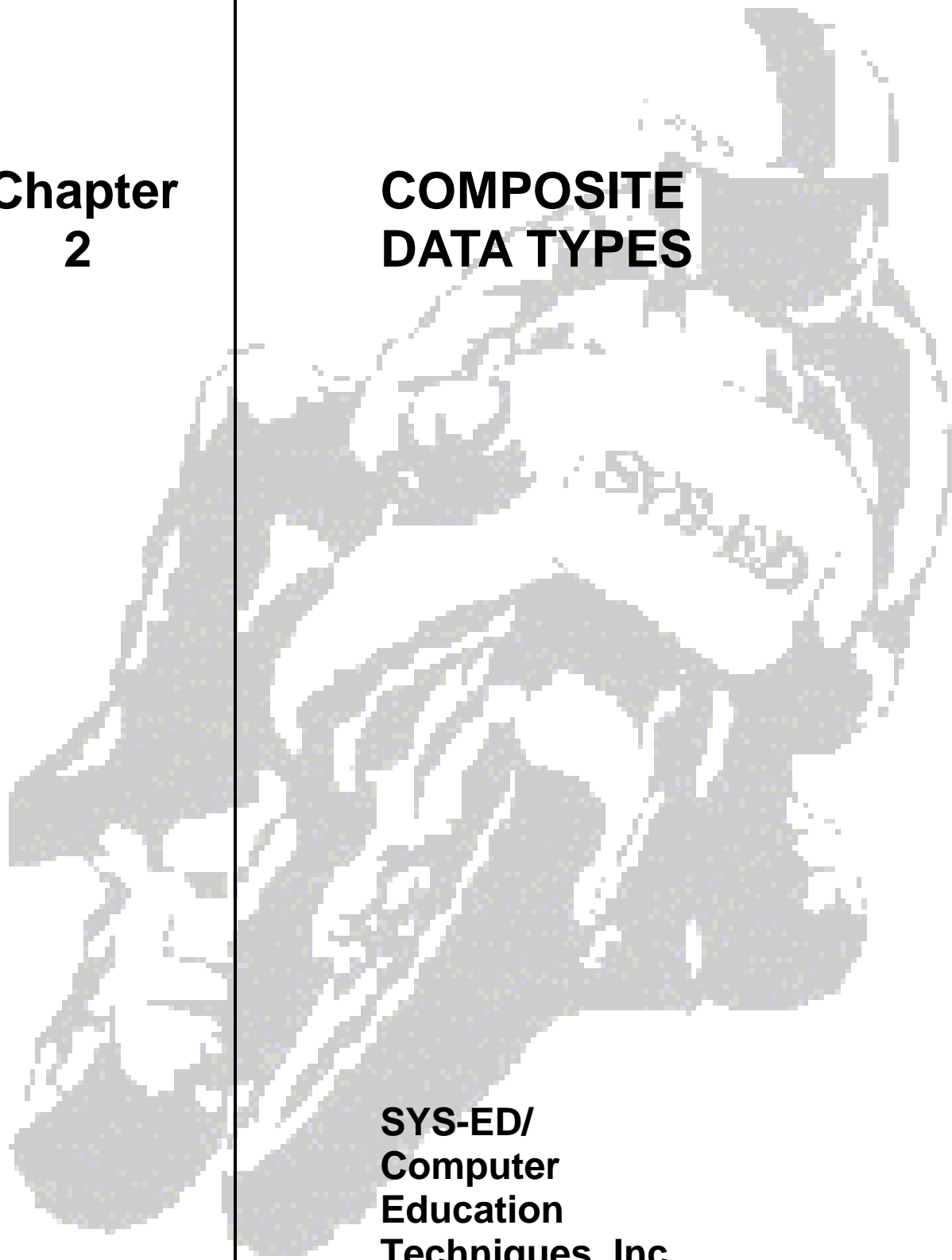


**Chapter
2**

**COMPOSITE
DATA TYPES**



**SYS-ED/
Computer
Education
Techniques, Inc.**

Objectives

You will learn:

- User-defined PL/SQL records.
- Creating a record with the %ROWTYPE attribute.
- Creating a INDEX BY table.
- Creating INDEX BY table of records.
- Difference between records, tables, and tables of records.

1 Composite Data Types

As with scalar variables, composite variables have a data type. Composite data types are also known as collections.

RECORD, TABLE, NESTED TABLE, and VARRAY are composite data types.

The RECORD data type are used to treat related, but dissimilar data as a logical unit. The TABLE data type are used to reference and manipulate collections of data as a whole object.

1.1 Types of Composite Data Types

Composite data types are of two types:

- PL/SQL RECORDS
- PL/SQL Collections -
 - INDEX BY Table
 - Nested Table
 - VARRAY

Composite data types contain internal components and they are reusable

1.2 RECORD and TABLE Data Types

A record is a group of related data items stored as fields, each with its own name and data type. A table contains a column and a primary key in order to provide array-like access to rows.

After tables and records are defined, they can be reused.

2 PL/SQL Records

A record is a group of related data items stored in fields, each with its own name and data type. When a record type for related fields is declared, they can be manipulated as a unit.

- 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.
- RECORD types and user-defined records can be declared in the declarative part of any block, subprogram, or package.
- One record can be nested inside of another record.

3 Creating a PL/SQL Record

Syntax:

```
TYPE type_name IS RECORD
    (field_declaration [, field_declaration]...);
identifier type_name;
```

Where field_declaration is:

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

To create a record, a RECORD type is defined and then records of that type are declared.

Keyword	Description
type_name	Name of the RECORD type. (This identifier is used to declare records.)
field_name	Name of a field within the record.
field_type	Data type of the field. It represents any PL/SQL data type except REF CURSOR. The %TYPE and %ROWTYPE attributes can be used.
expr	Field_type or an initial value.

Example 1:

```

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

```

Field declarations 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, the record type must be created before declaring an identifier using that type.

The EMP_RECORD_TYPE record type is defined to hold the values for the last_name, job_id, and salary. In the next step, a record EMP_RECORD, of the type EMP_RECORD_TYPE is declared.

Example 2:

```

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;
...

```

The %TYPE attribute can be used to specify a field data type.

The NOT NULL constraint can be added to any field declaration to prevent assigning nulls to that field. Fields declared as NOT NULL must be initialized.

4 PL/SQL Record Structure

Fields in a record are accessed by name. To reference or initialize an individual field, dot notation is used as in the following syntax:

```
record_name field_name
```

The `job_id` field in the `emp_record` record is referenced as follows:

```
emp_record.job_id
```

Values can be assigned to the record field as follows:

```
emprecord.job_id := 'PARKSL';
```

In a block or subprogram, user-defined records are instantiated when the block or subprogram is entered and cease to exist when out of the block or subprogram.

5 Declaring Records with the %ROWTYPE Attribute

To declare a record based on a collection of columns in a database table or view, the %ROWTYPE attribute is used. 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.

Example:

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

A record is declared using %ROWTYPE as a data type specifier.

The record, emp_record, will have a structure consisting of the following fields, each representing a column in the EMPLOYEES table.

```
(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))
```

Syntax:

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

Keyword	Description
identifier	Name chosen for the record as a whole.
reference	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.

5.1 Dot Notation

Dot notation is used for referencing an individual field:

```
record_name.field_name
```

Example:

```
emp_record.commission_pct
```

This code references the `commission_pct` field in the `emp_record` record.

A value is then assigned to the record field as follows:

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

5.2 Assigning Values to Records

A list of column values can be assigned to a record by using the `SELECT` or `FETCH` statement.

The column names must appear in the same order as the fields in the record. Also one record can be assigned to another if they have the same data type. A user-defined record and a `%ROWTYPE` record never have the same data type.

5.3 %ROWTYPE Advantages

The `%ROWTYPE` attribute is used when the structure of the underlying database table is not known.

This attribute ensures that the data types of the variables declared using this attribute change dynamically, in case the underlying table is altered. This attribute is particularly useful while retrieving an entire row from a table. In the absence of this attribute, the programmer would be forced to declare a variable for each of the columns retrieved by the `SELECT *` statement.

Examples 1:

```
dept_record departments%ROWTYPE,  
emp_record employees%ROWTYPE,
```

The first declaration creates a record with the same field names and field data types as a row in the DEPARTMENTS table. The fields are DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, and LOCATION ID.

The second declaration creates a record with the same field names, field data types, and order as a row in the EMPLOYEES table. The fields are EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE, JOB_ID, SALARY, COMMISSION PCT, MANAGER_ID, DEPARTMENT_ID.

Example 2:

```
DEFINE employee_number = 124  
DECLARE  
    emprec employees%ROWTYPE;  
BEGIN  
    SELECT * INTO emprec FROM employees  
    WHERE employee_id = &employee_number;  
    INSERT INTO retired_emps(empno, ename, lob, mgr,  
        hiredate, leavedate, sal, comm, deptno)  
    VALUES (emprec.employeeid, emprec.lastname, emprec.lob_id,  
        emprec.managerid, emprec.hiredate, SYSDATE,  
        emprec.salary, emprec.commission_pct,  
        emprec.departmentid);  
    COMMIT;  
END;
```

In this code, an employee is retiring. Information about a retired employee is added to a table that holds information about retired employees. The user supplies the employee's number. The record of the employee specified by the user is retrieved from the EMPLOYEES and stored into the emprec variable, which is declared using the %ROWTYPE attribute.

6 INDEX BY Tables

Objects of the TABLE type are called INDEX BY tables. They are modeled as database tables. INDEX BY tables use a primary key to provide you with array-like access to rows.

An INDEX BY table:

- is similar to an array.
- must contain two components:
 - A primary key of data type BINARY_INTEGER that indexes the INDEX BY table.
 - A column of a scalar or record data type, which stores the INDEX BY table elements.
- Can increase dynamically because it is unconstrained.

6.1 INDEX BY Table Structure

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

INDEX BY tables can have one column and a unique identifier to that one 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 type BINARY_INTEGER.

An INDEX BY table cannot be initialized at the time of its declaration. An INDEX BY table is not populated at the time of declaration. It contains no keys or no values. An explicit executable statement is required to initialize (populate) the INDEX BY table.

6.2 INDEX BY Table - Creation

Syntax:

```
DECLARE
    TYPE ename_table_type IS TABLE OF
        employees.last_name%TYPE
        INDEX BY BINARY_INTEGER;
    TYPE hiredate_table_type IS TABLE OF DATE
        INDEX BY BINARY_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;
```

6.3 Referencing an INDEX BY Table

Syntax:

```
INDEX_BY_table_name(primary_key_value)
```

where:

primary key value belongs to type BINARY_INTEGER.

Example:

```
ename_table(3) ...
```

This code references the third row in an INDEX BY table ENAME TABLE.

The magnitude range of a BINARY_INTEGER is -2147483647 ... 2147483647, so the primary key value can be negative.

Indexing does not need to start with 1

The table EXISTS (1) statement returns TRUE if a row with index 1 is returned. The EXISTS statement is used to prevent an error that is raised in reference to a non-existing table element.

6.4 INDEX BY Table Methods

The following methods are available for INDEX BY tables.

EXISTS	NEXT
COUNT	TRIM
FIRST	LAST
DELETE	PRIOR

6.5 INDEX BY Table Methods

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

Syntax:

`table_name.method_name[(parameters)]`

Method	Description
EXISTS(n)	Returns TRUE if the nth 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
TRIM	TRIM removes one element from the end of a PL/SQL table. TRIM(n) removes n elements from the end of a PL/SQL table.
DELETE	DELETE removes all elements from a PL/SQL table.
DELETE(n)	removes the nth element from a PL/SQL table.
DELETE(m, n)	removes all elements in the range m ... n from a

7 INDEX BY Table of Records

An INDEX BY table can store only the details of any one of the columns of a database table.

There is a requirement to store all the columns retrieved by a query. The INDEX BY table of records offer an approach to implementing this solution. Since only one table definition is needed to hold information about all of the fields of a database table, the table of records greatly increases the functionality of INDEX BY tables.

INDEX BY table of records:

- Define a TABLE variable with a permitted PL/SQL data type.
- Declare a PL/SQL variable to hold department information.

Example:

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

8 Referencing a Table of Records

Syntax:

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

Example 1:

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

Fields in the DEPT_TABLE record can be referenced because each element of this table is a record. LOCATION_ID represents a field in DEPT_TABLE.

The %ROWTYPE attribute can be used to declare a record that represents a row in a database table. The difference between the %ROWTYPE attribute and the composite data type RECORD is that RECORD allows the programmer to specify the data types of fields in the record or to declare user defined fields.

Example 2:

```
SET SERVEROUTPUT ON
DECLARE
    TYPE emp_table_type is table of
        employees%ROWTYPE INDEX BY BINARY_INTEGER;
    my_emp_table      emp_table_type;
    v_count           NUMBER (3) := 104;
BEGIN
    FOR i IN 100..v_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;
```

This code declares an INDEX BY table of records `emp_table_type` to temporarily store the details of the employees whose `EMPLOYEE_ID` lies 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 information regarding the last names from the INDEX BY table.