end of a collection to the other. For example, the following statement assigns NULL to n because the first element in a collection has no predecessor:

n := courses.PRIOR(courses.FIRST); -- assigns NULL to n

PRIOR is the inverse of NEXT. For instance, if element i exists, the following statement assigns element i to itself:

projects(i) := projects.PRIOR(projects.NEXT(i));

You can use PRIOR or NEXT to traverse collections indexed by any series of subscripts. In the following example, you use NEXT to traverse a nested table from which some elements have been deleted:

i := courses.FIRST; -- get subscript of first element

WHILE i IS NOT NULL LOOP

-- do something with courses(i)

i := courses.NEXT(i); -- get subscript of next element

END LOOP;

When traversing elements, PRIOR and NEXT ignore deleted elements.

**Increasing the Size of a Collection (EXTEND Method)**

To increase the size of a nested table or varray, use EXTEND. You cannot use EXTEND with index-by tables.

This procedure has three forms:

* + EXTEND appends one null element to a collection.
  + EXTEND(n) appends n null elements to a collection.
  + EXTEND(n,i) appends n copies of the *i*th element to a collection.

For example, the following statement appends 5 copies of element 1 to nested table courses:

courses.EXTEND(5,1);

You cannot use EXTEND to initialize an atomically null collection. Also, if you impose the NOT NULL constraint on a TABLE or VARRAY type, you cannot apply the first two forms of EXTEND to collections of that type.

EXTEND operates on the internal size of a collection, which includes any deleted elements. So, if EXTEND encounters deleted elements, it includes them in its tally. PL/SQL keeps placeholders for deleted elements so that you can replace them if you wish. Consider the following example:

DECLARE

TYPE CourseList IS TABLE OF VARCHAR2(10);

courses CourseList;

BEGIN

courses := CourseList('Biol 4412', 'Psyc 3112', 'Anth 3001');

courses.DELETE(3); -- delete element 3

/\* PL/SQL keeps a placeholder for element 3. So, the

next statement appends element 4, not element 3. \*/

courses.EXTEND; -- append one null element

/\* Now element 4 exists, so the next statement does

not raise SUBSCRIPT\_BEYOND\_COUNT. \*/

courses(4) := 'Engl 2005';

When it includes deleted elements, the internal size of a nested table differs from the values returned by COUNT and LAST. For instance, if you initialize a nested table with five elements, then delete elements 2 and 5, the internal size is 5, COUNT returns 3, and LAST returns 4. All deleted elements (whether leading, in the middle, or trailing) are treated alike.

**Decreasing the Size of a Collection (TRIM Method)**

This procedure has two forms:

* + TRIM removes one element from the end of a collection.
  + TRIM(n) removes n elements from the end of a collection.

For example, this statement removes the last three elements from nested table courses:

courses.TRIM(3);

If n is too large, TRIM(n) raises SUBSCRIPT\_BEYOND\_COUNT.

TRIM operates on the internal size of a collection. So, if TRIM encounters deleted elements, it includes them in its tally. Consider the following example:

DECLARE

TYPE CourseList IS TABLE OF VARCHAR2(10);

courses CourseList;

BEGIN

courses := CourseList('Biol 4412', 'Psyc 3112', 'Anth 3001');

courses.DELETE(courses.LAST); -- delete element 3

/\* At this point, COUNT equals 2, the number of valid

elements remaining. So, you might expect the next

statement to empty the nested table by trimming

elements 1 and 2. Instead, it trims valid element 2

and deleted element 3 because TRIM includes deleted

elements in its tally. \*/

courses.TRIM(courses.COUNT);

dbms\_output.put\_line(courses(1)); -- prints 'Biol 4412'

In general, do not depend on the interaction between TRIM and DELETE. It is better to treat nested tables like fixed-size arrays and use only DELETE, or to treat them like stacks and use only TRIM and EXTEND.

PL/SQL does not keep placeholders for trimmed elements. So, you cannot replace a trimmed element simply by assigning it a new value.

**Deleting Collection Elements (DELETE Method)**

This procedure has various forms:

* + DELETE removes all elements from a collection.
  + DELETE(n) removes the *n*th element from an associative array with a numeric key or a nested table. If the associative array has a string key, the element corresponding to the key value is deleted. If n is null, DELETE(n) does nothing.
  + DELETE(m,n) removes all elements in the range m..n from an associative array or nested table. If m is larger than n or if m or n is null, DELETE(m,n) does nothing.

For example:

BEGIN

courses.DELETE(2); -- deletes element 2

courses.DELETE(7,7); -- deletes element 7

courses.DELETE(6,3); -- does nothing

courses.DELETE(3,6); -- deletes elements 3 through 6

projects.DELETE; -- deletes all elements

nicknames.DELETE('Chip'); -- deletes element denoted by this key

nicknames.DELETE('Buffy','Fluffy'); -- deletes elements with keys

-- in this alphabetic range

END;

Varrays are dense, so you cannot delete their individual elements.

If an element to be deleted does not exist, DELETE simply skips it; no exception is raised. PL/SQL keeps placeholders for deleted elements. So, you can replace a deleted element simply by assigning it a new value.

DELETE lets you maintain sparse nested tables. In the following example, you retrieve nested table prospects into a temporary table, prune it, then store it back in the database:

DECLARE

my\_prospects ProspectList;

revenue NUMBER;

BEGIN

SELECT prospects INTO my\_prospects FROM customers WHERE ...

FOR i IN my\_prospects.FIRST..my\_prospects.LAST LOOP

estimate\_revenue(my\_prospects(i), revenue); -- call procedure

IF revenue < 25000 THEN

my\_prospects.DELETE(i);

END IF;

END LOOP;

UPDATE customers SET prospects = my\_prospects WHERE ...

The amount of memory allocated to a nested table can increase or decrease dynamically. As you delete elements, memory is freed page by page. If you delete the entire table, all the memory is freed.

**Applying Methods to Collection Parameters**

Within a subprogram, a collection parameter assumes the properties of the argument bound to it. So, you can apply the built-in collection methods (FIRST, LAST, COUNT, and so on) to such parameters. In the following example, a nested table is declared as the formal parameter of a packaged procedure:

CREATE PACKAGE personnel AS

TYPE Staff IS TABLE OF Employee;

...

PROCEDURE award\_bonuses (members IN Staff);

END personnel;

CREATE PACKAGE BODY personnel AS

...

PROCEDURE award\_bonuses (members IN Staff) IS

BEGIN

...

IF members.COUNT > 10 THEN -- apply method

...

END IF;

END;

END personnel;

**Note:** For varray parameters, the value of LIMIT is always derived from the parameter type definition, regardless of the parameter mode.

**Avoiding Collection Exceptions**

In most cases, if you reference a nonexistent collection element, PL/SQL raises a predefined exception. Consider the following example:

DECLARE

TYPE NumList IS TABLE OF NUMBER;

nums NumList; -- atomically null

BEGIN

/\* Assume execution continues despite the raised exceptions. \*/

nums(1) := 1; -- raises COLLECTION\_IS\_NULL (1)

nums := NumList(1,2); -- initialize table

nums(NULL) := 3 -- raises VALUE\_ERROR (2)

nums(0) := 3; -- raises SUBSCRIPT\_OUTSIDE\_LIMIT (3)

nums(3) := 3; -- raises SUBSCRIPT\_BEYOND\_COUNT (4)

nums.DELETE(1); -- delete element 1

IF nums(1) = 1 THEN ... -- raises NO\_DATA\_FOUND (5)

In the first case, the nested table is atomically null. In the second case, the subscript is null. In the third case, the subscript is outside the legal range. In the fourth case, the subscript exceeds the number of elements in the table. In the fifth case, the subscript designates a deleted element.

The following list shows when a given exception is raised:

| **Collection Exception** | **Raised when...** |
| --- | --- |
| COLLECTION\_IS\_NULL | you try to operate on an atomically null collection. |
| NO\_DATA\_FOUND | a subscript designates an element that was deleted, or a nonexistent element of an associative array. |
| SUBSCRIPT\_BEYOND\_COUNT | a subscript exceeds the number of elements in a collection. |
| SUBSCRIPT\_OUTSIDE\_LIMIT | a subscript is outside the allowed range. |
| VALUE\_ERROR | a subscript is null or not convertible to the key type. This exception might occur if the key is defined as a PLS\_INTEGER range, and the subscript is outside this range. |

In some cases, you can pass invalid subscripts to a method without raising an exception. For instance, when you pass a null subscript to procedure DELETE, it does nothing. Also, you can replace deleted elements without raising NO\_DATA\_FOUND, as the following example shows:

DECLARE

TYPE NumList IS TABLE OF NUMBER;

nums NumList := NumList(10,20,30); -- initialize table

BEGIN

nums.DELETE(-1); -- does not raise SUBSCRIPT\_OUTSIDE\_LIMIT

nums.DELETE(3); -- delete 3rd element

dbms\_output.put\_line(nums.COUNT); -- prints 2

nums(3) := 30; -- allowed; does not raise NO\_DATA\_FOUND

dbms\_output.put\_line(nums.COUNT); -- prints 3

END;

Packaged collection types and local collection types are never compatible. For example, suppose you want to call the following packaged procedure:

CREATE PACKAGE pkg1 AS

TYPE NumList IS VARRAY(25) OF NUMBER(4);

PROCEDURE delete\_emps (emp\_list NumList);

END pkg1;

CREATE PACKAGE BODY pkg1 AS

PROCEDURE delete\_emps (emp\_list NumList) IS ...

...

END pkg1;

When you run the PL/SQL block below, the second procedure call fails with a *wrong number or types of arguments* error. That is because the packaged and local VARRAY types are incompatible even though their definitions are identical.

DECLARE

TYPE NumList IS VARRAY(25) OF NUMBER(4);

emps pkg1.NumList := pkg1.NumList(7369, 7499);

emps2 NumList := NumList(7521, 7566);

BEGIN

pkg1.delete\_emps(emps);

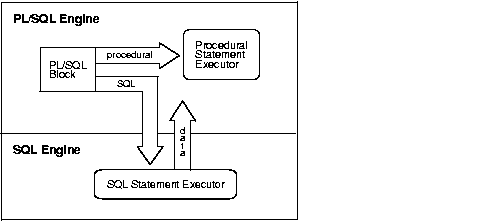
pkg1.delete\_emps(emps2); -- causes a compilation error

END;

**Reducing Loop Overhead for Collections with Bulk Binds**

As [Figure 5-3](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/05_colls.htm#28103) shows, the PL/SQL engine executes procedural statements but sends SQL statements to the SQL engine, which executes the SQL statements and, in some cases, returns data to the PL/SQL engine.

***Figure 5-3 Context Switching***

 [Text description of the illustration pls81027\_context\_switching.gif](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/img_text/pls81027_context_switching.htm)

Too many context switches between the PL/SQL and SQL engines can harm performance. That can happen when a loop executes a separate SQL statement for each element of a collection, specifying the collection element as a bind variable. For example, the following DELETE statement is sent to the SQL engine with each iteration of the FOR loop:

DECLARE

TYPE NumList IS VARRAY(20) OF NUMBER;

depts NumList := NumList(10, 30, 70); -- department numbers

BEGIN

...

FOR i IN depts.FIRST..depts.LAST LOOP

DELETE FROM emp WHERE deptno = depts(i);

END LOOP;

END;

In such cases, if the SQL statement affects four or more database rows, the use of bulk binds can improve performance considerably.

**How Do Bulk Binds Improve Performance?**

The assigning of values to PL/SQL variables in SQL statements is called **binding**. PL/SQL binding operations fall into three categories:

* + **in-bind** When a PL/SQL variable or host variable is stored in the database by an INSERT or UPDATE statement.
  + **out-bind** When a database value is assigned to a PL/SQL variable or a host variable by the RETURNING clause of an INSERT, UPDATE, or DELETE statement.
  + **define** When a database value is assigned to a PL/SQL variable or a host variable by a SELECT or FETCH statement.

A DML statement can transfer all the elements of a collection in a single operation, a process known as **bulk binding**. If the collection has 20 elements, bulk binding lets you perform the equivalent of 20 SELECT, INSERT, UPDATE, or DELETE statements using a single operation. This technique improves performance by minimizing the number of context switches between the PL/SQL and SQL engines. With bulk binds, entire collections, not just individual elements, are passed back and forth.

To do bulk binds with INSERT, UPDATE, and DELETE statements, you enclose the SQL statement within a PL/SQL FORALL statement.

To do bulk binds with SELECT statements, you include the BULK COLLECT clause in the SELECT statement instead of using INTO.

For full details of the syntax and restrictions for these statements, see ["FORALL Statement"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/13_elems22.htm#34325) and ["SELECT INTO Statement"](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/13_elems45.htm#36067).

**Example: Performing a Bulk Bind with DELETE**

The following DELETE statement is sent to the SQL engine just once, even though it performs three DELETE operations:

DECLARE

TYPE NumList IS VARRAY(20) OF NUMBER;

depts NumList := NumList(10, 30, 70); -- department numbers

BEGIN

FORALL i IN depts.FIRST..depts.LAST

DELETE FROM emp WHERE deptno = depts(i);

END;

**Example: Performing a Bulk Bind with INSERT**

In the example below, 5000 part numbers and names are loaded into index-by tables. All table elements are inserted into a database table twice: first using a FOR loop, then using a FORALL statement. The FORALL version is much faster.

SQL> SET SERVEROUTPUT ON

SQL> CREATE TABLE parts (pnum NUMBER(4), pname CHAR(15));

Table created.

SQL> GET test.sql

1 DECLARE

2 TYPE NumTab IS TABLE OF NUMBER(4) INDEX BY BINARY\_INTEGER;

3 TYPE NameTab IS TABLE OF CHAR(15) INDEX BY BINARY\_INTEGER;

4 pnums NumTab;

5 pnames NameTab;

6 t1 NUMBER(5);

7 t2 NUMBER(5);

8 t3 NUMBER(5);

9

10

11 BEGIN

12 FOR j IN 1..5000 LOOP -- load index-by tables

13 pnums(j) := j;

14 pnames(j) := 'Part No. ' || TO\_CHAR(j);

15 END LOOP;

16 t1 := dbms\_utility.get\_time;

17 FOR i IN 1..5000 LOOP -- use FOR loop

18 INSERT INTO parts VALUES (pnums(i), pnames(i));

19 END LOOP;

20 t2 := dbms\_utility.get\_time;

21 FORALL i IN 1..5000 -- use FORALL statement

22 INSERT INTO parts VALUES (pnums(i), pnames(i));

23 get\_time(t3);

24 dbms\_output.put\_line('Execution Time (secs)');

25 dbms\_output.put\_line('---------------------');

26 dbms\_output.put\_line('FOR loop: ' || TO\_CHAR(t2 - t1));

27 dbms\_output.put\_line('FORALL: ' || TO\_CHAR(t3 - t2));

28\* END;

SQL> /

Execution Time (secs)

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

FOR loop: 32

FORALL: 3

PL/SQL procedure successfully completed.

**Using the FORALL Statement**

The keyword FORALL instructs the PL/SQL engine to bulk-bind input collections before sending them to the SQL engine. Although the FORALL statement contains an iteration scheme, it is *not* a FOR loop. Its syntax follows:

FORALL *index* IN *lower\_bound*..*upper\_bound*

*sql\_statement*;

The index can be referenced only within the FORALL statement and only as a collection subscript. The SQL statement must be an INSERT, UPDATE, or DELETE statement that references collection elements. And, the bounds must specify a valid range of consecutive index numbers. The SQL engine executes the SQL statement once for each index number in the range.

**Example: Using FORALL with Part of a Collection**

As the following example shows, the bounds of the FORALL loop can apply to part of a collection, not necessarily all the elements:

DECLARE

TYPE NumList IS VARRAY(10) OF NUMBER;

depts NumList := NumList(20,30,50,55,57,60,70,75,90,92);

BEGIN

FORALL j IN 4..7 -- bulk-bind only part of varray

UPDATE emp SET sal = sal \* 1.10 WHERE deptno = depts(j);

END;

**Example: Bulk Bind Requires Subscripted Collection**

The SQL statement can reference more than one collection. However, the PL/SQL engine bulk-binds only subscripted collections. So, in the following example, it does not bulk-bind the collection sals, which is passed to the function median:

FORALL i IN 1..20

INSERT INTO emp2 VALUES (enums(i), names(i), median(sals), ...);

**Example: Inserting into an Object Table with FORALL**

In addition to relational tables, the FORALL statement can manipulate object tables, as the following example shows:

CREATE TYPE PNum AS OBJECT (n NUMBER);

/

CREATE TABLE partno OF PNum;

DECLARE

TYPE NumTab IS TABLE OF NUMBER;

nums NumTab := NumTab(1, 2, 3, 4);

TYPE PNumTab IS TABLE OF PNum;

pnums PNumTab := PNumTab(PNum(1), PNum(2), PNum(3), PNum(4));

BEGIN

FORALL i IN pnums.FIRST..pnums.LAST

INSERT INTO partno VALUES(pnums(i));

FORALL i IN nums.FIRST..nums.LAST

DELETE FROM partno WHERE n = 2 \* nums(i);

FORALL i IN nums.FIRST..nums.LAST

INSERT INTO partno VALUES(100 + nums(i));

END;

**How FORALL Affects Rollbacks**

In a FORALL statement, if any execution of the SQL statement raises an unhandled exception, all database changes made during previous executions are rolled back. However, if a raised exception is caught and handled, changes are rolled back to an implicit savepoint marked before each execution of the SQL statement. Changes made during previous executions are *not* rolled back. For example, suppose you create a database table that stores department numbers and job titles, as follows:

CREATE TABLE emp2 (deptno NUMBER(2), job VARCHAR2(15));

Next, you insert some rows into the table, as follows:

INSERT INTO emp2 VALUES(10, 'Clerk');

INSERT INTO emp2 VALUES(10, 'Clerk');

INSERT INTO emp2 VALUES(20, 'Bookkeeper'); -- 10-char job title

INSERT INTO emp2 VALUES(30, 'Analyst');

INSERT INTO emp2 VALUES(30, 'Analyst');

Then, you try to append the 7-character string ' (temp)' to certain job titles using the following UPDATE statement:

DECLARE

TYPE NumList IS TABLE OF NUMBER;

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

BEGIN

FORALL j IN depts.FIRST..depts.LAST

UPDATE emp2 SET job = job || ' (temp)'

WHERE deptno = depts(j);

-- raises a "value too large" exception

EXCEPTION

WHEN OTHERS THEN

COMMIT;

END;

The SQL engine executes the UPDATE statement three times, once for each index number in the specified range, that is, once for depts(10), once for depts(20), and once for depts(30). The first execution succeeds, but the second execution fails because the string value 'Bookkeeper (temp)' is too large for the job column. In this case, only the second execution is rolled back.

When any execution of the SQL statement raises an exception, the FORALL statement halts. In our example, the second execution of the UPDATE statement raises an exception, so the third execution is never done.

**Counting Rows Affected by FORALL Iterations with the %BULK\_ROWCOUNT Attribute**

To process SQL data manipulation statements, the SQL engine opens an implicit cursor named SQL. This cursor's scalar attributes, %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT, return useful information about the most recently executed SQL data manipulation statement.

The SQL cursor has one composite attribute, %BULK\_ROWCOUNT, designed for use with the FORALL statement. This attribute has the semantics of an index-by table. Its *i*th element stores the number of rows processed by the *i*th execution of an INSERT, UPDATE or DELETE statement. If the *i*th execution affects no rows, %BULK\_ROWCOUNT(i) returns zero. An example follows:

DECLARE

TYPE NumList IS TABLE OF NUMBER;

depts NumList := NumList(10, 20, 50);

BEGIN

FORALL j IN depts.FIRST..depts.LAST

UPDATE emp SET sal = sal \* 1.10 WHERE deptno = depts(j);

-- Did the 3rd UPDATE statement affect any rows?

IF SQL%BULK\_ROWCOUNT(3) = 0 THEN ...

END;

The FORALL statement and %BULK\_ROWCOUNT attribute use the same subscripts. For example, if FORALL uses the range 5 .. 10, so does %BULK\_ROWCOUNT.

%BULK\_ROWCOUNT is usually equal to 1 for inserts, because a typical insert operation affects only a single row. But for the INSERT ... SELECT construct, %BULK\_ROWCOUNT might be greater than 1. For example, the FORALL statement below inserts an arbitrary number of rows for each iteration. After each iteration, %BULK\_ROWCOUNT returns the number of items inserted:

SET SERVEROUTPUT ON;

DECLARE

TYPE num\_tab IS TABLE OF NUMBER;

deptnums num\_tab;

BEGIN

SELECT deptno BULK COLLECT INTO deptnums FROM DEPT;

FORALL i IN 1..deptnums.COUNT

INSERT INTO emp\_by\_dept

SELECT empno, deptno FROM emp WHERE deptno =

deptnums(i);

FOR i IN 1..deptnums.COUNT LOOP

-- Count how many rows were inserted for each department; that is,

-- how many employees are in each department.

dbms\_output.put\_line('Dept '||deptnums(i)||': inserted '||

SQL%BULK\_ROWCOUNT(i)||' records');

END LOOP;

dbms\_output.put\_line('Total records inserted =' || SQL%ROWCOUNT);

END;

/

You can also use the scalar attributes %FOUND, %NOTFOUND, and %ROWCOUNT with bulk binds. For example, %ROWCOUNT returns the total number of rows processed by all executions of the SQL statement.

%FOUND and %NOTFOUND refer only to the last execution of the SQL statement. However, you can use %BULK\_ROWCOUNT to infer their values for individual executions. For example, when %BULK\_ROWCOUNT(i) is zero, %FOUND and %NOTFOUND are FALSE and TRUE, respectively.

**Handling FORALL Exceptions with the %BULK\_EXCEPTIONS Attribute**

PL/SQL provides a mechanism to handle exceptions raised during the execution of a FORALL statement. This mechanism enables a bulk-bind operation to save information about exceptions and continue processing.

To have a bulk bind complete despite errors, add the keywords SAVE EXCEPTIONS to your FORALL statement. The syntax follows:

FORALL index IN lower\_bound..upper\_bound SAVE EXCEPTIONS

{insert\_stmt | update\_stmt | delete\_stmt}

All exceptions raised during the execution are saved in the new cursor attribute %BULK\_EXCEPTIONS, which stores a collection of records. Each record has two fields. The first field, %BULK\_EXCEPTIONS(i).ERROR\_INDEX, holds the "iteration" of the FORALL statement during which the exception was raised. The second field, %BULK\_EXCEPTIONS(i).ERROR\_CODE, holds the corresponding Oracle error code.

The values stored by %BULK\_EXCEPTIONS always refer to the most recently executed FORALL statement. The number of exceptions is saved in the count attribute of %BULK\_EXCEPTIONS, that is, %BULK\_EXCEPTIONS.COUNT. Its subscripts range from 1 to COUNT.

If you omit the keywords SAVE EXCEPTIONS, execution of the FORALL statement stops when an exception is raised. In that case, SQL%BULK\_EXCEPTIONS.COUNT returns 1, and SQL%BULK\_EXCEPTIONS contains just one record. If no exception is raised during execution, SQL%BULK\_EXCEPTIONS.COUNT returns 0.

The following example shows how useful the cursor attribute %BULK\_EXCEPTIONS can be:

DECLARE

TYPE NumList IS TABLE OF NUMBER;

num\_tab NumList := NumList(10,0,11,12,30,0,20,199,2,0,9,1);

errors NUMBER;

dml\_errors EXCEPTION;

PRAGMA exception\_init(dml\_errors, -24381);

BEGIN

FORALL i IN num\_tab.FIRST..num\_tab.LAST SAVE EXCEPTIONS

DELETE FROM emp WHERE sal > 500000/num\_tab(i);

EXCEPTION

WHEN dml\_errors THEN

errors := SQL%BULK\_EXCEPTIONS.COUNT;

dbms\_output.put\_line('Number of errors is ' || errors);

FOR i IN 1..errors LOOP

dbms\_output.put\_line('Error ' || i || ' occurred during '||

'iteration ' || SQL%BULK\_EXCEPTIONS(i).ERROR\_INDEX);

dbms\_output.put\_line('Oracle error is ' ||

SQLERRM(-SQL%BULK\_EXCEPTIONS(i).ERROR\_CODE));

END LOOP;

END;

In this example, PL/SQL raised the predefined exception ZERO\_DIVIDE when i equaled 2, 6, 10. After the bulk-bind completed, SQL%BULK\_EXCEPTIONS.COUNT returned 3, and the contents of SQL%BULK\_EXCEPTIONS were (2,1476), (6,1476), and (10,1476). To get the Oracle error message (which includes the code), we negated the value of SQL%BULK\_EXCEPTIONS(i).ERROR\_CODE and passed the result to the error-reporting function SQLERRM, which expects a negative number. Here is the output:

Number of errors is 3

Error 1 occurred during iteration 2

Oracle error is ORA-01476: divisor is equal to zero

Error 2 occurred during iteration 6

Oracle error is ORA-01476: divisor is equal to zero

Error 3 occurred during iteration 10

Oracle error is ORA-01476: divisor is equal to zero

**Retrieving Query Results into Collections with the BULK COLLECT Clause**

The keywords BULK COLLECT tell the SQL engine to bulk-bind output collections before returning them to the PL/SQL engine. You can use these keywords in the SELECT INTO, FETCH INTO, and RETURNING INTO clauses. Here is the syntax:

... BULK COLLECT INTO collection\_name[, collection\_name] ...

The SQL engine bulk-binds all collections referenced in the INTO list. The corresponding columns can store scalar or composite values including objects. In the following example, the SQL engine loads the entire empno and ename database columns into nested tables before returning the tables to the PL/SQL engine:

DECLARE

TYPE NumTab IS TABLE OF emp.empno%TYPE;

TYPE NameTab IS TABLE OF emp.ename%TYPE;

enums NumTab; -- no need to initialize

names NameTab;

BEGIN

SELECT empno, ename BULK COLLECT INTO enums, names FROM emp;

...

END;

In the next example, the SQL engine loads all the values in an object column into a nested table before returning the table to the PL/SQL engine:

CREATE TYPE Coords AS OBJECT (x NUMBER, y NUMBER);

CREATE TABLE grid (num NUMBER, loc Coords);

INSERT INTO grid VALUES(10, Coords(1,2));

INSERT INTO grid VALUES(20, Coords(3,4));

DECLARE

TYPE CoordsTab IS TABLE OF Coords;

pairs CoordsTab;

BEGIN

SELECT loc BULK COLLECT INTO pairs FROM grid;

-- now pairs contains (1,2) and (3,4)

END;

The SQL engine initializes and extends collections for you. (However, it cannot extend varrays beyond their maximum size.) Then, starting at index 1, it inserts elements consecutively and overwrites any pre-existent elements.

The SQL engine bulk-binds entire database columns. So, if a table has 50,000 rows, the engine loads 50,000 column values into the target collection. However, you can use the pseudocolumn ROWNUM to limit the number of rows processed. In the following example, you limit the number of rows to 100:

DECLARE

TYPE SalList IS TABLE OF emp.sal%TYPE;

sals SalList;

BEGIN

SELECT sal BULK COLLECT INTO sals FROM emp

WHERE ROWNUM <= 100;

...

END;

**Examples of Bulk Fetching from a Cursor**

**Into One or More Collections**

You can bulk-fetch from a cursor into one or more collections:

DECLARE

TYPE NameList IS TABLE OF emp.ename%TYPE;

TYPE SalList IS TABLE OF emp.sal%TYPE;

CURSOR c1 IS SELECT ename, sal FROM emp WHERE sal > 1000;

names NameList;

sals SalList;

BEGIN

OPEN c1;

FETCH c1 BULK COLLECT INTO names, sals;

END;

**Into a Collection of Records**

You can bulk-fetch from a cursor into a collection of records:

DECLARE

TYPE DeptRecTab IS TABLE OF dept%ROWTYPE;

dept\_recs DeptRecTab;

CURSOR c1 IS

SELECT deptno, dname, loc FROM dept WHERE deptno > 10;

BEGIN

OPEN c1;

FETCH c1 BULK COLLECT INTO dept\_recs;

END;

**Limiting the Rows for a Bulk FETCH Operation with the LIMIT Clause**

The optional LIMIT clause, allowed only in bulk (not scalar) FETCH statements, lets you limit the number of rows fetched from the database. The syntax is

FETCH ... BULK COLLECT INTO ... [LIMIT rows];

where rows can be a literal, variable, or expression but must evaluate to a number. Otherwise, PL/SQL raises the predefined exception VALUE\_ERROR. If the number is not positive, PL/SQL raises INVALID\_NUMBER. If necessary, PL/SQL rounds the number to the nearest integer.

In the example below, with each iteration of the loop, the FETCH statement fetches ten rows (or less) into index-by table empnos. The previous values are overwritten.

DECLARE

TYPE NumTab IS TABLE OF NUMBER INDEX BY BINARY\_INTEGER;

CURSOR c1 IS SELECT empno FROM emp;

empnos NumTab;

rows NATURAL := 10;

BEGIN

OPEN c1;

LOOP

/\* The following statement fetches 10 rows (or less). \*/

FETCH c1 BULK COLLECT INTO empnos LIMIT rows;

EXIT WHEN c1%NOTFOUND;

...

END LOOP;

CLOSE c1;

END;

**Retrieving DML Results into a Collection with the RETURNING INTO Clause**

You can use the BULK COLLECT clause in the RETURNING INTO clause of an INSERT, UPDATE, or DELETE statement, as the following example shows:

DECLARE

TYPE NumList IS TABLE OF emp.empno%TYPE;

enums NumList;

BEGIN

DELETE FROM emp WHERE deptno = 20

RETURNING empno BULK COLLECT INTO enums;

-- if there were five employees in department 20,

-- then enums contains five employee numbers

END;

**Restrictions on BULK COLLECT**

The following restrictions apply to the BULK COLLECT clause:

* + You cannot bulk collect into an associative array that has a string type for the key.
  + You can use the BULK COLLECT clause only in server-side programs (not in client-side programs). Otherwise, you get the error *this feature is not supported in client-side programs*.
  + All targets in a BULK COLLECT INTO clause must be collections, as the following example shows:
  + DECLARE
  + TYPE NameList IS TABLE OF emp.ename%TYPE;
  + names NameList;
  + salary emp.sal%TYPE;
  + BEGIN
  + SELECT ename, sal BULK COLLECT INTO names, salary -- illegal target
  + FROM emp WHERE ROWNUM < 50;
  + ...
  + END;
  + Composite targets (such as objects) cannot be used in the RETURNING INTO clause. Otherwise, you get the error *unsupported feature with* *RETURNING* *clause*.
  + When implicit datatype conversions are needed, multiple composite targets cannot be used in the BULK COLLECT INTO clause.
  + When an implicit datatype conversion is needed, a collection of a composite target (such as a collection of objects) cannot be used in the BULK COLLECT INTO clause.

**Using FORALL and BULK COLLECT Together**

You can combine the BULK COLLECT clause with a FORALL statement, in which case, the SQL engine bulk-binds column values incrementally. In the following example, if collection depts has 3 elements, each of which causes 5 rows to be deleted, then collection enums has 15 elements when the statement completes:

FORALL j IN depts.FIRST..depts.LAST

DELETE FROM emp WHERE empno = depts(j)

RETURNING empno BULK COLLECT INTO enums;

The column values returned by each execution are added to the values returned previously. (With a FOR loop, the previous values are overwritten.)

You cannot use the SELECT ... BULK COLLECT statement in a FORALL statement. Otherwise, you get the error *implementation restriction: cannot use* *FORALL* *and* *BULK* *COLLECT* *INTO* *together in* *SELECT* *statements*.

**Using Host Arrays with Bulk Binds**

Client-side programs can use anonymous PL/SQL blocks to bulk-bind input and output host arrays. In fact, that is the most efficient way to pass collections to and from the database server.

Host arrays are declared in a host environment such as an OCI or Pro\*C program and must be prefixed with a colon to distinguish them from PL/SQL collections. In the example below, an input host array is used in a DELETE statement. At run time, the anonymous PL/SQL block is sent to the database server for execution.

DECLARE

...

BEGIN

-- assume that values were assigned to the host array

-- and host variables in the host environment

FORALL i IN :lower..:upper

DELETE FROM emp WHERE deptno = :depts(i);

...

END;

**What Is a Record?**

A *record* is a group of related data items stored in *fields*, each with its own name and datatype. Suppose you have various data about an employee such as name, salary, and hire date. These items are logically related but dissimilar in type. A record containing a field for each item lets you treat the data as a logical unit. Thus, records make it easier to organize and represent information.

The attribute %ROWTYPE lets you declare a record that represents a row in a database table. However, you cannot specify the datatypes of fields in the record or declare fields of your own. The datatype RECORD lifts those restrictions and lets you define your own records.

**Defining and Declaring Records**

To create records, you define a RECORD type, then declare records of that type. You can define RECORD types in the declarative part of any PL/SQL block, subprogram, or package using the syntax

TYPE type\_name IS RECORD (field\_declaration[,field\_declaration]...);

where field\_declaration stands for

field\_name field\_type [[NOT NULL] {:= | DEFAULT} expression]

and where type\_name is a type specifier used later to declare records, field\_type is any PL/SQL datatype except REF CURSOR, and expression yields a value of the same type as field\_type.

**Note:** Unlike VARRAY and (nested) TABLE types, RECORD types cannot be CREATEd and stored in the database.

You can use %TYPE and %ROWTYPE to specify field types. In the following example, you define a RECORD type named DeptRec:

DECLARE

TYPE DeptRec IS RECORD (

dept\_id dept.deptno%TYPE,

dept\_name VARCHAR2(14),

dept\_loc VARCHAR2(13));

BEGIN

...

END;

Notice that field declarations are like variable declarations. Each field has a unique name and specific datatype. So, the value of a record is actually a collection of values, each of some simpler type.

As the example below shows, PL/SQL lets you define records that contain objects, collections, and other records (called *nested* records). However, object types cannot have attributes of type RECORD.

DECLARE

TYPE TimeRec IS RECORD (

seconds SMALLINT,

minutes SMALLINT,

hours SMALLINT);

TYPE FlightRec IS RECORD (

flight\_no INTEGER,

plane\_id VARCHAR2(10),

captain Employee, -- declare object

passengers PassengerList, -- declare varray

depart\_time TimeRec, -- declare nested record

airport\_code VARCHAR2(10));

BEGIN

...

END;

The next example shows that you can specify a RECORD type in the RETURN clause of a function specification. That allows the function to return a user-defined record of the same type.

DECLARE

TYPE EmpRec IS RECORD (

emp\_id NUMBER(4)

last\_name VARCHAR2(10),

dept\_num NUMBER(2),

job\_title VARCHAR2(9),

salary NUMBER(7,2));

...

FUNCTION nth\_highest\_salary (n INTEGER) RETURN EmpRec IS ...

BEGIN

...

END;

**Declaring Records**

Once you define a RECORD type, you can declare records of that type, as the example below shows. The identifier item\_info represents an entire record.

DECLARE

TYPE StockItem IS RECORD (

item\_no INTEGER(3),

description VARCHAR2(50),

quantity INTEGER,

price REAL(7,2));

item\_info StockItem; -- declare record

BEGIN

...

END;

Like scalar variables, user-defined records can be declared as the formal parameters of procedures and functions. An example follows:

DECLARE

TYPE EmpRec IS RECORD (

emp\_id emp.empno%TYPE,

last\_name VARCHAR2(10),

job\_title VARCHAR2(9),

salary NUMBER(7,2));

...

PROCEDURE raise\_salary (emp\_info EmpRec);

BEGIN

...

END;

**Initializing Records**

The example below shows that you can initialize a record in its type definition. When you declare a record of type TimeRec, its three fields assume an initial value of zero.

DECLARE

TYPE TimeRec IS RECORD (

secs SMALLINT := 0,

mins SMALLINT := 0,

hrs SMALLINT := 0);

BEGIN

...

END;

The next example shows that you can impose the NOT NULL constraint on any field, and so prevent the assigning of nulls to that field. Fields declared as NOT NULL must be initialized.

DECLARE

TYPE StockItem IS RECORD (

item\_no INTEGER(3) NOT NULL := 999,

description VARCHAR2(50),

quantity INTEGER,

price REAL(7,2));

BEGIN

...

END;

**Referencing Records**

Unlike elements in a collection, which are accessed using subscripts, fields in a record are accessed by name. To reference an individual field, use dot notation and the following syntax:

record\_name.field\_name

For example, you reference field hire\_date in record emp\_info as follows:

emp\_info.hire\_date ...

When calling a function that returns a user-defined record, use the following syntax to reference fields in the record:

function\_name(parameter\_list).field\_name

For example, the following call to function nth\_highest\_sal references the field salary in record emp\_info:

DECLARE

TYPE EmpRec IS RECORD (

emp\_id NUMBER(4),

job\_title VARCHAR2(9),

salary NUMBER(7,2));

middle\_sal NUMBER(7,2);

FUNCTION nth\_highest\_sal (n INTEGER) RETURN EmpRec IS

emp\_info EmpRec;

BEGIN

...

RETURN emp\_info; -- return record

END;

BEGIN

middle\_sal := nth\_highest\_sal(10).salary; -- call function

...

END;

When calling a parameterless function, use the following syntax:

function\_name().field\_name -- note empty parameter list

To reference nested fields in a record returned by a function, use extended dot notation. The syntax follows:

function\_name(parameter\_list).field\_name.nested\_field\_name

For instance, the following call to function item references the nested field minutes in record item\_info:

DECLARE

TYPE TimeRec IS RECORD (minutes SMALLINT, hours SMALLINT);

TYPE AgendaItem IS RECORD (

priority INTEGER,

subject VARCHAR2(100),

duration TimeRec);

FUNCTION item (n INTEGER) RETURN AgendaItem IS

item\_info AgendaItem;

BEGIN

...

RETURN item\_info; -- return record

END;

BEGIN

...

IF item(3).duration.minutes > 30 THEN ... -- call function

END;

Also, use extended dot notation to reference the attributes of an object stored in a field, as the following example shows:

DECLARE

TYPE FlightRec IS RECORD (

flight\_no INTEGER,

plane\_id VARCHAR2(10),

captain Employee, -- declare object

passengers PassengerList, -- declare varray

depart\_time TimeRec, -- declare nested record

airport\_code VARCHAR2(10));

flight FlightRec;

BEGIN

...

IF flight.captain.name = 'H Rawlins' THEN ...

END;

**Assigning Null Values to Records**

To set all the fields in a record to null, simply assign to it an uninitialized record of the same type, as shown in the following example:

DECLARE

TYPE EmpRec IS RECORD (

emp\_id emp.empno%TYPE,

job\_title VARCHAR2(9),

salary NUMBER(7,2));

emp\_info EmpRec;

emp\_null EmpRec;

BEGIN

emp\_info.emp\_id := 7788;

emp\_info.job\_title := 'ANALYST';

emp\_info.salary := 3500;

emp\_info := emp\_null; -- nulls all fields in emp\_info

...

END;

**Assigning Records**

You can assign the value of an expression to a specific field in a record using the following syntax:

record\_name.field\_name := expression;

In the following example, you convert an employee name to upper case:

emp\_info.ename := UPPER(emp\_info.ename);

Instead of assigning values separately to each field in a record, you can assign values to all fields at once. This can be done in two ways. First, you can assign one user-defined record to another if they have the same datatype. Having fields that match exactly is not enough. Consider the following example:

DECLARE

TYPE DeptRec IS RECORD (

dept\_num NUMBER(2),

dept\_name VARCHAR2(14));

TYPE DeptItem IS RECORD (

dept\_num NUMBER(2),

dept\_name VARCHAR2(14));

dept1\_info DeptRec;

dept2\_info DeptItem;

BEGIN

...

dept1\_info := dept2\_info; -- illegal; different datatypes

END;

As the next example shows, you can assign a %ROWTYPE record to a user-defined record if their fields match in number and order, and corresponding fields have compatible datatypes:

DECLARE

TYPE DeptRec IS RECORD (

dept\_num NUMBER(2),

dept\_name VARCHAR2(14),

location VARCHAR2(13));

dept1\_info DeptRec;

dept2\_info dept%ROWTYPE;

BEGIN

SELECT \* INTO dept2\_info FROM dept WHERE deptno = 10;

dept1\_info := dept2\_info;

...

END;

Second, you can use the SELECT or FETCH statement to fetch column values into a record, as the example below shows. The columns in the select-list must appear in the same order as the fields in your record.

DECLARE

TYPE DeptRec IS RECORD (

dept\_num NUMBER(2),

dept\_name VARCHAR2(14),

location VARCHAR2(13));

dept\_info DeptRec;

BEGIN

SELECT \* INTO dept\_info FROM dept WHERE deptno = 20;

...

END;

However, you cannot assign a list of values to a record using an assignment statement. The following syntax is not allowed:

record\_name := (value1, value2, value3, ...); -- not allowed

The example below shows that you can assign one nested record to another if they have the same datatype. Such assignments are allowed even if the enclosing records have different datatypes.

DECLARE

TYPE TimeRec IS RECORD (mins SMALLINT, hrs SMALLINT);

TYPE MeetingRec IS RECORD (

day DATE,

time\_of TimeRec, -- nested record

room\_no INTEGER(4));

TYPE PartyRec IS RECORD (

day DATE,

time\_of TimeRec, -- nested record

place VARCHAR2(25));

seminar MeetingRec;

party PartyRec;

BEGIN

...

party.time\_of := seminar.time\_of;

END;

**Comparing Records**

Records cannot be tested for nullity, equality, or inequality. For instance, the following IF conditions are not allowed:

BEGIN

...

IF emp\_info IS NULL THEN ... -- illegal

IF dept2\_info > dept1\_info THEN ... -- illegal

END;

**Manipulating Records**

The datatype RECORD lets you collect information about the attributes of something. The information is easy to manipulate because you can refer to the collection as a whole. In the following example, you collect accounting figures from database tables assets and liabilities, then use ratio analysis to compare the performance of two subsidiary companies:

DECLARE

TYPE FiguresRec IS RECORD (cash REAL, notes REAL, ...);

sub1\_figs FiguresRec;

sub2\_figs FiguresRec;

FUNCTION acid\_test (figs FiguresRec) RETURN REAL IS ...

BEGIN

SELECT cash, notes, ... INTO sub1\_figs FROM assets, liabilities

WHERE assets.sub = 1 AND liabilities.sub = 1;

SELECT cash, notes, ... INTO sub2\_figs FROM assets, liabilities

WHERE assets.sub = 2 AND liabilities.sub = 2;

IF acid\_test(sub1\_figs) > acid\_test(sub2\_figs) THEN ...

...

END;

Notice how easy it is to pass the collected figures to the function acid\_test, which computes a financial ratio.

In SQL\*Plus, suppose you define object type Passenger, as follows:

SQL> CREATE TYPE Passenger AS OBJECT(

2 flight\_no NUMBER(3),

3 name VARCHAR2(20),

4 seat CHAR(5));

Next, you define VARRAY type PassengerList, which stores Passenger objects:

SQL> CREATE TYPE PassengerList AS VARRAY(300) OF Passenger;

Finally, you create relational table flights, which has a column of type PassengerList, as follows:

SQL> CREATE TABLE flights (

2 flight\_no NUMBER(3),

3 gate CHAR(5),

4 departure CHAR(15),

5 arrival CHAR(15),

6 passengers PassengerList);

Each item in column passengers is a varray that will store the passenger list for a given flight. Now, you can populate database table flights, as follows:

BEGIN

INSERT INTO flights

VALUES(109, '80', 'DFW 6:35PM', 'HOU 7:40PM',

PassengerList(Passenger(109, 'Paula Trusdale', '13C'),

Passenger(109, 'Louis Jemenez', '22F'),

Passenger(109, 'Joseph Braun', '11B'), ...));

INSERT INTO flights

VALUES(114, '12B', 'SFO 9:45AM', 'LAX 12:10PM',

PassengerList(Passenger(114, 'Earl Benton', '23A'),

Passenger(114, 'Alma Breckenridge', '10E'),

Passenger(114, 'Mary Rizutto', '11C'), ...));

INSERT INTO flights

VALUES(27, '34', 'JFK 7:05AM', 'MIA 9:55AM',

PassengerList(Passenger(27, 'Raymond Kiley', '34D'),

Passenger(27, 'Beth Steinberg', '3A'),

Passenger(27, 'Jean Lafevre', '19C'), ...));

END;

In the example below, you fetch rows from database table flights into record flight\_info. That way, you can treat all the information about a flight, including its passenger list, as a logical unit.

DECLARE

TYPE FlightRec IS RECORD (

flight\_no NUMBER(3),

gate CHAR(5),

departure CHAR(15),

arrival CHAR(15),

passengers PassengerList);

flight\_info FlightRec;

CURSOR c1 IS SELECT \* FROM flights;

seat\_not\_available EXCEPTION;

BEGIN

OPEN c1;

LOOP

FETCH c1 INTO flight\_info;

EXIT WHEN c1%NOTFOUND;

FOR i IN 1..flight\_info.passengers.LAST LOOP

IF flight\_info.passengers(i).seat = 'NA' THEN

dbms\_output.put\_line(flight\_info.passengers(i).name);

RAISE seat\_not\_available;

END IF;

...

END LOOP;

END LOOP;

CLOSE c1;

EXCEPTION

WHEN seat\_not\_available THEN

...

END;

**Inserting PL/SQL Records into the Database**

A PL/SQL-only extension of the INSERT statement lets you insert records into database rows using a single variable of type RECORD or %ROWTYPE instead of a list of fields. That makes your code more readable and maintainable.

The number of fields in the record must equal the number of columns listed in the INTO clause, and corresponding fields and columns must have compatible datatypes. To make sure the record is compatible with the table, you might find it most convenient to declare the variable as the type *table\_name*%ROWTYPE.

**Inserting a PL/SQL Record Using %ROWTYPE: Example**

This example declares a record variable using a %ROWTYPE qualifier. You can insert this variable without specifying a column list. The %ROWTYPE declaration ensures that the record attributes have exactly the same names and types as the table columns.

DECLARE

dept\_info dept%ROWTYPE;

BEGIN

-- deptno, dname, and loc are the table columns.

-- The record picks up these names from the %ROWTYPE.

dept\_info.deptno := 70;

dept\_info.dname := 'PERSONNEL';

dept\_info.loc := 'DALLAS';

-- Using the %ROWTYPE means we can leave out the column list

-- (deptno, dname, loc) from the INSERT statement.

INSERT INTO dept VALUES dept\_info;

END;

**Updating the Database with PL/SQL Record Values**

A PL/SQL-only extension of the UPDATE statement lets you update database rows using a single variable of type RECORD or %ROWTYPE instead of a list of fields.

The number of fields in the record must equal the number of columns listed in the SET clause, and corresponding fields and columns must have compatible datatypes.

**Updating a Row Using a Record: Example**

You can use the keyword ROW to represent an entire row:

DECLARE

dept\_info dept%ROWTYPE;

BEGIN

dept\_info.deptno := 30;

dept\_info.dname := 'MARKETING';

dept\_info.loc := 'ATLANTA';

-- The row will have values for the filled-in columns, and null

-- for any other columns.

UPDATE dept SET ROW = dept\_info WHERE deptno = 30;

END;

The keyword ROW is allowed only on the left side of a SET clause.

**SET ROW Not Allowed with Subquery: Example**

You cannot use ROW with a subquery. For example, the following UPDATE statement is not allowed:

UPDATE emp SET ROW = (SELECT \* FROM mgrs); -- not allowed

**Updating a Row Using a Record Containing an Object: Example**

Records containing object types are allowed:

CREATE TYPE Worker AS OBJECT (name VARCHAR2(25), dept VARCHAR2(15));

/

CREATE TABLE teams (team\_no NUMBER, team\_member Worker);

DECLARE

team\_rec teams%ROWTYPE;

BEGIN

team\_rec.team\_no := 5;

team\_rec.team\_member := Worker('Paul Ocker', 'Accounting');

UPDATE teams SET ROW = team\_rec;

END;

/

**Updating a Row Using a Record Containing a Collection: Example**

The record can also contain collections:

CREATE TYPE Worker AS OBJECT (name VARCHAR2(25), dept VARCHAR2(15));

/

CREATE TYPE Roster AS TABLE OF Worker;

/

CREATE TABLE teams (team\_no NUMBER, members Roster)

NESTED TABLE members STORE AS teams\_store;

INSERT INTO teams VALUES (1, Roster(

Worker('Paul Ocker', 'Accounting'),

Worker('Gail Chan', 'Sales')

Worker('Marie Bello', 'Operations')

Worker('Alan Conwright', 'Research')));

DECLARE

team\_rec teams%ROWTYPE;

BEGIN

team\_rec.team\_no := 3;

team\_rec.members := Roster(

Worker('William Bliss', 'Sales'),

Worker('Ana Lopez', 'Sales')

Worker('Bridget Towner', 'Operations')

Worker('Ajay Singh', 'Accounting'));

UPDATE teams SET ROW = team\_rec;

END;

/

**Using the RETURNING Clause with a Record: Example**

The INSERT, UPDATE, and DELETE statements can include a RETURNING clause, which returns column values from the affected row into a PL/SQL record variable. This eliminates the need to SELECT the row after an insert or update, or before a delete. You can use this clause only when operating on exactly one row.

In the following example, you update the salary of an employee and, at the same time, retrieve the employee's name, job title, and new salary into a record variable:

DECLARE

TYPE EmpRec IS RECORD (

emp\_name VARCHAR2(10),

job\_title VARCHAR2(9),

salary NUMBER(7,2));

emp\_info EmpRec;

emp\_id NUMBER(4);

BEGIN

emp\_id := 7782;

UPDATE emp SET sal = sal \* 1.1

WHERE empno = emp\_id

RETURNING ename, job, sal INTO emp\_info;

END;

**Restrictions on Record Inserts/Updates**

Currently, the following restrictions apply to record inserts/updates:

* + Record variables are allowed only in the following places:
    - On the right side of the SET clause in an UPDATE statement
    - In the VALUES clause of an INSERT statement
    - In the INTO subclause of a RETURNING clause

Record variables are not allowed in a SELECT list, WHERE clause, GROUP BY clause, or ORDER BY clause.

* + The keyword ROW is allowed only on the left side of a SET clause. Also, you cannot use ROW with a subquery.
  + In an UPDATE statement, only one SET clause is allowed if ROW is used.
  + If the VALUES clause of an INSERT statement contains a record variable, no other variable or value is allowed in the clause.
  + If the INTO subclause of a RETURNING clause contains a record variable, no other variable or value is allowed in the subclause.
  + The following are *not* supported:
    - Nested record types
    - Functions that return a record
    - Record inserts/updates using the EXECUTE IMMEDIATE statement.

**Querying Data into Collections of Records**

PL/SQL binding operations fall into three categories:

* + **define** Refers to database values retrieved by a SELECT or FETCH statement into PL/SQL variables or host variables.
  + **in-bind** Refers to database values inserted by an INSERT statement or modified by an UPDATE statement.
  + **out-bind** Refers to database values returned by the RETURNING clause of an INSERT, UPDATE, or DELETE statement into PL/SQL variables or host variables.

PL/SQL supports the bulk binding of collections of records in DML statements. Specifically, a define or out-bind variable can be a collection of records, and in-bind values can be stored in a collection of records. The syntax follows:

SELECT select\_items BULK COLLECT INTO record\_variable\_name

FROM rest\_of\_select\_stmt

FETCH { cursor\_name

| cursor\_variable\_name

| :host\_cursor\_variable\_name}

BULK COLLECT INTO record\_variable\_name

[LIMIT numeric\_expression];

FORALL index IN lower\_bound..upper\_bound

INSERT INTO { table\_reference

| THE\_subquery} [{column\_name[, column\_name]...}]

VALUES (record\_variable\_name(index)) rest\_of\_insert\_stmt

FORALL index IN lower\_bound..upper\_bound

UPDATE {table\_reference | THE\_subquery} [alias]

SET (column\_name[, column\_name]...) = record\_variable\_name(index)

rest\_of\_update\_stmt

RETURNING row\_expression[, row\_expression]...

BULK COLLECT INTO record\_variable\_name;

In each statement and clause above, the record variable stores a collection of records. The number of fields in the record must equal the number of items in the SELECT list, the number of columns in the INSERT INTO clause, the number of columns in the UPDATE ... SET clause, or the number of row expressions in the RETURNING clause, respectively. Corresponding fields and columns must have compatible datatypes. Here are several examples:

CREATE TABLE tab1 (col1 NUMBER, col2 VARCHAR2(20));

/

CREATE TABLE tab2 (col1 NUMBER, col2 VARCHAR2(20));

/

DECLARE

TYPE RecTabTyp IS TABLE OF tab1%ROWTYPE

INDEX BY BINARY\_INTEGER;

TYPE NumTabTyp IS TABLE OF NUMBER

INDEX BY BINARY\_INTEGER;

TYPE CharTabTyp IS TABLE OF VARCHAR2(20)

INDEX BY BINARY\_INTEGER;

CURSOR c1 IS SELECT col1, col2 FROM tab2;

rec\_tab RecTabTyp;

num\_tab NumTabTyp := NumTabTyp(2,5,8,9);

char\_tab CharTabTyp := CharTabTyp('Tim', 'Jon', 'Beth', 'Jenny');

BEGIN

FORALL i IN 1..4

INSERT INTO tab1 VALUES(num\_tab(i), char\_tab(i));

SELECT col1, col2 BULK COLLECT INTO rec\_tab FROM tab1

WHERE col1 < 9;

FORALL i IN rec\_tab.FIRST..rec\_tab.LAST

INSERT INTO tab2 VALUES rec\_tab(i);

FOR i IN rec\_tab.FIRST..rec\_tab.LAST LOOP

rec\_tab(i).col1 := rec\_tab(i).col1 + 100;

END LOOP;

FORALL i IN rec\_tab.FIRST..rec\_tab.LAST

UPDATE tab1 SET (col1, col2) = rec\_tab(i) WHERE col1 < 8;

OPEN c1;

FETCH c1 BULK COLLECT INTO rec\_tab;

CLOSE c1;

END;

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | [Go to previous page](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/04_struc.htm) | [Go to next page](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/06_ora.htm) | | [[Oracle](http://docs.oracle.com/cd/B10500_01/dcommon/html/cpyr.htm) Copyright © 1996, 2002 Oracle Corporation.](http://docs.oracle.com/cd/B10500_01/dcommon/html/cpyr.htm) All Rights Reserved. | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | [[Go To Documentation Library](http://docs.oracle.com/cd/B10500_01/index.htm) Home](http://docs.oracle.com/cd/B10500_01/index.htm) | [[Go To Product List](http://docs.oracle.com/cd/B10500_01/nav/docindex.htm) Book List](http://docs.oracle.com/cd/B10500_01/nav/docindex.htm) | [[Go To Table Of Contents](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/toc.htm) Contents](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/toc.htm) | [[Go To Index](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/index.htm) Index](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/index.htm) | [[http://docs.oracle.com/cd/B10500_01/dcommon/gifs/mix.gif](http://docs.oracle.com/cd/B10500_01/mix.920/a96625/toc.htm) Master Index](http://docs.oracle.com/cd/B10500_01/mix.920/a96625/toc.htm) | [[http://docs.oracle.com/cd/B10500_01/dcommon/gifs/feedback.gif](http://docs.oracle.com/cd/B10500_01/dcommon/html/feedback.htm) Feedback](http://docs.oracle.com/cd/B10500_01/dcommon/html/feedback.htm) | |