6

Working with Composite Data Types

ORACLE

Objectives

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

- **Create user-defined PL/SQL records**
- **Create a record with the `%ROWTYPE` attribute**
- **Create an `INDEX BY` table**
- **Create an `INDEX BY` table of records**
- **Describe the differences among records, tables, and tables of records**

ORACLE

Lesson Aim

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)**
- **Are of two types:**
  - **PL/SQL records**
  - **PL/SQL collections**
    - **`INDEX BY` tables or associative arrays**
    - **Nested table**
    - **`VARRAY`**

ORACLE

Composite Data Types

You have learned that variables of scalar data type can hold only one value, whereas a variable of composite data type can hold multiple values of scalar data type or 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:
  - `INDEX BY` tables or associative arrays
  - Nested table
  - `VARRAY`

Why Use Composite Data Types?

You have all the related data as a single unit. You can easily access and modify the 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.

Composite Data Types

- **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.**

ORACLE

Composite Data Types (continued)

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. If you create a record to hold employee details, 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

- **Must contain one or more components (called *fields*) of any scalar, `RECORD`, or `INDEX BY` table data type**
- **Are similar to structures in most 3GL 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**

ORACLE

PL/SQL Records

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 can also be used when defining 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

Creating a PL/SQL Record

PL/SQL records are user-defined composite types. To use them:

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 *field_type* or an initial value

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

`REF CURSOR` is covered in appendix C ("`REF` Cursors").

Creating a PL/SQL Record

Declare variables to store the name, job, and salary of a new employee.

Example

```
...
  TYPE emp_record_type IS RECORD
    (last_name   VARCHAR2(25),
     job_id      VARCHAR2(10),
     salary      NUMBER(8,2));
  emp_record     emp_record_type;
...
```

ORACLE

Creating a PL/SQL Record (continued)

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.

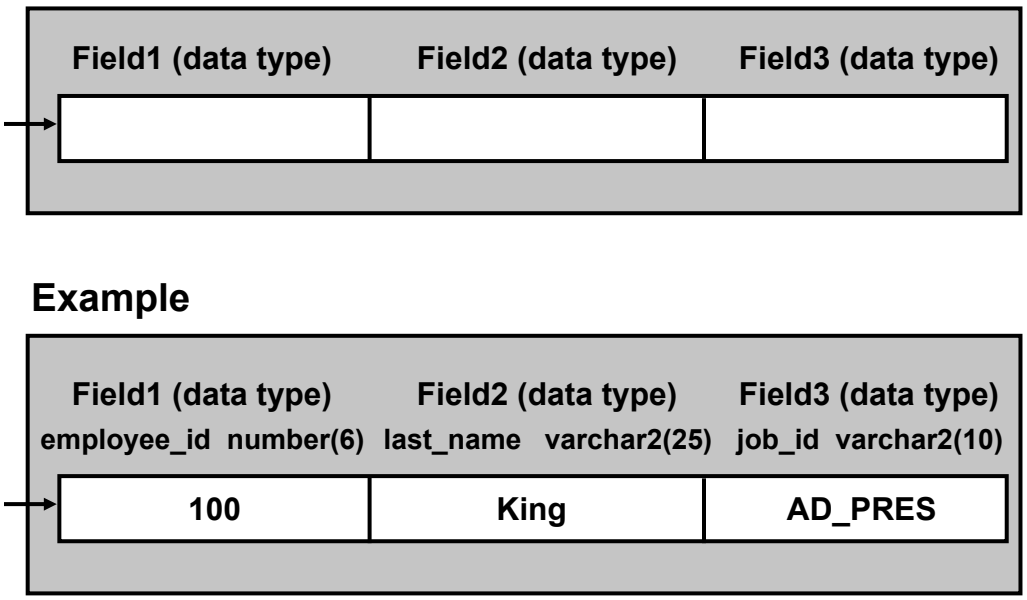
In the example in the slide, a record type (`emp_record_type`) is defined to hold the values for `last_name`, `job_id`, and `salary`. In the next step, a record (`emp_record`) of the type `emp_record_type` is declared.

The following example shows that you can use the `%TYPE` attribute to specify a field data type:

```
DECLARE
 TYPE emp_record_type IS RECORD
 (employee_id    NUMBER(6) NOT NULL := 100,
  last_name      employees.last_name%TYPE,
  job_id         employees.job_id%TYPE);
  emp_record     emp_record_type;
  ...
```

Note: You can add the `NOT NULL` constraint to any field declaration to prevent assigning nulls to that field. Remember that the fields declared as `NOT NULL` must be initialized.

PL/SQL Record Structure



PL/SQL Record Structure

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;
```


%ROWTYPE Attribute

You have learned that %TYPE is used to declare a variable of a 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 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;
 ...
```

%ROWTYPE Attribute (continued)

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,
salary              NUMBER(8,2),
commission_pct      NUMBER(2,2),
manager_id          NUMBER(6),
department_id       NUMBER(4))
```

To reference an individual field, use 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 user-defined record and a %ROWTYPE record never have the same data type.

Advantages of Using %ROWTYPE

- **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 retrieving a row with the SELECT * statement.**

Advantages of Using %ROWTYPE

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, then the PL/SQL program unit is invalidated. When the program is recompiled, it will automatically reflect 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.

%ROWTYPE Attribute

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

ORACLE

%ROWTYPE Attribute

An 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 record that is inserted into the retired_emps table is shown below:

EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
124	Mourgos	ST_MAN	100	16-NOV-99	26-JAN-04	5800		50

Inserting a Record by Using %ROWTYPE

```
...
DEFINE employee_number = 124
DECLARE
   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 emp_rec FROM employees
 WHERE  employee_id = &employee_number;
 INSERT INTO retired_emps VALUES emp_rec;
END;
/
SELECT * FROM retired_emps;
```

ORACLE

Inserting a Record by Using %ROWTYPE

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.

The create statement that creates `retired_emps` 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))
```

Examine the select statement in the slide. We 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 record that is inserted is shown below. (You will see how to update this in the next slide.)

EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
124	Mourgos	ST_MAN	100	16-NOV-99	16-NOV-99	5800		50

Updating a Row in a Table by Using a Record

```
SET SERVEROUTPUT ON
SET VERIFY OFF
DEFINE employee_number = 124
DECLARE
   emp_rec retired_emps%ROWTYPE;
BEGIN
 SELECT * INTO emp_rec FROM retired_emps;
 emp_rec.leavedate:=SYSDATE;
 UPDATE retired_emps SET ROW = emp_rec WHERE
  empno=&employee_number;
END;
/
SELECT * FROM retired_emps;
```

ORACLE

Updating a Row in a Table by Using a Record

You have learned to insert a row by using a record. This slide shows you how to update a row by using a record. The keyword `ROW` is used to represent the entire row. The code shown in the slide updates the `leavedate` of the employee. The record is updated.

EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
124	Mourgos	ST_MAN	100	16-NOV-99	27-JAN-04	5800		50

INDEX BY Tables or Associative Arrays

- **Are PL/SQL structures with two columns:**
  - **Primary key of integer or string data type**
  - **Column of scalar or record data type**
- **Are unconstrained in size. However, the size depends on the values that the key data type can hold.**

INDEX BY Tables or Associative Arrays

INDEX BY tables are composite types (collections) and are user defined. INDEX BY tables can store data using a primary key value as the index, where the key values are not sequential. INDEX BY tables are sets of key-value pairs. (You can imagine data stored in two columns, although the key and value pairs are not exactly stored in columns.)

INDEX BY tables have only two columns:

- A column of integer or string type that acts as the primary key. The key can be numeric, either BINARY_INTEGER or PLS_INTEGER. The BINARY_INTEGER and PLS_INTEGER keys require less storage than NUMBER. They are used to represent mathematical integers compactly and to implement arithmetic operations by using machine arithmetic. Arithmetic operations on these data types are faster than NUMBER arithmetic. The key can also be of type VARCHAR2 or one of its subtypes. The examples in this course use the PLS_INTEGER data type for the key column.
- A column of scalar or record data type to hold values. If the column is of scalar type, it can hold only one value. If the column is of record type, it can hold multiple values.

The INDEX BY tables are unconstrained in size. However, the key in the PLS_INTEGER column is restricted to the maximum value that a PLS_INTEGER can hold. Note that the keys can be both positive and negative. The keys in INDEX BY tables are not in sequence.

Creating an INDEX BY Table

Syntax:

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

Declare an INDEX BY table to store the last names of employees:

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

ORACLE

Creating an INDEX BY Table

There are two steps involved in creating an INDEX BY table.

1. Declare a TABLE data type.
2. Declare a variable of that data type.

In the syntax:

type_name	Is the name of the TABLE type (It is a type specifier used in subsequent declarations of PL/SQL table identifiers.)
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 PL/SQL table

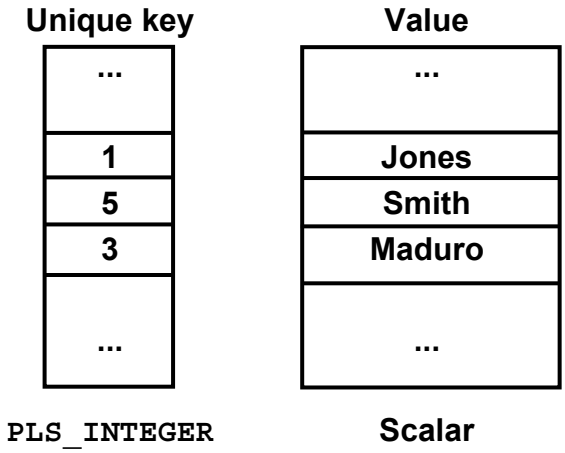
Creating an INDEX BY Table (continued)

The NOT NULL constraint prevents nulls from being assigned to the PL/SQL table of that type. Do not initialize the INDEX BY table.

INDEX BY tables can have the following element types: BINARY_INTEGER, BOOLEAN, LONG, LONG RAW, NATURAL, NATURALN, PLS_INTEGER, POSITIVE, POSITIVEN, SIGNTYPE, and STRING.

INDEX BY tables are not automatically populated when you create them. You must programmatically populate the INDEX BY tables in your PL/SQL programs and then use them.

INDEX BY Table Structure



ORACLE

INDEX BY Table Structure

Like the size of a database table, the size of an INDEX BY table is unconstrained. That is, the number of rows in an INDEX BY table can increase dynamically so that your INDEX BY table grows as new rows are added.

INDEX BY tables can have one column and a unique identifier to that column, neither of which can be named. The column can belong to any scalar or record data type, but the primary key must belong to the types PLS_INTEGER or BINARY_INTEGER. You cannot initialize an INDEX BY table in its declaration. An INDEX BY table is not populated at the time of declaration. It contains no keys or values. An explicit executable statement is required to populate the INDEX BY table.

Creating an INDEX BY Table

```
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;
/
```

ORACLE

Creating an INDEX BY Table

The example in the slide creates two INDEX BY tables.

Use the key of the INDEX BY table to access an element in the table.

Syntax:

```
INDEX_BY_table_name(index)
```

Here, index belongs to type PLS_INTEGER.

The following example shows how to reference the third row in an INDEX BY table called ename_table:

```
ename_table(3)
```

The magnitude range of a PLS_INTEGER is –2147483647 to 2147483647, so the primary key value can be negative. Indexing does not need to start with 1.

Note: 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.

Using INDEX BY Table Methods

The following methods make INDEX BY tables easier to use:

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

Using INDEX BY Table Methods

An INDEX BY table method is a built-in procedure or function that operates on a PL/SQL table and is called by using dot notation.

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

Method	Description
EXISTS(*n*)	Returns TRUE if the *n*th element in a PL/SQL table exists
COUNT	Returns the number of elements that a PL/SQL table currently contains
FIRST LAST	• Returns the first and last (smallest and largest) index numbers in a PL/SQL table • Returns NULL if the PL/SQL table is empty
PRIOR(*n*)	Returns the index number that precedes index *n* in a PL/SQL table
NEXT(*n*)	Returns the index number that succeeds index *n* in a PL/SQL table
DELETE	• DELETE removes all elements from a PL/SQL table. • DELETE(n) removes the *n*th element from a PL/SQL table. • DELETE(m, n) removes all elements in the range *m* ... *n* from a PL/SQL table.

INDEX BY Table of Records

Define an INDEX BY table variable to hold an entire row from a table.

Example

```
DECLARE
  TYPE dept_table_type IS TABLE OF
       departments%ROWTYPE
       INDEX BY PLS_INTEGER;
  dept_table dept_table_type;
  -- Each element of dept_table is a record
```

ORACLE

INDEX BY Table of Records

At any particular time, an INDEX BY table declared as a table of scalar data type can store the details of only one column in a database table. There is often a need to store all the columns retrieved by a query. The INDEX BY table of records offers a solution to this. Because only one table definition is needed to hold information about all the fields of a database table, the table of records greatly increases the functionality of INDEX BY tables.

Referencing a Table of Records

In the example in the slide, you can refer to fields in the dept_table record because each element of the table is a record.

Syntax:

```
table(index).field
```

Example:

```
dept_table(15).location_id := 1700;
```

location_id represents a field in dept_table.

Referencing a Table of Records (continued)

You can use the %ROWTYPE attribute to declare a record that represents a row in a database table. The differences between the %ROWTYPE attribute and the composite data type PL/SQL record include the following:

- 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. %ROWTYPE records are dynamic because the table structures are altered in the database.

INDEX BY Table of Records: Example

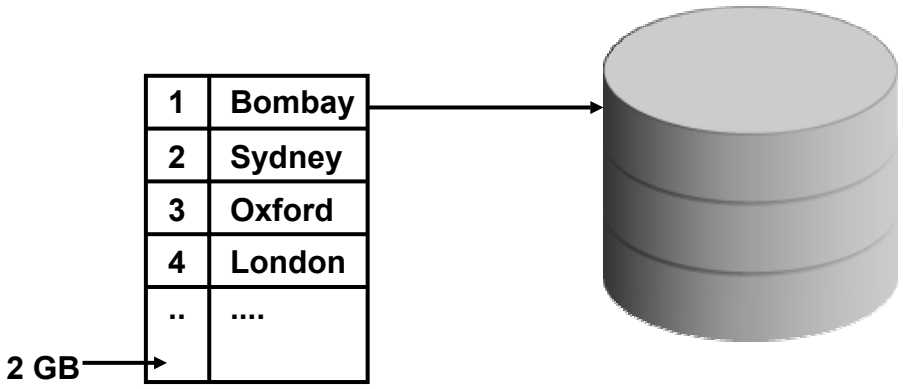
```
SET SERVEROUTPUT ON
DECLARE
   TYPE emp_table_type IS TABLE OF
      employees%ROWTYPE INDEX BY PLS_INTEGER;
   my_emp_table  emp_table_type;
   max_count         NUMBER(3):= 104;
BEGIN
  FOR i IN 100..max_count
  LOOP
   SELECT * INTO my_emp_table(i) FROM employees
   WHERE employee_id = i;
  END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

ORACLE

INDEX BY Table of Records: Example

The example in the slide declares an INDEX BY table of records emp_table_type to temporarily store the details of employees whose employee IDs are between 100 and 104. Using a loop, the information of the employees from the EMPLOYEES table is retrieved and stored in the INDEX BY table. Another loop is used to print the last names from the INDEX BY table. Note the use of the first and last methods in the example.

Nested Tables



ORACLE

Nested Tables

The functionality of nested tables is similar to that of `INDEX BY` tables; however, there are differences in the nested table implementation. The nested table is a valid data type in a schema-level table, but an `INDEX BY` table is not. The key type for nested tables is not `PLS_INTEGER`. The key cannot be a negative value (unlike in the `INDEX BY` table). Though we are referring to the first column as key, there is no key in a nested table. There is a column with numbers in sequence that is considered as the key column. 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. Nested tables can be stored in the database (unlike `INDEX BY` tables).

Syntax:

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

In Oracle Database 10*g*, nested tables can be compared for equality. You can check if an element exists in a nested table and also if a nested table is a subset of another.

Nested Tables (continued)

Example:

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

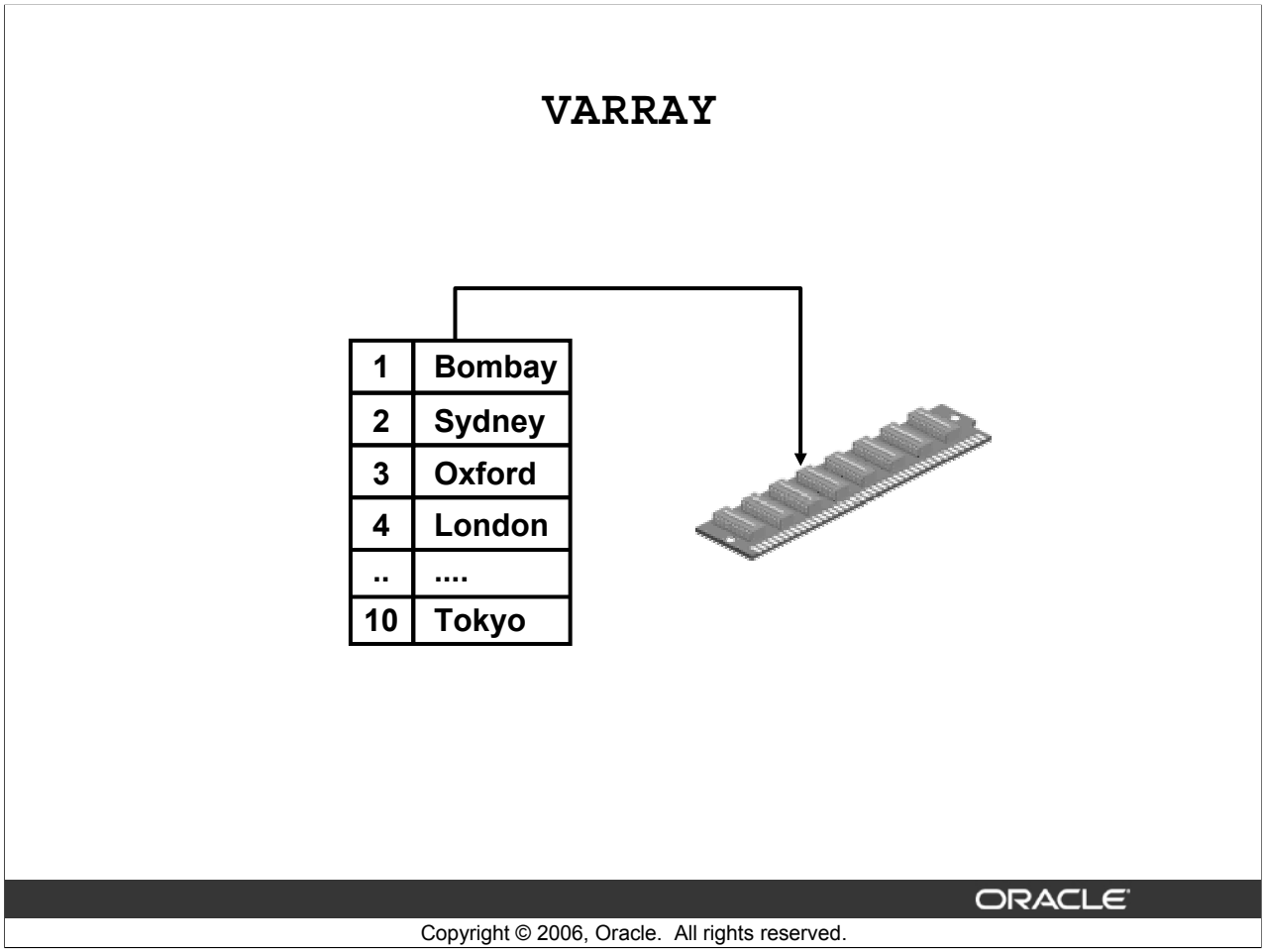
If you do not initialize an INDEX BY table, it is empty. If you do not initialize a nested table, it is automatically initialized to NULL. You can initialize the offices nested table by using a constructor:

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

Complete example:

```
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');
   table_count := offices.count();
   FOR i in 1..table_count LOOP
     DBMS_OUTPUT.PUT_LINE(offices(i));
   END LOOP;
END;
/
```

```
Bombay
Tokyo
Singapore
Oxford
PL/SQL procedure successfully completed.
```



VARRAY

A variable-size array (VARRAY) is similar to a PL/SQL table, except that a VARRAY is constrained in size. 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 the 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 contiguously in memory and not in the database. 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

In this lesson, you should have learned how to:

- **Define and reference PL/SQL variables of composite data types**
  - **PL/SQL record**
  - **`INDEX BY` table**
  - **`INDEX BY` table of records**
- **Define a PL/SQL record by using the `%ROWTYPE` attribute**

ORACLE

Summary

A PL/SQL record is a collection of individual fields that represent a row in the 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 easier to maintain and understand.

Like PL/SQL records, the table is another composite data type. `INDEX BY` tables are objects of `TABLE` type and look similar to database tables but with a slight difference. `INDEX BY` tables use a primary key to give you array-like access to rows. The size of an `INDEX BY` table is unconstrained. `INDEX BY` tables store a key and a value pair. The key column must be of the `PLS_INTEGER` or `BINARY_INTEGER` type; the column that holds the value can be of any data type.

The key type for nested tables is not `PLS_INTEGER`. The key cannot have a negative value, unlike the case with `INDEX BY` tables. The key must also be in a sequence.

Variable-size arrays (`VARRAYs`) are similar to PL/SQL tables, except that a `VARRAY` is constrained in size.

Practice 6: Overview

This practice covers the following topics:

- **Declaring `INDEX BY` tables**
- **Processing data by using `INDEX BY` tables**
- **Declaring a PL/SQL record**
- **Processing data by using a PL/SQL record**

ORACLE

Practice 6: Overview

In this practice, you define, create, and use `INDEX BY` tables and a PL/SQL record.

Practice 6

1. Write a PL/SQL block to print information about a given country.
   a. Declare a PL/SQL record based on the structure of the `countries` table.
   b. Use the `DEFINE` command to define a variable `countryid`. Assign CA to `countryid`. Pass the value to the PL/SQL block through an *i*SQL*Plus substitution variable.
   c. In the declarative section, use the `%ROWTYPE` attribute and declare the variable `country_record` of type `countries`.
   d. In the executable section, get all the information from the `countries` table by using `countryid`. Display selected information about the country. A sample output is shown below.

      ```
      Country Id: CA Country Name: Canada Region: 2
      PL/SQL procedure successfully completed.
      ```

   e. You may want to execute and test the PL/SQL block for the countries with the IDs DE, UK, US.
2. Create a PL/SQL block to retrieve the name of some departments from the `departments` table and print each department name on the screen, incorporating an `INDEX BY` table. Save the script as `lab_06_02_soln.sql`.
   a. Declare an `INDEX BY` table `dept_table_type` of type `departments.department_name`. Declare a variable `my_dept_table` of type `dept_table_type` to temporarily store the name of the departments.
   b. Declare two variables: `loop_count` and `deptno` of type `NUMBER`. Assign 10 to `loop_count` and 0 to `deptno`.
   c. Using a loop, retrieve the name of 10 departments and store the names in the `INDEX BY` table. Start with `department_id` 10. Increase `deptno` by 10 for every iteration of the loop. The following table shows the `department_id` for which you should retrieve the `department_name` and store in the `INDEX BY` table.

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
30	Purchasing
40	Human Resources
50	Shipping
60	IT
70	Public Relations
80	Sales
90	Executive
100	Finance

Practice 6 (continued)

d. Using another loop, retrieve the department names from the `INDEX BY` table and display them.

e. Execute and save your script as `lab_06_02_soln`.sql. The output is shown below.

```
Administration
Marketing
Purchasing
Human Resources
Shipping
IT
Public Relations
Sales
Executive
Finance
PL/SQL procedure successfully completed.
```

Practice 6 (continued)

3. Modify the block that you created in question 2 to retrieve all information about each department from the `departments` table and display the information. Use an `INDEX BY` table of records.
   a. Load the script `lab_06_02_soln.sql`.
   b. You have declared the `INDEX BY` table to be of type `departments.department_name`. Modify the declaration of the `INDEX BY` table, to temporarily store the number, name, and location of the departments. Use the `%ROWTYPE` attribute.
   c. Modify the select statement to retrieve all department information currently in the `departments` table and store it in the `INDEX BY` table.
   d. Using another loop, retrieve the department information from the `INDEX BY` table and display the information. A sample output is shown below.

```
Department Number: 10 Department Name: Administration Manager
Id: 200 Location Id: 1700
Department Number: 20 Department Name: Marketing Manager Id:
201 Location Id: 1800
Department Number: 30 Department Name: Purchasing Manager Id:
114 Location Id: 1700
Department Number: 40 Department Name: Human Resources
Manager Id: 203 Location Id: 2400
Department Number: 50 Department Name: Shipping Manager Id:
121 Location Id: 1500
Department Number: 60 Department Name: IT Manager Id: 103
Location Id: 1400
Department Number: 70 Department Name: Public Relations
Manager Id: 204 Location Id: 2700
Department Number: 80 Department Name: Sales Manager Id: 145
Location Id: 2500
Department Number: 90 Department Name: Executive Manager Id:
100 Location Id: 1700
Department Number: 100 Department Name: Finance Manager Id:
108 Location Id: 1700
PL/SQL procedure successfully completed.
```

4. Load the script `lab_05_03_soln.sql`.
   a. Look for the comment "DECLARE AN INDEX BY TABLE OF TYPE VARCHAR2(50). CALL IT ename_table_type" and include the declaration.
   b. Look for the comment "DECLARE A VARIABLE ename_table OF TYPE ename_table_type" and include the declaration.
   c. Save your script as `lab_06_04_soln.sql`.