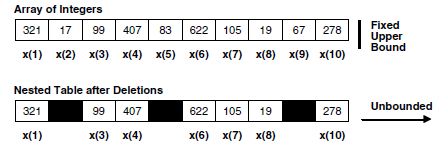
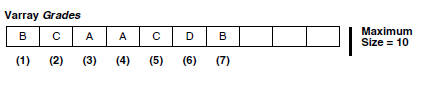
**Collections and Records in PL/SQL**

Collections and records are composite types that have internal components that can be manipulated individually, such as the elements of an array, record, or table.

**PL/SQL Collections**

A collection is an ordered group of elements, all of the same type. It is a general concept that encompasses lists, arrays, and other data types used in classic programming algorithms. Each element is addressed by a unique subscript.  
PL/SQL offers these collection types:

* **Associative arrays**, also known as index-by tables, let you look up elements using arbitrary numbers and strings for subscript values. These 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](https://intellipaat.com/blog/tutorial/sql-tutorial/introduction-to-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.

[](https://cdn.intellipaat.com/wp-content/uploads/2015/09/Array-versus-Nested-Table.png)       [](https://cdn.intellipaat.com/wp-content/uploads/2015/09/varray-of-size-10.png)

**PL/SQL Records**

**A record** is a group of related data items stored in fields, each with its own name and datatype. You can think of a record as a variable that can hold a table row or some columns from a table row. The fields correspond to table columns.  
Records are composed of a group of fields, similar to the columns in a row. The **%ROWTYPE** attribute lets you declare a **PL/SQL record** that represents a row in a database table, without listing all the columns. Your code keeps working even after columns are added to the table. If you want to represent a subset of columns in a table, or columns from different tables, you can define a view or declare a cursor to select the right columns and do any necessary joins, and then apply **%ROWTYPE** to the view or cursor.

***Want to know more about SQL? Read this extensive***[***SQL Tutorial***](https://intellipaat.com/blog/tutorial/sql-tutorial/)***and enhance your knowledge!***

**Defining Collection Types and Declaring Collection Variables**  
To create collections, you define a collection type, and then declare variables of that type. Collections follow the same scoping and instantiation rules as other types and variables. Collections are instantiated when you enter a 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.  
You can define TABLE and VARRAY types in the declarative part of any PL/SQL block, subprogram, or package using a TYPE definition.  
For nested tables and varrays declared within PL/SQL, the element type of the table or varray can be any PL/SQL datatype except REF CURSOR.  
When defining a VARRAY type, you must specify its maximum size with a positive integer. In the following example, you define a type that stores up to 366 dates:

DECLARE

TYPE Calendar IS VARRAY(366) OF DATE;

**Declaring PL/SQL Collection Variables**  
After defining a collection type, you declare variables of that type. You use the new type name in the declaration, the same as with predefined types such as NUMBER.

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

DECLARE

TYPE nested\_type IS TABLE OF VARCHAR2(30);

TYPE varray\_type IS VARRAY(5) OF INTEGER;

TYPE assoc\_array\_num\_type IS TABLE OF NUMBER INDEX BY PLS\_INTEGER;

TYPE assoc\_array\_str\_type IS TABLE OF VARCHAR2(32) INDEX BY PLS\_INTEGER;

TYPE assoc\_array\_str\_type2 IS TABLE OF VARCHAR2(32) INDEX BY VARCHAR2(64);

v1 nested\_type;

v2 varray\_type;

v3 assoc\_array\_num\_type;

v4 assoc\_array\_str\_type;

v5 assoc\_array\_str\_type2;

BEGIN

-- an arbitrary number of strings can be inserted v1

v1 := nested\_type('Shipping','Sales','Finance','Payroll');

v2 := varray\_type(1, 2, 3, 4, 5); -- Up to 5 integers

v3(99) := 10; -- Just start assigning to elements

v3(7) := 100; -- Subscripts can be any integer values

v4(42) := 'Smith'; -- Just start assigning to elements

v4(54) := 'Jones'; -- Subscripts can be any integer values

v5('Canada') := 'North America'; -- Just start assigning to elements

v5('Greece') := 'Europe'; -- Subscripts can be string values

END;

/

**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**

DECLARE

TYPE dnames\_tab IS TABLE OF VARCHAR2(30);

dept\_names dnames\_tab;

BEGIN

dept\_names := dnames\_tab('Shipping','Sales','Finance','Payroll');

END;

/

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

**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 .. 2147483647 (the upper limit of PLS\_INTEGER).
* For varrays, 1 .. size\_limit, where you specify the limit in the declaration (not to exceed 2147483647).
* For associative arrays with a numeric key, -2147483648 to 2147483647.
* 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**

DECLARE

TYPE Roster IS TABLE OF VARCHAR2(15);

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

PROCEDURE verify\_name(the\_name VARCHAR2) IS

BEGIN

DBMS\_OUTPUT.PUT\_LINE(the\_name);

END;

BEGIN

FOR i IN names.FIRST .. names.LAST

LOOP

IF names(i) = 'J Hamil' THEN

DBMS\_OUTPUT.PUT\_LINE(names(i)); -- reference to nested table element

END IF;

END LOOP;

verify\_name(names(3)); -- procedure call with reference to element

END;

/

**Assigning Collections**

One collection can be assigned to another by an [INSERT](https://intellipaat.com/blog/tutorial/sql-tutorial/insert-statement/), [UPDATE](https://intellipaat.com/blog/tutorial/sql-tutorial/update-query/), FETCH, or [SELECT](https://intellipaat.com/blog/tutorial/sql-tutorial/select-statement/" \t "_blank)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.  
You can use operators such as SET, MULTISET UNION, MULTISET INTERSECT, and MULTISET EXCEPT to transform nested tables as part of an assignment statement.

**Comparing Collections**  
You can check whether a collection is null. Comparisons such as greater than, less than, and so on are not allowed. This restriction also applies to implicit comparisons.  
For example, collections cannot appear in a [select DISTINCT](https://intellipaat.com/blog/tutorial/sql-tutorial/distinct/), GROUP BY, or [ORDER BY](https://intellipaat.com/blog/tutorial/sql-tutorial/order-by-group-by/) list.

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

DECLARE

TYPE emp\_name\_rec is RECORD (

firstname employees.first\_name%TYPE,

lastname employees.last\_name%TYPE,

hiredate employees.hire\_date%TYPE

);

TYPE staff IS TABLE OF emp\_name\_rec;

members staff;

BEGIN

-- Condition yields TRUE because we have not used a constructor.

IF members IS NULL THEN

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

ELSE

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

END IF;

END;

**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.

***Visit our***[***Community***](https://intellipaat.com/community/)***to get answers to all your queries!***

**Collection Methods**

Collection methods make collections easier to use and make your applications easier to maintain. These methods include COUNT, DELETE, EXISTS, EXTEND, FIRST, LAST, LIMIT, NEXT, PRIOR, and TRIM.  
A collection method is a built-in function or procedure that operates on collections and is called using dot notation. The following applies to collection methods:  
■ Collection methods cannot be called from SQL statements.  
■ 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 string 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.

**1. EXISTS Method –**EXISTS(n) returns TRUE if the nth element in a collection exists. Otherwise, EXISTS(n) returns FALSE.

**2. COUNT Method –**COUNT returns the number of elements that a collection currently contains.

**3. LIMIT Method –**For nested tables and associative arrays, which have no declared size, LIMIT returns NULL. For varrays, LIMIT returns the maximum number of elements that a varray can contain.

**4. FIRST and LAST Methods –**FIRST and LAST return the first and last (smallest and largest) index numbers in a collection that uses integer subscripts.

**5. 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. If n has no successor, NEXT(n)returns NULL.

**6. EXTEND Method –**To increase the size of a nested table or varray, use EXTEND.  
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 ith element to a collection.

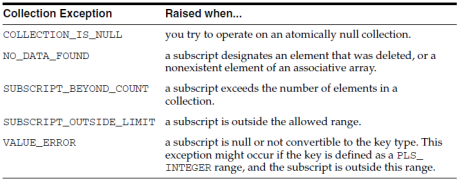
**7. TRIM Method –**  
This procedure has two forms:  
■ TRIM removes one element from the end of a collection.  
■ TRIM(n) removes n elements from the end of a collection.  
If you want to remove all elements, use [DELETE](https://intellipaat.com/blog/tutorial/sql-tutorial/delete-query/) without parameters.

**8. DELETE Method –**  
This procedure has various forms:

* DELETE with no parameters removes all elements from a collection, setting COUNT to 0.
* DELETE(n) removes the nth 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

***Go for this in-depth job-oriented***[***PL/SQL Training***](https://intellipaat.com/pl-sql-online-training/)***Course now!***

**Avoiding Collection Exceptions**

**[](https://cdn.intellipaat.com/wp-content/uploads/2015/09/avoiding-collection-exceptions.png)**

**Defining and Declaring Records**

To create records, you define a RECORD type, then declare records of that type. You can also create or find a table, view, or PL/SQL cursor with the values you want, and use the %ROWTYPE attribute to create a matching record.  
You can define RECORD types in the declarative part of any PL/SQL block, Pl/SQL subprogram, or PL/SQL package. When you define your own RECORD type, you can specify a NOT NULL constraint on fields, or give them default values.

**Example: Declaring and Initializing a Simple Record Type**

DECLARE

TYPE DeptRecTyp IS RECORD (

deptid NUMBER(4) NOT NULL := 99,

dname departments.department\_name%TYPE,

loc departments.location\_id%TYPE,

region regions%ROWTYPE );

dept\_rec DeptRecTyp;

BEGIN

dept\_rec.dname := 'PURCHASING';

END;

/

**Assigning Values to Records**  
To set all the fields in a record to default values, assign to it an uninitialized record of the same type.

**Example: Assigning Default Values to a Record**

DECLARE

TYPE RecordTyp IS RECORD (field1 NUMBER,

field2 VARCHAR2(32) DEFAULT 'something');

rec1 RecordTyp;

rec2 RecordTyp;

BEGIN

-- At first, rec1 has the values we assign.

rec1.field1 := 100; rec1.field2 := 'something else';

-- Assigning an empty record to rec1 resets fields to their default values.

-- Field1 is NULL and field2 is 'something' due to the DEFAULT clause

rec1 := rec2;

DBMS\_OUTPUT.PUT\_LINE('Field1 = ' || NVL(TO\_CHAR(rec1.field1),'<NULL>') || ',

field2 = ' || rec1.field2);

END;

/

You can assign a value to a field in a record using an assignment statement with dot notation:

emp\_info.last\_name := 'Fields';

**Comparing Records**

Records cannot be tested for nullity or compared for equality, or inequality. If you want to make such comparisons, write your own function that accepts two records as parameters and does the appropriate checks or comparisons on the corresponding fields.

***Check out the top***[***SQL Interview Questions***](https://intellipaat.com/blog/interview-question/pl-sql-interview-questions/)***to learn what is expected from SQL professionals!***

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

A PL/SQL-only extension of the[INSERT](https://intellipaat.com/blog/tutorial/sql-tutorial/insert-statement/) statement lets you insert records into database rows, using a single variable of type RECORD or %ROWTYPE in the VALUES clause instead of a list of fields. That makes your code more readable and maintainable.  
If you issue the INSERT through the FORALL statement, you can insert values from an entire collection of records. 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 data types. 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.

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

DECLARE

dept\_info departments%ROWTYPE;

BEGIN

-- department\_id, department\_name, and location\_id are the table columns

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

dept\_info.department\_id := 300;

dept\_info.department\_name := 'Personnel';

dept\_info.location\_id := 1700;

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

-- (department\_id, department\_name, and location\_id) from the INSERT statement

INSERT INTO departments VALUES dept\_info;

END;

/

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

A PL/SQL-only extension of the[UPDATE](https://intellipaat.com/blog/tutorial/sql-tutorial/update-query/) statement lets you update database rows using a single variable of type RECORD or %ROWTYPE on the right side of the SET clause, instead of a list of fields.

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

DECLARE

dept\_info departments%ROWTYPE;

BEGIN

-- department\_id, department\_name, and location\_id are the table columns

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

dept\_info.department\_id := 300;

dept\_info.department\_name := 'Personnel';

dept\_info.location\_id := 1700;

-- The fields of a %ROWTYPE can completely replace the table columns

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

-- for any other columns

UPDATE departments SET ROW = dept\_info WHERE department\_id = 300;

END;

/

**Querying Data into Collections of Records**

You can use the BULK COLLECT clause with a [SELECT](https://intellipaat.com/blog/tutorial/sql-tutorial/select-statement/) INTO or FETCH statement to retrieve a set of rows into a collection of records.