Composite Data Types

Objectives

After completing this lesson, you should be able to do the following:

- Describe PL/SQL collections and records
- Create user-defined PL/SQL records
- Create a PL/SQL record with the `%ROWTYPE` attribute
- Create associative arrays
  - `INDEX BY` table
  - `INDEX BY` table of records

You have already been introduced to composite data types. In this lesson, you learn more about composite data types and their uses.

Composite Data Types

- Can hold multiple values (unlike scalar types) or the composite data type.
- Are of two types:
  - PL/SQL records: Records are used to treat related but dissimilar data as a logical unit. This makes data access and manipulation easier. Concept is same as structures.
  - PL/SQL collections: Collections are used to treat data as a single unit. Concept is same as Arrays.
    - Associative array (`INDEX BY` table)
    - Nested table
    - `VARRAY`

ORACLE

You learned that variables of the scalar data type can hold only one value, whereas variables of the composite data type can hold multiple values of the scalar data type or the composite data type. There are two types of composite data types:

- **PL/SQL records:** Records are used to treat related but dissimilar data as a logical unit. A PL/SQL record can have variables of different types. For example, you can define a record to hold employee details. This involves storing an employee number as `NUMBER`, a first name and last name as `VARCHAR2`, and so on. By creating a record to store employee details, you create a logical collective unit. This makes data access and manipulation easier.
- **PL/SQL collections:** Collections are used to treat data as a single unit. Collections are of three types:
  - Associative array
  - Nested table
  - `VARRAY`

Why Use Composite Data Types?

You have all the related data as a single unit. You can easily access and modify data. Data is easier to manage, relate, and transport if it is composite. An analogy is having a single bag for

all your laptop components rather than a separate bag for each component.

Why Use Composite Data Types?

- You have all the related data as a single unit.
- You can easily access and modify data.
- Data is easier to manage, relate, and transport if it is composite.
- An analogy is having a single bag for all your laptop components rather than a separate bag for each component.

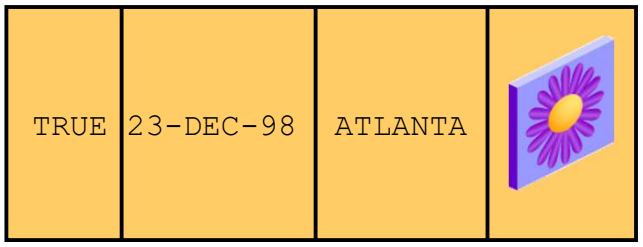
If both PL/SQL records and PL/SQL collections are composite types, how do you choose which one to use?

- Use PL/SQL records when you want to store values of different data types that are logically related. For example, you can create a PL/SQL record to hold employee details and indicate that all the values stored are related because they provide information about a particular employee.
- Use PL/SQL collections when you want to store values of the same data type. Note that this data type can also be of the composite type (such as records). You can define a collection to hold the first names of all employees. You may have stored *n* names in the collection; however, name 1 is not related to name 2. The relation between these names is only that they are employee names. These collections are similar to arrays in programming languages such as C, C++, and Java.

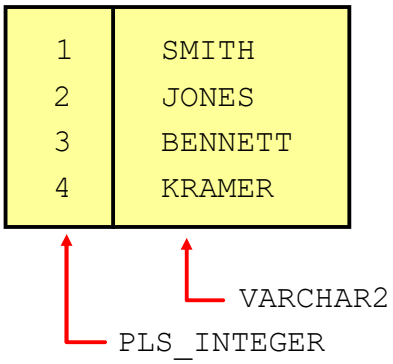
PL/SQL Records or Collections?

- Use PL/SQL records when you want to store values of different data types but only one occurrence at a time.
- Use PL/SQL collections when you want to store values of the same data type.

PL/SQL Record:



PL/SQL Collection:



If both PL/SQL records and PL/SQL collections are composite types, how do you choose which one to use?

- Use PL/SQL records when you want to store values of different data types that are logically related. For example, you can create a PL/SQL record to hold employee details and indicate that all the values stored are related because they provide information about a particular employee.
- Use PL/SQL collections when you want to store values of the same data type. Note that this data type can also be of the composite type (such as records). You can define a collection to hold the first names of all employees. You may have stored *n* names in the collection; however, name 1 is not related to name 2. The relation between these names is only that they are employee names. These collections are similar to arrays in programming languages such as C, C++, and Java.

PL/SQL Records

- A record is a group of related data items stored in fields, each with its own name and data type.
- Must contain one or more components (called *fields*) of any scalar, `RECORD`, or `INDEX BY` table data type
- Are similar to structures in most third-generation languages (including C and C++)
- Are user-defined and can be a subset of a row in a table
- Treat a collection of fields as a logical unit
- Are convenient for fetching a row of data from a table for processing

A record is a group of related data items stored in fields, each with its own name and data type.

- Each record defined can have as many fields as necessary.
- Records can be assigned initial values and can be defined as `NOT NULL`.
- Fields without initial values are initialized to `NULL`.
- The `DEFAULT` keyword as well as `:=` can be used in initializing fields.
- You can define `RECORD` types and declare user-defined records in the declarative part of any block, subprogram, or package.
- You can declare and reference nested records. One record can be the component of another record.

Creating a PL/SQL Record

Syntax:

1
```
TYPE type_name IS RECORD
     (field_declaration[, field_declaration]…);
```

2
```
identifier    type_name;
```

field_declaration:

```
field_name {field_type | variable%TYPE
           | table.column%TYPE | table%ROWTYPE}
           [[NOT NULL] {:= | DEFAULT} expr]
```

ORACLE

PL/SQL records are user-defined composite types. To use them, perform the following steps:

1. Define the record in the declarative section of a PL/SQL block. The syntax for defining the record is shown in the slide.
2. Declare (and optionally initialize) the internal components of this record type.

In the syntax:

`type_name`	Is the name of the `RECORD` type (This identifier is used to declare records.)
`field_name`	Is the name of a field within the record
`field_type`	Is the data type of the field (It represents any PL/SQL data type except `REF CURSOR`. You can use the `%TYPE` and `%ROWTYPE` attributes.)
expr	Is the initial value

The `NOT NULL` constraint prevents assigning of nulls to the specified fields. Be sure to initialize the `NOT NULL` fields.

Creating a PL/SQL Record: Example

```
DECLARE
  TYPE t_rec IS RECORD
    (v_sal number(8),
     v_minsal number(8) default 1000,
     v_hire_date employees.hire_date%type,
     v_rec1 employees%rowtype);
  v_myrec t_rec;
BEGIN
  v_myrec.v_sal := v_myrec.v_minsal + 500;
  v_myrec.v_hire_date := sysdate;
  SELECT * INTO v_myrec.v_rec1
      FROM employees WHERE employee_id = 100;
  DBMS_OUTPUT.PUT_LINE(v_myrec.v_rec1.last_name ||' '||
  to_char(v_myrec.v_hire_date) ||' '|| to_char(v_myrec.v_sal));
END;
```

```
anonymous block completed
King 16-OCT-12 1500
```

ORACLE

The field declarations used in defining a record are like variable declarations. Each field has a unique name and a specific data type. There are no predefined data types for PL/SQL records, as there are for scalar variables. Therefore, you must create the record type first, and then declare an identifier using that type.

In the example in the slide, a PL/SQL record is created using the required two-step process:

1. A record type (`t_rec`) is defined.
2. A record (`v_myrec`) of the `t_rec` type is declared.

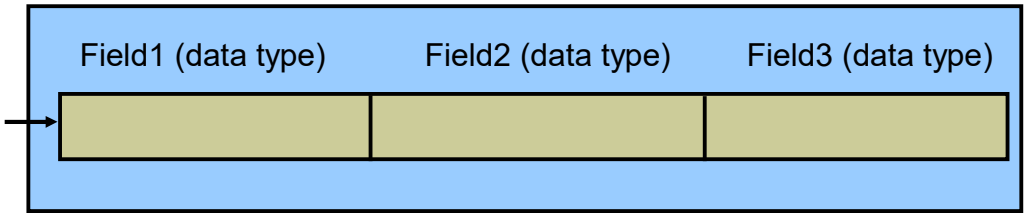
Note

- The record contains four fields: `v_sal`, `v_minsal`, `v_hire_date`, and `v_rec1`.
- `v_rec1` is defined using the `%ROWTYPE` attribute, which is similar to the `%TYPE` attribute. With `%TYPE`, a field inherits the data type of a specified column. With `%ROWTYPE`, a field inherits the column names and data types of all columns in the referenced table.
- PL/SQL record fields are referenced using the `<record>.<field>` notation, or the `<record>.<field>.<column>` notation for fields that are defined with the `%ROWTYPE` attribute.
- You can add the `NOT NULL` constraint to any field declaration to prevent assigning nulls

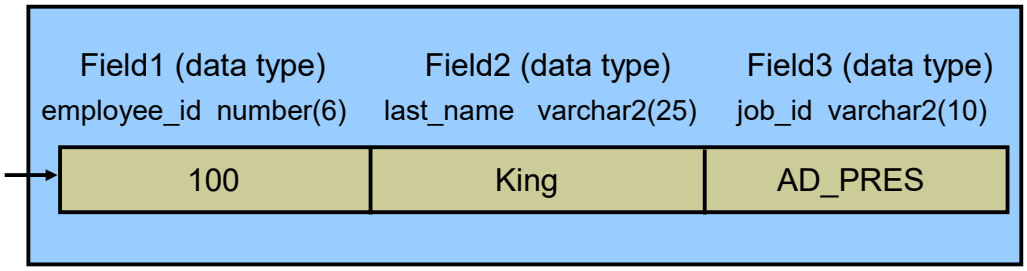
to that field. Remember that fields that are declared as `NOT NULL` must be initialized.

PL/SQL Record Structure

Field declarations:



Example:



Fields in a record are accessed with the name of the record. To reference or initialize an individual field, use the dot notation:

```
record_name.field_name
```

For example, you reference the `job_id` field in the `emp_record` record as follows:

```
emp_record.job_id
```

You can then assign a value to the record field:

```
emp_record.job_id := 'ST_CLERK';
```

In a block or subprogram, user-defined records are instantiated when you enter the block or subprogram. They cease to exist when you exit the block or subprogram.

%ROWTYPE Attribute

- Declare a variable according to a collection of columns in a database table or view.
- Prefix %ROWTYPE with the database table or view.
- Fields in the record take their names and data types from the columns of the table or view.

Syntax:

```
DECLARE
   identifier  reference%ROWTYPE;
```


You learned that %TYPE is used to declare a variable of the column type. The variable has the same data type and size as the table column. The benefit of %TYPE is that you do not have to change the variable if the column is altered. Also, if the variable is a number and is used in any calculations, you need not worry about its precision.

The %ROWTYPE attribute is used to declare a record that can hold an entire row of a table or view. The fields in the record take their names and data types from the columns of the table or view. The record can also store an entire row of data fetched from a cursor or cursor variable.

The slide shows the syntax for declaring a record. In the syntax:

identifier	Is the name chosen for the record as a whole
reference	Is the name of the table, view, cursor, or cursor variable on which the record is to be based (The table or view must exist for this reference to be valid.)

In the following example, a record is declared using %ROWTYPE as a data type specifier:

```
DECLARE
 emp_record  employees%ROWTYPE;
 ...
```

The emp_record record has a structure consisting of the following fields, each representing a column in the employees table.

Note: This is not code, but simply the structure of the composite variable.

```
(employee_id        NUMBER(6),
first_name         VARCHAR2(20),
last_name          VARCHAR2(20),
email              VARCHAR2(20),
phone_number       VARCHAR2(20),
hire_date          DATE,
job_id             VARCHAR2(10),
salary             NUMBER(8,2),
commission_pct     NUMBER(2,2),
manager_id         NUMBER(6),
department_id      NUMBER(4))
```

To reference an individual field, use the dot notation:

```
record_name.field_name
```

For example, you reference the commission_pct field in the emp_record record as follows:

```
emp_record.commission_pct
```

You can then assign a value to the record field:

```
emp_record.commission_pct:= .35;
```

Assigning Values to Records

You can assign a list of common values to a record by using the SELECT or FETCH statement. Make sure that the column names appear in the same order as the fields in your record. You can also assign one record to another if both have the same corresponding data types. A record of type employees%ROWTYPE and a user-defined record type having analogous fields of the employees table will have the same data type. Therefore, if a user-defined record contains fields similar to the fields of a %ROWTYPE record, you can assign that user-defined record to the %ROWTYPE record.

Advantages of Using the %ROWTYPE Attribute

- The number and data types of the underlying database columns need not be known—and, in fact, might change at run time.
- The %ROWTYPE attribute is useful when you want to retrieve a row with:
  - The SELECT * statement
  - Row-level INSERT and UPDATE statements

The advantages of using the %ROWTYPE attribute are listed in the slide. Use the %ROWTYPE attribute when you are not sure about the structure of the underlying database table.

The main advantage of using %ROWTYPE is that it simplifies maintenance. Using %ROWTYPE ensures that the data types of the variables declared with this attribute change dynamically when the underlying table is altered. If a DDL statement changes the columns in a table, the PL/SQL program unit is invalidated. When the program is recompiled, it automatically reflects the new table format.

The %ROWTYPE attribute is particularly useful when you want to retrieve an entire row from a table. In the absence of this attribute, you would be forced to declare a variable for each of the columns retrieved by the SELECT statement.

Another %ROWTYPE Attribute Example

```
DECLARE
  v_employee_number number:= 124;
  v_emp_rec   employees%ROWTYPE;
BEGIN
 SELECT * INTO v_emp_rec FROM employees
 WHERE  employee_id = v_employee_number;
 INSERT INTO retired_emps(empno, ename, job, mgr,
                 hiredate, leavedate, sal, comm, deptno)
   VALUES (v_emp_rec.employee_id, v_emp_rec.last_name,
           v_emp_rec.job_id, v_emp_rec.manager_id,
           v_emp_rec.hire_date, SYSDATE,
           v_emp_rec.salary,  v_emp_rec.commission_pct,
           v_emp_rec.department_id);
END;
/
```

```
SELECT * FROM retired_emps;
/
```

Query Result | SQL | All Rows Fetched: 1 in 0.005 seconds

	EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-07	16-OCT-12	5800	(null)	50

ORACLE

Another example of the %ROWTYPE attribute is shown in the slide. If an employee is retiring, information about that employee is added to a table that holds information about retired employees. The user supplies the employee number. The record of the employee specified by the user is retrieved from the employees table and stored in the emp_rec variable, which is declared using the %ROWTYPE attribute.

The CREATE statement that creates the retired_emps table is:

```
CREATE TABLE retired_emps
   (EMPNO      NUMBER(4), ENAME      VARCHAR2(10),
    JOB        VARCHAR2(9),MGR        NUMBER(4),
    HIREDATE   DATE, LEAVEDATE  DATE,
    SAL        NUMBER(7,2), COMM       NUMBER(7,2),
    DEPTNO     NUMBER(2))
```

Note

- The record that is inserted into the retired_emps table is shown in the slide.
- To see the output shown in the slide, place your cursor on the SELECT statement at the bottom of the code example in SQL Developer and press F9.

- The complete code example is found under slide 14_s-n in `code_ex_07.sql`.

Inserting a Record by Using %ROWTYPE

```
...
DECLARE
  v_employee_number number:= 124;
  v_emp_rec retired_emps%ROWTYPE;
BEGIN
 SELECT employee_id, last_name, job_id, manager_id,
 hire_date, hire_date, salary, commission_pct,
 department_id INTO v_emp_rec FROM employees
 WHERE  employee_id = v_employee_number;
 INSERT INTO retired_emps VALUES v_emp_rec;
END;
/
SELECT * FROM retired_emps;
```

Query Result

SQL | All Rows Fetched: 1 in 0.002 seconds

	EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-07	16-NOV-07	5800	(null)	50

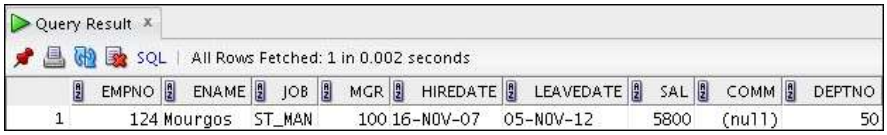
Compare the `INSERT` statement in the previous slide with the `INSERT` statement in this slide. The `emp_rec` record is of type `retired_emps`. The number of fields in the record must be equal to the number of field names in the `INTO` clause. You can use this record to insert values into a table. This makes the code more readable.

Examine the `SELECT` statement in the slide. You select `hire_date` twice and insert the `hire_date` value in the `leavedate` field of `retired_emps`. No employee retires on the hire date. The inserted record is shown in the slide. (You will see how to update this in the next slide.)

Note: To see the output shown in the slide, place your cursor on the `SELECT` statement at the bottom of the code example in SQL Developer and press F9.

Updating a Row in a Table by Using a Record

```
DECLARE
  v_employee_number number:= 124;
  v_emp_rec  retired_emps%ROWTYPE;
BEGIN
  SELECT * INTO v_emp_rec FROM retired_emps WHERE
  empno = v_employee_number;
  v_emp_rec.leavedate:= CURRENT_DATE;
  UPDATE retired_emps SET ROW = v_emp_rec WHERE
  empno=v_employee_number;
END;
/
SELECT * FROM retired_emps;
```


Query Result

SQL | All Rows Fetched: 1 in 0.002 seconds

	EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-07	05-NOV-12	5800	(null)	50

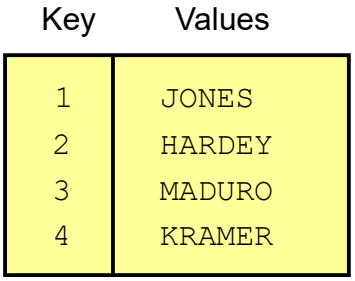
You learned to insert a row by using a record. This slide shows you how to update a row by using a record.

- The `ROW` keyword is used to represent the entire row.
- The code shown in the slide updates the `leavedate` of the employee.
- The record is updated as shown in the slide.

Note: To see the output shown in the slide, place your cursor on the `SELECT` statement at the bottom of the code example in SQL Developer and press F9.

Associative Arrays (`INDEX BY` Tables)

- INDEX BY tables are sets of key-value pairs
- Key can be of integer or string data type
- Value Column can be scalar or record data type
- Are unconstrained in size.



An associative array is a type of PL/SQL collection. It is a composite data type, and is user-defined. Associative arrays are sets of key-value pairs. They can store data using a primary key value as the index, where the key values are not necessarily sequential. Associative arrays are also known as *`INDEX BY`* tables.

Associative arrays have only two columns, neither of which can be named:

- The first column, of integer or string type, acts as the primary key.
- The second column, of scalar or record data type, holds values.

Associative Arrays (`INDEX BY` Tables)

- Key can of integer or strings
- Key can be negative
- key values need not to be sequential
- The size is not fixed.
- It cannot be created as a database object
- It can not be valid data type for table
- Item can be deleted from anywhere

Steps to Create an Associative Array

Syntax:

```
1  TYPE type_name IS TABLE OF
   { column_type [NOT NULL] | variable%TYPE [NOT NULL]
   | table.column%TYPE [NOT NULL]
   | table%ROWTYPE }
   INDEX BY { PLS_INTEGER | BINARY_INTEGER
   | VARCHAR2(<size>) } ;
2  identifier type_name;
```

Example:

```
...
TYPE ename_table_type IS TABLE OF
 employees.last_name%TYPE
 INDEX BY PLS_INTEGER;
...
ename_table ename_table_type;
```


There are two steps involved in creating an associative array:

1. Declare a `TABLE` data type using the `INDEX BY` option.
2. Declare a variable of that data type.

Syntax

type_name	Is the name of the `TABLE` type (This name is used in the subsequent declaration of the array identifier.)
column_type	Is any scalar or composite data type such as `VARCHAR2`, `DATE`, `NUMBER`, or `%TYPE` (You can use the `%TYPE` attribute to provide the column data type.)
identifier	Is the name of the identifier that represents an entire associative array

Note: The `NOT NULL` constraint prevents nulls from being assigned to the associative array.

Example

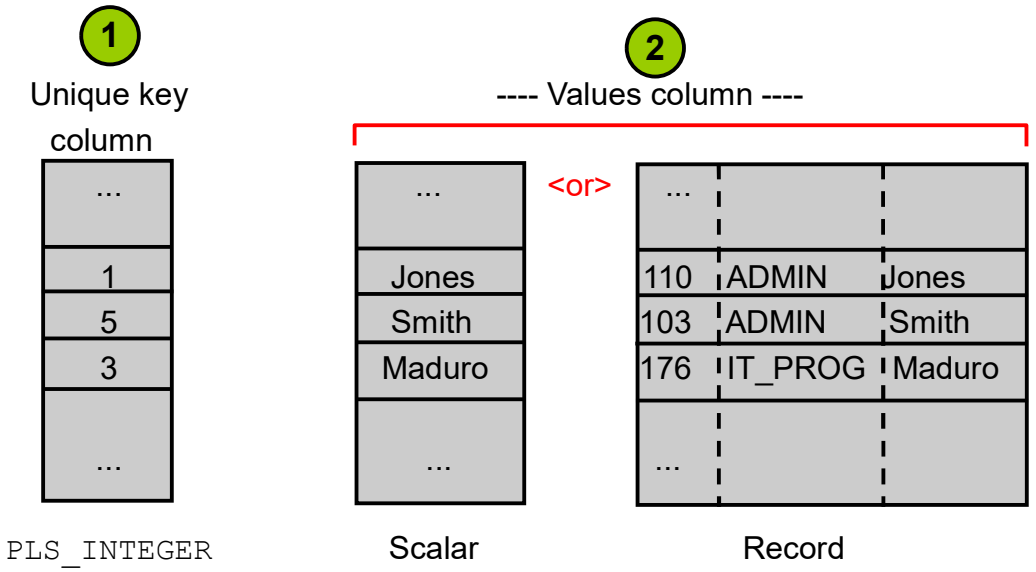
In the example, an associative array with the variable name `ename_table` is declared to

store the last names of employees.

Creating an INDEX BY Table

```
declare

type tab_no is table of varchar2(100)
index by pls_integer;

v_tab_no tab_no;

begin

v_tab_no(1):='John';
v_tab_no(6):='Allan';
v_tab_no(4):='Roy';

dbms_output.put_line(v_tab_no(1));
dbms_output.put_line(v_tab_no(6));
dbms_output.put_line(v_tab_no(4));
end;
```

Associative Array Structure



As previously mentioned, associative arrays have two columns. The second column holds either one value per row or multiple values.

Unique Key Column: The data type of the key column can be:

- Numeric, either `BINARY_INTEGER` or `PLS_INTEGER`. These two numeric data types require less storage than `NUMBER`, and arithmetic operations on these data types are faster than the `NUMBER` arithmetic.
- `VARCHAR2` or one of its subtypes

"Value" Column: The value column can be either a scalar data type or a record data type. A column with scalar data type can hold only one value per row, whereas a column with record data type can hold multiple values per row.

Other Characteristics

- An associative array is not populated at the time of declaration. It contains no keys or values, and you cannot initialize an associative array in its declaration.
- An explicit executable statement is required to populate the associative array.
- Like the size of a database table, the size of an associative array is unconstrained. That is, the number of rows can increase dynamically so that your associative array grows as

new rows are added. Note that the keys do not have to be sequential, and can be both positive and negative.

Creating and Accessing Associative Arrays

```
...
DECLARE
  TYPE ename_table_type IS TABLE OF
    employees.last_name%TYPE
    INDEX BY PLS_INTEGER;
  TYPE hiredate_table_type IS TABLE OF DATE
    INDEX BY PLS_INTEGER;
  ename_table       ename_table_type;
  hiredate_table    hiredate_table_type;
BEGIN
  ename_table(1)    := 'CAMERON';
  hiredate_table(8) := SYSDATE + 7;
    IF ename_table.EXISTS(1) THEN
    INSERT INTO ...
    ...
END;
/
...
```

```
Script Output
Task completed in 0.047 seconds
anonymous block completed
ENAME                HIREDT
-------------------- ---------
CAMERON              23-OCT-12
```


The example in the slide creates two associative arrays, with the identifiers `ename_table` and `hiredate_table`.

The key of each associative array is used to access an element in the array, by using the following syntax:

```
identifier(index)
```

In both arrays, the `index` value belongs to the `PLS_INTEGER` type.

- To reference the first row in the `ename_table` associative array, specify: `ename_table(1)`
- To reference the eighth row in the `hiredate_table` associative array, specify: `hiredate_table(8)`

Note

- The magnitude range of a `PLS_INTEGER` is –2,147,483,647 through 2,147,483,647, so the primary key value can be negative. Indexing does not need to start with 1.
- The `exists(i)` method returns `TRUE` if a row with index *i* is returned. Use the `exists` method to prevent an error that is raised in reference to a nonexistent table element.
- The complete code example is found under slide 21_sa in `code_ex_07.sql`.

Using INDEX BY Table Methods

The following methods make associative arrays easier to use:

- EXISTS
- COUNT
- FIRST
- LAST
- PRIOR
- NEXT
- DELETE

EXISTS(n)	Return true if nth element in PL/SQL Table exists
COUNT	Return number of element in PL/SQL Table
FIRST	Return first index number in PL/SQL Table or NULL
LAST	Return last index number in PL/SQL Table or NULL
DELETE	Remove all elements in PL/SQL Table

ORACLE

An `INDEX BY` table method is a built-in procedure or function that operates on an associative array and is called by using the dot notation.

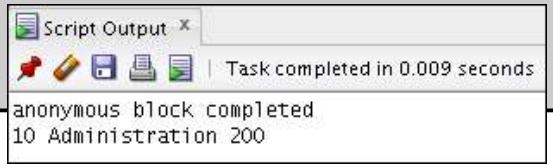
Syntax: *`table_name.method_name[ (parameters) ]`*

Method	Description
`EXISTS(n)`	Returns `TRUE` if the index *n* in an associative array exists
`COUNT`	Returns the number of elements that an associative array currently contains
`FIRST`	• Returns the first (smallest) index number in an associative array • Returns `NULL` if the associative array is empty
`LAST`	• Returns the last (largest) index number in an associative array • Returns `NULL` if the associative array is empty
`PRIOR(n)`	Returns the index number that precedes index *n* in an associative array
`NEXT(n)`	Returns the index number that succeeds index *n* in an associative array
`DELETE`	• `DELETE` removes all elements from an associative array. • `DELETE(n)` removes the index *n* from an associative array. • `DELETE(m, n)` removes all elements in the range *m* ... *n* from an associative array.

INDEX BY Table of Records Option

Define an associative array to hold an entire row from a table.

```
DECLARE
TYPE dept_table_type
IS
  TABLE OF departments%ROWTYPE INDEX BY VARCHAR2(20);
  dept_table dept_table_type;
  -- Each element of dept_table is a record
BEGIN
   SELECT * INTO dept_table(1) FROM departments
   WHERE department_id = 10;
   DBMS_OUTPUT.PUT_LINE(dept_table(1).department_id ||' '||
   dept_table(1).department_name ||' '||
   dept_table(1).manager_id);
END;
/
```



ORACLE

As previously discussed, an associative array that is declared as a table of scalar data type can store the details of only one column in a database table. However, there is often a need to store all the columns retrieved by a query. The `INDEX BY` table of records option enables one array definition to hold information about all the fields of a database table.

Creating and Referencing a Table of Records

As shown in the associative array example in the slide, you can:

- Use the `%ROWTYPE` attribute to declare a record that represents a row in a database table
- Refer to fields within the `dept_table` array because each element of the array is a record

The differences between the `%ROWTYPE` attribute and the composite data type PL/SQL record are as follows:

- PL/SQL record types can be user-defined, whereas `%ROWTYPE` implicitly defines the record.
- PL/SQL records enable you to specify the fields and their data types while declaring them. When you use `%ROWTYPE`, you cannot specify the fields. The `%ROWTYPE` attribute represents a table row with all the fields based on the definition of that table.

- User-defined records are static, but `%ROWTYPE` records are dynamic—they are based on a table structure. If the table structure changes, the record structure also picks up the change.

INDEX BY Table of Records Option: Example 2

```
DECLARE
   TYPE emp_table_type IS TABLE OF
      employees%ROWTYPE INDEX BY PLS_INTEGER;
   my_emp_table  emp_table_type;
   max_count         NUMBER(3):= 104;
BEGIN
  FOR i IN 100..max_count
  LOOP
   SELECT * INTO my_emp_table(i) FROM employees
   WHERE employee_id = i;
  END LOOP;
  FOR i IN my_emp_table.FIRST..my_emp_table.LAST
  LOOP
     DBMS_OUTPUT.PUT_LINE(my_emp_table(i).last_name);
  END LOOP;
END;
/
```


The example in the slide declares an associative array, using the `INDEX BY` table of records option, to temporarily store the details of employees whose employee IDs are between 100 and 104. The variable name for the array is `emp_table_type`.

Using a loop, the information of the employees from the `EMPLOYEES` table is retrieved and stored in the array. Another loop is used to print the last names from the array. Note the use of the `first` and `last` methods in the example.

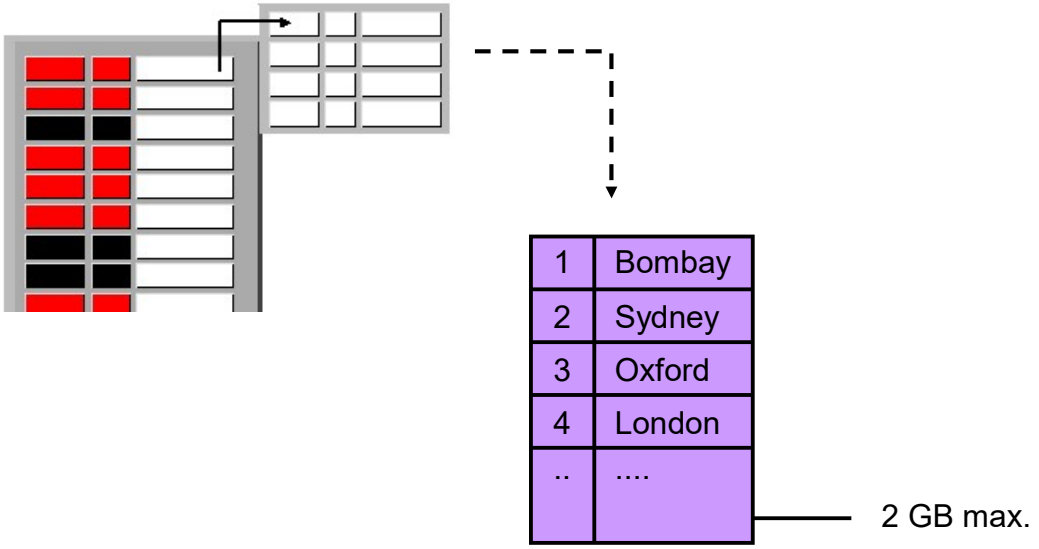
Note: The slide demonstrates one way to work with an associative array that uses the `INDEX BY` table of records method. However, you can do the same more efficiently using cursors. Cursors are explained in the lesson titled “Using Explicit Cursors.”

The results of the code example are as follows:

Nested Tables

- Nested tables is similar to that of INDEX BY tables;
- The nested table is a valid data type in a schema-level table, but an INDEX BY table is not
- Populated sequentially starting with the index '1'.
- Elements can be deleted from anywhere in a nested table
- Key can not be negative , unlike Index by table
- It can be created as a database object
- The Nested table has no upper size limit.
- Maximum size is 2Gb

Nested Tables



The functionality of nested tables is similar to that of associative arrays; however, there are differences in the nested table implementation.

- The nested table is a valid data type in a schema-level table, but an associative array is not. Therefore, unlike associative arrays, nested tables can be stored in the database.
- The size of a nested table can increase dynamically, although the maximum size is 2 GB.
- The “key” cannot be a negative value (unlike in the associative array). Though reference is made to the first column as key, there is no key in a nested table. There is a column with numbers.
- Elements can be deleted from anywhere in a nested table, leaving a sparse table with nonsequential “keys.” The rows of a nested table are not in any particular order.
- When you retrieve values from a nested table, the rows are given consecutive subscripts starting from 1.

Syntax

```
TYPE type_name IS TABLE OF
 {column_type | variable%TYPE
 | table.column%TYPE} [NOT NULL]
```

```
| table.%ROWTYPE
```

Example:

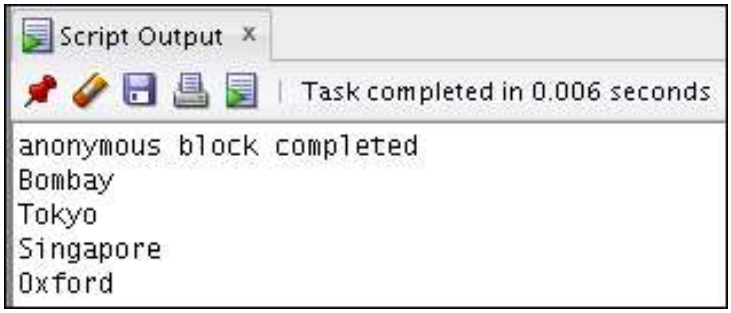
```
TYPE location_type IS TABLE OF locations.city%TYPE;
offices location_type;
```

If you do not initialize a nested table, it is automatically initialized to `NULL`. You can initialize the `offices` nested table by using a constructor:

```
offices := location_type('Bombay', 'Tokyo','Singapore',
'Oxford');
```

The complete code example and output is as follows:

```
SET SERVEROUTPUT ON;

DECLARE
   TYPE location_type IS TABLE OF locations.city%TYPE;
    offices location_type;
    table_count NUMBER;
BEGIN
   offices := location_type('Bombay', 'Tokyo','Singapore',
    'Oxford');
FOR i in 1.. offices.count() LOOP
     DBMS_OUTPUT.PUT_LINE(offices(i));
   END LOOP;
END;
/
```



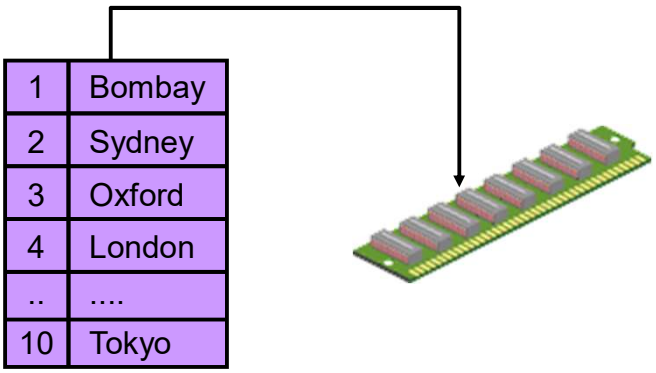
```
SET SERVEROUTPUT ON;

DECLARE
   TYPE location_type IS TABLE OF locations.city%TYPE;
    offices location_type;
    table_count NUMBER;
BEGIN
   offices := location_type('Bombay', 'Tokyo','Singapore',
    'Oxford');
      FOR i in 1.. offices.count() LOOP
         DBMS_OUTPUT.PUT_LINE(offices(i));
      END LOOP;
END;
/
```

VARRAY

- Upper limit size is fixed
- Populated sequentially starting with the index ‘1’
- Element will be deleted from end
- The maximum size of a VARRAY is 2 GB, as in nested tables.
- It can be created as a database object. VARRAY is valid data type in a schema-level table.

```
TYPE location_type IS VARRAY(3) OF locations.city%TYPE;
offices location_type;
```

VARRAY



A variable-size array (VARRAY) is similar to an associative array, except that a VARRAY is constrained in size.

- A VARRAY is valid in a schema-level table.
- Items of VARRAY type are called VARRAYs.
- VARRAYs have a fixed upper bound. You have to specify the upper bound when you declare them. This is similar to arrays in C language. The maximum size of a VARRAY is 2 GB, as in nested tables.
- The distinction between a nested table and a VARRAY is the physical storage mode. The elements of a VARRAY are stored inline with the table's data unless the size of the VARRAY is greater than 4 KB. Contrast that with nested tables, which are always stored out-of-line.
- You can create a VARRAY type in the database by using SQL.

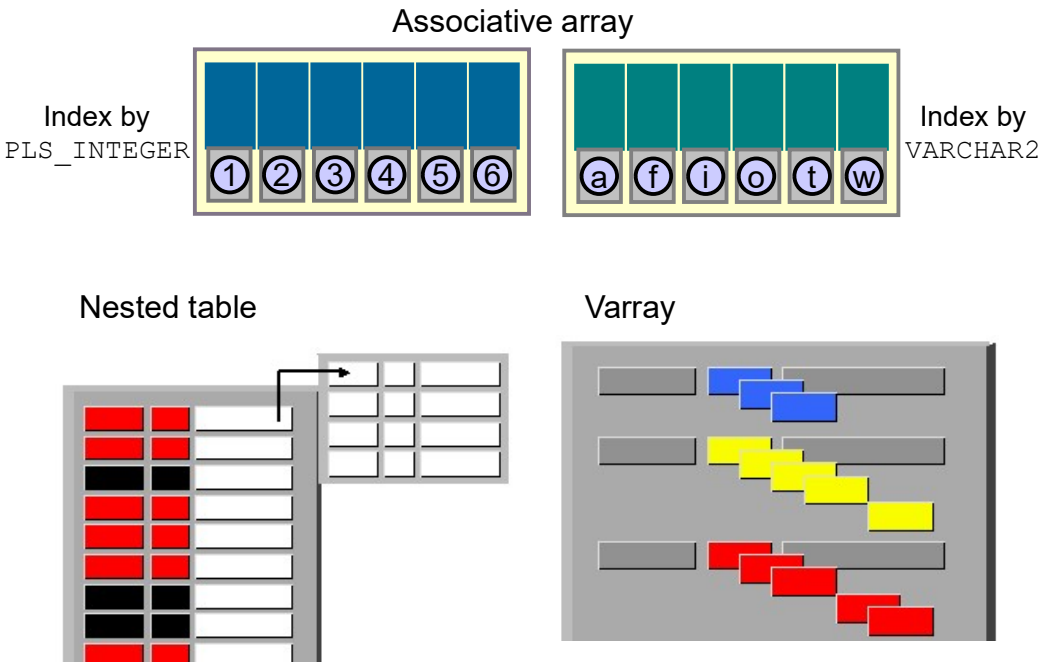
Example:

```
TYPE location_type IS VARRAY(3) OF locations.city%TYPE;
offices location_type;
```

The size of this VARRAY is restricted to 3. You can initialize a VARRAY by using constructors. If

you try to initialize the `VARRAY` with more than three elements, a “Subscript outside of limit” error message is displayed.

Summary of Collection Types



Associative Arrays

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 either integer- or character-based. The array value may be of the scalar data type (single value) or the record data type (multiple values).

Because associative arrays are intended for storing temporary data, you cannot use them with SQL statements such as `INSERT` and `SELECT INTO`.

Nested Tables

A nested table holds a set of values. That is, it is a table within a table. Nested tables are unbounded; that is, the size of the table can increase dynamically. Nested tables are available in both PL/SQL and the database. Within PL/SQL, nested tables are like one-dimensional arrays whose size can increase dynamically.

Varrays

Variable-size arrays, or varrays, are also collections of homogeneous elements that hold a

fixed number of elements (although you can change the number of elements at run time). They use sequential numbers as subscripts. You can define equivalent SQL types, thereby allowing varrays to be stored in database tables.

Quiz

Identify situations in which you can use the %ROWTYPE attribute.

a. When you are not sure about the structure of the underlying database table
b. When you want to retrieve an entire row from a table
c. When you want to declare a variable according to another previously declared variable or database column

Answer: a, b

Advantages of Using the %ROWTYPE Attribute

Use the %ROWTYPE attribute when you are not sure about the structure of the underlying database table.

The main advantage of using %ROWTYPE is that it simplifies maintenance. Using %ROWTYPE ensures that the data types of the variables declared with this attribute change dynamically when the underlying table is altered. If a DDL statement changes the columns in a table, the PL/SQL program unit is invalidated. When the program is recompiled, it automatically reflects the new table format.

The %ROWTYPE attribute is particularly useful when you want to retrieve an entire row from a table. In the absence of this attribute, you would be forced to declare a variable for each of the columns retrieved by the SELECT statement.

Summary

In this lesson, you should have learned to:

- Define and reference PL/SQL variables of composite data types
  - PL/SQL record
  - Associative array
    - INDEX BY table
    - INDEX BY table of records
- Define a PL/SQL record by using the %ROWTYPE attribute
- Compare and contrast the three PL/SQL collection types:
  - Associative array
  - Nested table
  - VARRAY

ORACLE

A PL/SQL record is a collection of individual fields that represent a row in a table. By using records, you can group the data into one structure, and then manipulate this structure as one entity or logical unit. This helps reduce coding and keeps the code easy to maintain and understand.

Like PL/SQL records, a PL/SQL collection is another composite data type. PL/SQL collections include:

- **Associative arrays (**also known as INDEX BY tables)**:** They are objects of TABLE type and look similar to database tables, but with a slight difference. The so-called INDEX BY tables use a primary key to give you array-like access to rows. The size of an associative array is unconstrained.
- **Nested tables:** The key for nested tables cannot have a negative value, unlike INDEX BY tables. The key must also be in a sequence.
- **Variable-size arrays** (VARRAY)**:** A VARRAY is similar to associative arrays, except that a VARRAY is constrained in size.