5 Working with Composite Data Types

Composite data types: 5

- plsql lets you define two kinds of composite data types: collection and record
- a composite data type stores values that have internal components. You can pass entire composite varibles to subprograms as parameters, and you can access internal components of composite variables individually
- internal components can be either scalar or composite. You can use scalar components wherever you can use scalar variables. You can use composite components wherever you can use composite variables of the same type
- in a **collection**, the internal compnenents always have the same data type, and are called **elements**.
    - You can access each element of a collection variable by its unique index, with this syntax: `variable_name(index)`
    - to create a collection variable, you either define a collection type and then create a variable of that type or use `%TYPE`
- in a **record**, the internal components can have different data types, and are called **fields**
    - you can access each field of a record variable by its name, with this syntax: `variable_name.field_name`
    - to create a record variable, you either define a `RECORD` type and then create a variable of that type or use `%ROWTYPE or %TYPE`
- you can create a collection of records, and a record that contains collections

✅Create user-defined PL/SQL records

Record variables: 5.12 - 5.12.3 Working with records: 5.13 - 5.17

Record variables

- you can create a record variable in any of these ways:
    - define a `RECORD` type and then declare a variable of that type
    - use a `%ROWTYPE` to declare a record variable that represents either a full or partial row of a database table or view
    - use `%TYPE` to declare a record variable of the same type as a previously declared record variable
- syntax record type definition:

```
TYPE record_type IS RECORD (
    field datatype [[NOT NULL] {:= | DEFAULT} expression]
    , ...
);
```

- syntax record variable declaration: `record_1 {record_type | rowtype_attribute | record_2 %TYPE} ;`

- Initial values of record variables
    - for a record variable of a `RECORD` type, the initial value of each field is `NULL` unless you specify a different initial value for it when you define the type
    - for a record variable declared with `%ROWTYPE or %TYPE`, the initial value of each field is `NULL`. The variable does not inherit the initial value of the referenced item
- Declaring record constants
    - when declaring a record constant, you must create a function that populates the record with its initial value and then invoke the function in the constant declaration e.g. `rec1 CONSTANT my_rec_type := init_my_rec()`, where `init_my_rec()` returns an initialized record of type `my_rec_type`
- Record types
    - a `RECORD` type defined in a plsql block is a **local type**. It is available only in the block and is stored in the database only if the block is in a standalone or package subprogram
    - a `RECORD` type defined in a package specification is a **public item**. You can reference it from outside the package by qualifying it with the package name (package_name.type_name). It is stored in the database until you drop the package with the `DROP PACKAGE` statement
    - you cannot create a `RECORD` type at schema level. Therefore, a `RECORD` type cannot be an ADT attribute data type
    - to define a `RECORD` type, specify its name and define its fields. To define a field, specify its name and data type. By default the initial value of a field is `NULL`. You can specify the `NOT NULL` constraint for a field, in which case you must also specify a non-NULL initial value. Without the `NOT NULL` constraint, a non-NULL initial value is optional
    - a `RECORD` type defined in a package specification is incompatible with an identically defined local `RECORD` type

Assigning values to record variables

- a record variable means either a record variable or a record component of a composite variable
- to any record variable, you can assign a value to each field individually
- you can assign values using qualified expressions
- in some cases you can assign the value of one record variable to another record variable
- if a record variable represents a full or partial row of a database table or view, you can assign the represented row to the record variable
    - using `SELECT INTO`: `SELECT select_list INTO record_variable_name FROM table_or_view_name;` for each column in select_list, the record variable must have a corresponding, type-compatible field. The columns in select_list must appear in the same order as the record fields
    - using `FETCH`: `FETCH cursor INTO record_variable_name` a cursor is associated with a query. For every column that the query selects, the record variable must have a corresponding, type-compatible field. The cursor must be either an explicit cursor or a strong cursor variable
    - using the optional `RETURNING INTO` clause of the sql `INSERT, UPDATE and DELETE` statements that can return the affected row in a plsql record variable
- you can assign the value of one record variable to another record variable only in these cases:
    - the two variables have the same `RECORD` type

    - the target variable is declared with a `RECORD` type, the source variable is declared with `%ROWTYPE`, their fields match in number and order, and corresponding fields have the same data type
- assigning NULL to a record variable
    - assigning the value `NULL` to a record variable assigns the value `NULL` to each of its fields
    - this assignment is recursive; that is, if a field is a record, then its fields are also assigned the value `NULL`

Record comparisons

- records cannot be tested natively for nullity, equality, or inequality
- these `BOOLEAN` expressions are illegal:
    - `my_record IS NULL`
    - `my_record_1 = my_record_2`
    - `my_record_1 > my_record_2`
- you must write your own functions to implement such tests

Inserting records into tables

- the plsql extension to the sql `INSERT` statement lets you insert a record into a table `INSERT INTO table VALUES record`
- the record must represent a row of the table
- to efficiently insert a collection of records into a table, put the `INSERT` statement inside a `FORALL` statement

Updating rows with records

- the plsql extension to the sql `UPDATE` statement lets you update one or more table rows with a record: `UPDATE table SET ROW = record`
- the record must represent a row of the table
- to efficiently updat a set of rows with a collection of records, put the `UPDATE` statement inside a `FORALL` statement

Restrictions on record inserts and updates

- record variables are allowed only in these places:

    - on the right side of the `SET` clause in an `UPDATE` statement
    - in the `VALUES` clause of an `INSERT` statement
    - in the `INTO` clause 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
- these are not supported:
    - nested `RECORD` types
    - functions that return a `RECORD` type
    - record inserts and updates using the `EXECUTE IMMEDIATE` statement

✅Create a record with the %ROWTYPE attribute

Declaring records using %ROWTYPE: 5.12.4

See Chapter 1 1_declaring_variables.md#rowtype

✅Create an INDEX BY table and INDEX BY table of records

Collection types: 5.1 Associative arrays: 5.2

Collection types

- plsql has three collection types: associative array, `VARRAY` (variable size array), and nested table
- Associative array (or index-by table)
    - **number of elements:** unspecified
    - **index type:** string or `PLS_INTEGER`
    - **dense or sparse:** either
    - **uninitialized status:** empty
    - **where defined:** in plsql block or package
    - **can be ADT attribute data type:** no
- `VARRAY`
    - **number of elements:** specified
    - **index type:** integer
    - **dense or sparse:** always dense
    - **uninitialized status:** null
    - **where defined:** in plsql block or package or at schema level
    - **can be ADT attribute data type:** only if defined at schema level
- Nested table
    - **number of elements:** unspecified
    - **index type:** integer
    - **dense or sparse:** starts dense, can become sparse
    - **uninitialized status:** null
    - **where defined:** in plsql block or package or at schema level
    - **can be ADT attribute data type:** only if defined at schema level
- Number of elements

    - if the number of elements is specified, it is the maximum number of elements in the collection
    - if the number of elements is unspecified, the maximum number of elements in the collection is the upper limit of the index type

- Dense or sparse

    - a **dense collection** has no gaps between elements. Every element between the first and last element is defined and has a value (the value can be `NULL` unless the element has a `NOT NULL` constraint)
    - a **sparse collection** has gaps between elements

- Uninitialized status

    - an **empty collection** exists but has no elements. To add elements to an empty collection, invoke the `EXTEND` method
    - a **null collection** (also called an **atomically null collection**) does not exist. To change a null collection to an existing collection, you must initialize it, either by making it empty or by assigning a non-NULL value to it. You cannot use the `EXTEND` method to initialize a null collection

- Where defined

    - a collection type defined in a plsql block is a **local type**. It is available only in the block, and is stored in the database only if the block is in a statndalone or package subprogram
    - a collection type defined in a package specification is a **public item**. You can reference it from outside the package by qualifying it with the package name. It is stored in the database until you drop the package
    - a collection type defined at schema level is a **standalone type**. You create it with the `CREATE TYPE` statement. It is stored in the database until you drop it with the `DROP TYPE` statement

- Can be ADT attribute data type

    - to be an ADT attribute data type, a collection type must be a standalone collection type

- Translating non-plsql composite data types to plsql composite types

    - associative array <=> hash table, unordered table
    - nested table <=> set, bag
    - varray <=> array

- You can create a collection variable in either of these ways:

    - define a collection type and then declare a variable of that type
    - use `%TYPE` to declare a collection variable of the same type as a previously declared collection variable

- with `CREATE TYPE` statement, you can create nested table types and varray types, but not associative array types

Associative arrays

- an **associative array** (formerly called **PL/SQL table** or **index-by table**) is a set of key-value pairs

- type def syntax:

```
TYPE type IS TABLE OF datatype[(length)] [NOT NULL] INDEX BY
{PLS_INTEGER | BINARY_INTEGER | {VARCHAR2|VARCHAR|STRING} (v_size) | LONG |
type_attr | rowtype_attr};
```

- datatype can be any PL/SQL data type except `REF CURSOR`
- each key is a unique index, used to locate the associated value with the syntax `variable_name(index)`
- the datatype of index can be either a string type (`VARCHAR2, VARCHAR, STRING, or LONG`) or `PLS_INTEGER`
- indexes are sorted in sort order, not creation order. For string types, sort order is determined by the initialization parameters `NLS_SORT` and `NLS_COMP`
- like a database table, an associative array:
    - is empty (but not null) until you populate it
    - can hold an unspecified number of elements, which you can access without knowing their positions
- unlike a database table, an associative array:
    - does not need disk space or network operations
    - cannot be manipulated with DML statements
- Declaring associative array constants
    - when declaring an associative array constant, you must create a function that populates the associative array with its initial value and then invoke the function in the constant declaration
- NLS parameter values affect associative arrays indexed by string
    - Changing NLS parameter values after populating associative arrays
        - the initialization parameters `NLS_SORT` and `NLS_COMP` determine the storage order of string indexes of an associative array
        - if you change the value of either parameter after populating an associative array indexed by string, then the collection methods `FIRST, LAST, NEXT, and PRIOR` might return unexpected values or raise exceptions
        - if you must change these parameter values during your session, restore their original values before operating on associative arrays indexed by string
    - Indexes of data types other than VARCHAR2
        - in the declaration of an associative array indexed by string, the string type must be `VARCHAR2` or one of its subtypes
        - however, you can populate the associative array with indexes of any data type that the `TO_CHAR` function can convert to `VARCHAR2`
        - if your indexes have data types other than `VARCHAR2` and its subtypes, ensure that these indexes remain consistent and unique if the values of initialization parameters change, e.g.

        - do not use `TO_CHAR(SYSDATE)` as an index (if `NLS_DATE_FORMAT` changes, then value of this might also change)
        - do not use different `NVARCHAR2` indexes that might be converted to the same `VARCHAR2` value
        - do not use `CHAR` or `VARCHAR2` indexes that differ only in case, accented characters, or punctuation characters (if `NLS_SORT` ends in `_CI`, case-insensitive comparisons, or `_AI`, accent- and case-insensitive comparisons, then these indexes might be converted to the same value)
    - Passing associative arrays to remote databases
        - if you pass an associative array as a parameter to a remote database, and the local and remote databases have different `NLS_SORT or NLS_COMP` values, then
            - colletion methods might return unexpected results
            - indexes unique on local db might not be unique on remote db, raising the predefined exception `VALUE_ERROR`

- Appropriate uses for associative arrays
    - a relatively small lookup table, which can be constructed in memory each time you invoke the subprogram or initialize the package that declares it
    - passing collections to and from the database server
        - declare formal subprogram parameters of associative array types
        - with oracle call interface (OCI) or an oracle precompiler, bind the host arrays to the corresponding actual parameters. Plsql automatically converts between host arrays and associative arrays indexed by `PLS_INTEGER`
        - > Note: you cannot bind an associative array indexed by VARCHAR
        - > Note: you cannot declare an associative array type at schema level. Therefore, to pass an associative array variable as a parameter to a standalone subprogram, you must declare the type of that variable in a package specificaton. Doing so makes the type available to both the invoked subprogram (which declares a formal parameter of that type) and the invoking subprogram or anonymous block (which declares and passes the variable of that type)
        - > Tip: the most efficient way to pass collections to and from the database server is to use associative arrays with the FORALL statement or BULK COLLECT clause
    - an associative array is intended for temporary data storage. To make an associative array persistent for the life of a database session, declare it in a package specification and populate it in the package body

✅Describe the differences among records, collections, and collections of records

Varray: 5.3 Nested table: 5.4

Varrays (variable-size arrays)

- a varray (variable-size array) is an array whose number of elements can vary from zero (empty) to the declared maximum size
- type def syntax:

```
TYPE type IS {VARRAY | [VARYING] ARRAY} (size_limit) OF datatype[(length)]
[NOT NULL];
```

- size_limit is the maximum number of elements that the varray can have. size_limit must be an integer literal in the range from 1 through 2147483647
- datatype can be any PL/SQL data type except `REF CURSOR`
- to access an element of a varray variable, use the syntax `variable_name(index)`
- the lower bound of index is 1; the upper bound is the current number of elements. The upper bound changes as you add or delete elements, but it cannot exceed the maximum size
- when you store and retrieve a varray from the database, its indexes and element order remain stable
- the database stores a varray variable as a single object. If a varray variable is less than 4 KB, it resides inside the table of which it is a column; otherwise, it resides outside the table but in the same tablespace
- an uninitialized varray variable is a null collection. You must initialize it, either by making it empty or by assigning a non-NULL value to it
- a varray is appropriate when:
    - you know the maximum number of elements
    - you usually access the elements sequentially
    - because you must store or retrieve all elements at the same time, a varray might be impractical for large numbers of elements

Nested tables

- in the database, a nested table is a column type that stores an unspecified number of rows in no particular order
- type def syntax:

```
TYPE type IS TABLE OF datatype[(length)] [NOT NULL];
```

- datatype can be any PL/SQL data type except `REF CURSOR` or `NCLOB`
- when you retrieve a nested table value from the database into a plsql nested table variable, plsql gives the rows consecutive indexes, starting at 1

- using these indexes, you can access the individual rows of the nested table variable, the syntax is `variable_name(index)`
- the indexes and row order of a nested table might not remain stable as you store and retrieve the nested table from the database
- the amount of memory that a nested table variable occupies can increase or decrease dynamically, as you add or delete elements
- an uninitialized nested table variable is a null collection. You must initialize it, either by making it empty or by assigning a non-NULL value to it
- conceptually, a nested table is like a one-dimensional array with an arbitrary number of elements, however a nested table differs from an array in these important ways:
    - an array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically
    - an array is always dense. A nested array is dense initially, but it can become sparse, becuase you can delete elements from it
- a nested table is appropriate when:
    - the number of elements is not set
    - index values are not consecutive
    - you must delete or update some elements, but not all elements simultaneously Nested table data is stored in a separate store table, a system-generated database table. When you access a nested table, the database joins the nested table with its store table. This makes nested tables suitable for queries and updates that affect only some elements of the collection
    - you would create a separate lookup table, with multiple entries for each row of the main table, and access it through join queries

✅Initialize collections and records

Collection constructors: 5.5 Qualified expressions: 5.6 Assign values to collection variables: 5.7 Collection comparison: 5.9 Collection methods: 5.10 Collection types defined in package spec: 5.11 Iterating through collections LiveSQL examples

Collection constructors

- a collection constructor (constructor) is a system-defined function with the same name as a collection type, which returns a collection of that type
- collection constructors apply only to varrays and nested tables, associative arrays use qualified expressions and aggregates
- constructor invocation syntax: `collection_type ( [ value [, value ] ] )`
- if the parameter list is empty, the constructor returns an empty collection. Otherwise the constructor returns a collection that contains the specified values
- you can assign the returned collection to a collection variable (of the same type) in the variable declaration and in the executable part of a block

Qualified expressions overview

- qualified expressions improve program clarity and developer productivity by providing the ability to declare and define a complex value in a compact form wher the value is needed
- a qualified expression combines expression elements to create values of a `RECORD` type or associative array type
- qualified expressions use an explicit type indication to provide the type of the qualified item. This explicit indication is known as a typemark
- example, assigning values to associative array type variables using qualified expressions:

```
DECLARE
  TYPE t_aa IS TABLE OF VARCHAR2(50) INDEX BY PLS_INTEGER;
  v_aa1 t_aa := t_aa(1 => 'hello', 2 => 'world', 3 => '!');
BEGIN
  DBMS_OUTPUT.PUT_LINE(v_aa1(1) || v_aa1(2) || v_aa1(3));
END;
/
-- output
helloworld!
```

Assigning values to collection variables

You can assign a value to a collection variable in these ways:

- invoke a constructor to create a collection and assign it to the collection variable
- use the assignment statement to assign it the value of another existing collection variable
- pass it to a subprogram as an OUT or IN OUT parameter, and then assign the value inside the subprogram
- use a qualified expression to assign values to an associative array

To assign a value to a scalar element of a collection variable, reference the element as collection_variable_name(index) and assign it a value

- you can assign a collection to a collection variable only if they have the same data type. Having the same element type is not enough, e.g.

```
DECLARE
  TYPE twin IS VARRAY(2) OF VARCHAR2(10);
  TYPE duo IS VARRAY(2) OF VARCHAR2(10);
  group1 twin := twin('hello', 'world');  -- datatype twin
  group2 twin;                            -- datatype twin
  group3 duo;                             -- datatype duo
BEGIN
  group2 := group1; --succeeds
  group3 := group1; --fails
END;
```

```
/
ORA-06550: PLS-00382: expression is of wrong type
```

- to a varray or nested table variable, you can assign the value `NULL` or a null collection of the same data type, either assignment makes the variable null

Collection comparisons

- to determine if one collection variable is less than another (for example), you must define what less than means in that context and write a function that returns `TRUE or FALSE`
- you cannot compare associative array variables to the value `NULL` or to each other
- you cannot natively compare two collection variables with relational operators. This restriction also applies to implicit comparisons, e.g. a collection variable cannot appear in a `DISTINCT, GROUP BY, or ORDER BY` clause
    - two nested variables are equal if and only if they have the same set of elements (in any order).
    - if two nested table variables have the same nested table type, and that nested table type does not have elements of a record type, then you can compare the two variables for equality or inequality with the relational operators equal (=) and not equal (<>, !=, ~=, ^=)
- use the `IS [NOT] NULL` operator when comparing to the NULL value. You can compare varray and nested table variables to the value `NULL` with the `IS [NOT] NULL` operator, but not with the relational operators equal (=) and not equal (<>,...)

Collection methods

A collection method is a plsql subprogram, either a function that returns information about a collection or a procedure that operates on a collection. Collection methods make collections easier to use and your applications easier to maintain. Collection method invocation syntax: `collection_name.method` A collection method invocation can appear anywhere that an invocation of a plsql subprogram of its type can appear, except in a sql statement

- DELETE collection method
    - `DELETE` is a procedure that deletes elements from a collection
    - this method has these forms:
        - `DELETE` deletes all elements from a collection of any type. This operation immediately frees the memory allocated to the deleted elements
        - from an associative array or nested table (but not varray):
            - `DELETE(n)` deletes the element whose index is n, if that element exists; otherwise does nothing
            - `DELETE(m,n)` deletes all elements whose indexes are in the range `m .. n`, if both m and n exist and m <= n; otherwise does nothing

    for these two forms of `DELETE`, plsql keeps placeholders for the deleted elements, therefore the deleted elements are included in the internal size of the collection and you can restore a deleted element by assigning a valid value to it

- TRIM collection method

    - `TRIM` is a procedure that deletes elements from the end of a varray or nested table
    - forms:
        - `TRIM` removes one element from the end of the collection, if the collection has at least one element; otherwise, predefined exception `SUBSCRIPT_BEYOND_COUNT` raised
        - `TRIM(n)` removes n elements from the end of the collection, if there are at least n elements at the end; otherwise exception `SUBSCRIPT_BEYOND_COUNT`
    - `TRIM` operates on the internal size of a collection. That is, if `DELETE` deletes an element but keeps a placeholder for it, then `TRIM` considers the element to exist, therefore `TRIM` can delete a deleted element (Caution: do not depend on this interaction between `TRIM` and `DELETE`)
    - plsql does not keep placeholders for trimmed elements. Therefore trimmed elements are not included in the internal size of the collection, and you cannot restore a trimmed element by assigning a valid value to it

- EXTEND collection method

    - `EXTEND` is a procedure that adds elements to the end of a varray or nested table
    - the collection can be empty, but not null (to make collection empty or add elements to a null collection, use a constructor)
    - forms:
        - `EXTEND` appends one null element to the collection
        - `EXTEND(n)` appends n null elements to the collection
        - `EXTEND(n,i)` appends n copies of the *i*th element to the collection (this is the only form that can be used for a collection whose elements have the `NOT NULL` constraint)
    - `EXTEND` operates on the internal size of a collection. That is, if `DELETE` deletes an element but keeps a placeholder for it, then `EXTEND` considers the element to exist (can result in skipping of indexes)

- EXISTS collection method

    - `EXISTS` is a function that tells you whether the specified element of a varray or nested table exists
    - `EXISTS(n)` returns `TRUE` if the nth element of the collection exists and `FALSE` otherwise.
    - if n is out of range, `FALSE` is returned instead of raising exception `SUBSCRIPT_OUTSIDE_LIMIT`
    - for a deleted element, `EXISTS(n)` returns `FALSE`, even if `DELETE` kept a placeholder for it

- FIRST and LAST collection methods

    - `FIRST` and `LAST` are functions
    - if the collection has at least one element, `FIRST and LAST` return the indexes of the first and last elements, respectively (ignoring deleted elements, even if placeholders kept)
    - if the collection has only one element, `FIRST` and `LAST` return the same index
    - if collection empty, they both return `NULL`
    - behaviour in associative array:
        - if indexed by `PLS_INTEGER`, the first and last elements are those with the smallest and largest indexes, respectively
        - if indexed by string, first and last are those with lowest and highest key values, respectively (key values in sorted order)
    - behaviour in varray:

        - for varray that is not empty, `FIRST` always returns 1
        - for every varray, `LAST` always equals `COUNT`
    - behaviour in nested table:
        - for a nested table, `LAST` equals `COUNT` unless you delete elements from its middle, in which case `LAST` is larger than `COUNT`

- COUNT collection method
    - `COUNT` is a function that returns the number of elements in the collection (ignoring deleted elements, even if placeholders kept)
    - behaviour for varray:
        - `COUNT` always equals `LAST`
        - if you increase or decrease the size of a varray (with `EXTEND` or `TRIM`), the value of `COUNT` changes
    - behaviour for nested table:
        - `COUNT` always equals `LAST` unless you delete elements from the middle of the nested table, in which case `COUNT` is smaller than `LAST`

- LIMIT collection method
    - `LIMIT` is a function that returns the maximum number of elements that the collection can have
    - if the collection has no maximum number of elements, `LIMIT` returns `NULL`. Only a varray has a maximum size (thus, `LIMIT` always returns `NULL` for associative array and nested table)

- PRIOR and NEXT collection methods
    - `PRIOR` and `NEXT` are functions that let you move backward and forward in the collection (ignoring deleted elements, even if placeholders kept)
    - these methods are useful for traversing sparse collections
    - given an index:
        - `PRIOR` returns the index of the preceding existing element of the collection, if one exists; otherwise `PRIOR` returns `NULL`. For any collection c, `c.PRIOR(c.FIRST)` returns `NULL`
        - `NEXT` returns the index of the succeeding existing element of the collection, if one exists; otherwise, `NEXT` returns `NULL`. For any collection c, `c.NEXT(c.LAST)` returns `NULL`
    - the given index need not exist, however if the collection c is a varray, and the index exceeds `c.LIMIT`, then `c.PRIOR(index)` returns `c.LAST` and `c.NEXT(index)` returns `NULL`
    - for an associative array indexed by string, the prior and next indexes are determined by key values, which are in sorted order

Collection types defined in package specifications

- a collection type defined in a package specification is **incompatible** with an identically defined local or standalone collection type