**Working with Records**

***By Steven Feuerstein Oracle ACE Director***

**Part 7 in a series of articles on understanding and using PL/SQL**

|  |
| --- |
| **Answers to the Challenge**  Here are the answers to the PL/SQL Challenge questions in last issue’s “Error Management” article:  **Answer 1:** Choices (a), (c), and (d) all raise ORA-00001, also known as DUP\_VAL\_ON\_INDEX inside PL/SQL code.  **Answer 2:** To make it possible for the plch\_proc procedure to compile without error, change “AND” to “OR” in the WHEN clause of the exception section. Because only one exception can be raised at a time in a session, it doesn’t make any sense to allow you to check for two (or more) exceptions in a single handler.  For full explanations of both of these answers, visit [plsqlchallenge.com](http://www.plsqlchallenge.com/), register or log in, and click the **Closed/Taken** tab in Play a Quiz. |

The Oracle PL/SQL language was designed to be a portable, high-performance transaction processing language that is tightly integrated with the SQL language. It is rare, indeed, to find a PL/SQL program that does not either read from or make changes to tables in a database. Tables are made up of rows of data, each consisting of one or more columns, so it stands to reason that Oracle Database would make it as easy as possible to work with those rows of data inside a PL/SQL program. And it does precisely that through its implementation of the ***record***.

A record is a ***composite*** datatype, which means that it can hold more than one piece of information, as compared to a ***scalar*** datatype, such as a number or string. It’s rare, in fact, that the data with which you are working is just a single value, so records and other composite datatypes are likely to figure prominently in your PL/SQL programs.

This article explores how you declare records, populate them with rows from a table, and even insert or change an entire row in a table by using a record. It also takes a look at ***user-defined*** record types, which enable you to work with records that are not necessarily related to a relational table.

**Declare a Record with %ROWTYPE**

PL/SQL makes it very easy to declare records that have the same structure as a table, a view, or the result set of a cursor by offering the %ROWTYPE attribute.

Suppose I have an employees table in an application that looks like this:

SQL> DESCRIBE omag\_employees

Name Null? Type

——————————— —————————— —————————————————

EMPLOYEE\_ID NOT NULL NUMBER(38)

LAST\_NAME VARCHAR2(100)

SALARY NUMBER

Each row in the table consists of three columns, and each column has its own datatype. The following query retrieves all the columns in all the rows in the table:

SELECT employee\_id, last\_name, salary

FROM omag\_employees

I want to write a block of code that retrieves a single row of data from omag\_employees for an employee ID and then work with the column values in that row. I ***could*** declare a variable for each column and then fetch into those variables, as follows:

CREATE PROCEDURE process\_employee (

employee\_id\_in IN

omag\_employees.employee\_id%TYPE)

IS

l\_employee\_id

omag\_employees.employee\_id%TYPE;

l\_last\_name

omag\_employees.last\_name%TYPE;

l\_salary

omag\_employees.salary%TYPE;

BEGIN

SELECT employee\_id,

last\_name,

salary

INTO l\_employee\_id,

l\_last\_name,

l\_salary

FROM omag\_employees

WHERE employee\_id = employee\_id\_in;

END;

(Note that I use suffixes in my parameters to indicate their mode. Here \_in indicates an IN parameter.)

That is, however, an awful lot of code to write, read, and maintain. A much better approach is to fetch that row of data into a record, and the best way to declare that record is as follows:

CREATE PROCEDURE process\_employee (

employee\_id\_in IN

omag\_employees.employee\_id%TYPE)

IS

l\_employee omag\_employees%ROWTYPE;

BEGIN

SELECT employee\_id,

last\_name,

salary

INTO l\_employee

FROM omag\_employees

WHERE employee\_id = employee\_id\_in;

END;

When this procedure is compiled, PL/SQL looks up the structure of the omag\_employees table and defines a record that has a field for each column in the table, with the same name and datatype. By using %ROWTYPE to declare the record, I also tell Oracle Database that this procedure ***depends on*** the omag\_employees table. If the database administrator changes the maximum length of the last\_name column to 200, for instance, this procedure’s status will be changed to INVALID. When the procedure is recompiled, the compiler will update the definition of the record in this procedure to match the table’s new structure.

I can even shorten things further and write

CREATE PROCEDURE process\_employee (

employee\_id\_in IN

omag\_employees.employee\_id%TYPE)

IS

l\_employee omag\_employees%ROWTYPE;

BEGIN

SELECT \*

INTO l\_employee

FROM omag\_employees

WHERE employee\_id = employee\_id\_in;

END;

The SELECT \* syntax tells Oracle Database to fetch all the columns in the table.

I can also use %ROWTYPE to declare a record that has the same structure as a SELECT statement in a cursor. This is especially helpful for fetching either a subset of columns from a table or columns from multiple tables. Here’s an example:

DECLARE

CURSOR no\_ids\_cur

IS

SELECT last\_name, salary

FROM omag\_employees;

l\_employee no\_ids\_cur%ROWTYPE;

(Note that I usually add a “\_cur” suffix to the names of my explicitly declared cursors.)

Whenever you are fetching data from a cursor into PL/SQL variables, you should declare a record based on that cursor with %ROWTYPE and fetch into that record. This way, when and if the SELECT list of the cursor changes, the number and type of fields in the record will change accordingly and everything will stay in sync.

**Working with Record Variables**

Once you have declared a record in your block, you can both read and change the record’s value. You can do this at the record level or by referencing individual fields of that record, with the same dot notation used in SQL to refer to the column of a table.

So if I declare a record as follows,

DECLARE

l\_employee omag\_employees%ROWTYPE;

I will be able to display the value of the last\_name field of l\_employee in the executable section of the block as follows:

DBMS\_OUTPUT.put\_line (

l\_employee.last\_name);

I can ***change*** the value of a field, using an assignment operator:

l\_employee.last\_name := 'Picasso';

I can also perform the following record-level operations:

1. Set a record to NULL. This simple assignment will set the values of all fields to NULL.

l\_employee := NULL;

1. Assign one record to another.

DECLARE

l\_employee1 omag\_employees%ROWTYPE;

l\_employee2 omag\_employees%ROWTYPE;

BEGIN

l\_employee1 := l\_employee2;

END;

**Populating Records with Data**

Most of the time when you work with records, you will be assigning a row from a table to a record. You can also, however, assign values directly to individual fields or even to the record as a whole by using the PL/SQL assignment operator (:=). Let’s look at examples of the ways to populate a record.

1. Declare a record with the same structures as those of the omag\_employees table and then fill the record with the contents of one row from that table:

DECLARE

l\_employee omag\_employees%ROWTYPE;

BEGIN

SELECT \*

INTO l\_employee

FROM omag\_employees

WHERE employee\_id = 100;

END;

1. Declare a cursor that fetches the last name and salary of all employees. Then use %ROWTYPE to declare a record that contains two fields: l\_employee.last\_name and l\_employee.salary. Finally, open the cursor, fetch one row into the record, and close the cursor.

DECLARE

CURSOR no\_ids\_cur

IS

SELECT last\_name, salary

FROM omag\_employees;

l\_employee no\_ids\_cur%ROWTYPE;

BEGIN

OPEN no\_ids\_cur;

FETCH no\_ids\_cur INTO l\_employee;

CLOSE no\_ids\_cur;

END;

/

1. Populate a record by using native dynamic SQL. (Note: the SELECT statement is not dynamic; this is just to show that it is possible to populate a record with an EXECUTE IMMEDIATE . . . INTO statement).

DECLARE

l\_employee omag\_employees%ROWTYPE;

BEGIN

EXECUTE IMMEDIATE

'SELECT \* FROM omag\_employees'

INTO l\_employee;

END;

1. Populate the fields of a record by using assignments.

DECLARE

l\_employee omag\_employees%ROWTYPE;

BEGIN

l\_employee.last\_name := 'Renoir';

l\_employee.salary := 1500;

END;

Note that even though I define the record based on the table, I do not have to set the values of the record’s fields from the table. I might, for example, want to insert a ***new*** row into the employees table by using the record (see “Inserting and Updating with Records” for details).

1. Assign one record to another. Oracle Database supports record-level assignments, even the assignment of NULL to a record.

DECLARE

l\_old\_employee omag\_employees%ROWTYPE;

l\_new\_employee omag\_employees%ROWTYPE;

BEGIN

l\_new\_employee := l\_old\_employee;

l\_old\_employee := NULL;

END;

**Cursor FOR Loops and Implicitly Declared Records**

Suppose I want to write a program to display the last names of all employees. An elegant and simple way to do this in PL/SQL is to take advantage of the cursor FOR loop (which I discussed in part 2 of this PL/SQL 101 series). The cursor FOR loop is a variation on the numeric FOR loop, which looks like this:

FOR index IN low\_value .. high\_value

LOOP

loop\_body\_statements

END LOOP;

The index is implicitly declared by Oracle Database as an integer and can be referenced only inside the body of this loop.

A cursor FOR loop has a similar structure but replaces a numeric range with a query:

FOR index IN ( SELECT\_statement )

LOOP

loop\_body\_statements

END LOOP;

Oracle Database also implicitly declares ***this*** loop index as well, but in the case of a cursor FOR loop, it declares the index as a record by using %ROWTYPE against the query in the loop header.

The following block uses a cursor FOR loop to fetch only the last name of each employee, deposit that name into a record, and then display the value of the last\_name field of that record:

BEGIN

FOR employee\_rec

IN (SELECT last\_name

FROM omag\_employees

ORDER BY last\_name)

LOOP

DBMS\_OUTPUT.put\_line (

employee\_rec.last\_name);

END LOOP;

END;

/

**Passing Records as Parameters**

You can define parameters based on record types, and you can therefore pass records as arguments to subprograms. Suppose I need to write a procedure that displays an employee. I could implement it as follows:

CREATE PROCEDURE show\_employee (

employee\_id\_in IN

omag\_employees.employee\_id%TYPE,

last\_name\_in IN

omag\_employees.last\_name%TYPE,

salary\_in IN

omag\_employees.salary%TYPE)

IS

BEGIN

DBMS\_OUTPUT.put\_line (

employee\_id\_in

|| '-'

|| last\_name\_in

|| '-'

|| salary\_in);

END;

But I can also avoid having to declare each of those individual parameters (imagine a 100-column table!) by passing a record:

CREATE PROCEDURE show\_employee (

employee\_in IN

omag\_employees%ROWTYPE)

IS

BEGIN

DBMS\_OUTPUT.put\_line (

employee\_in.employee\_id

|| '-'

|| employee\_in.last\_name

|| '-'

|| employee\_in.salary);

END;

/

Of course, as new columns are added to the table, their contents will not automatically be displayed by this procedure. So when you use %ROWTYPE to pass arguments to subprograms, make sure to review the subprogram logic after any change to the table.

**Inserting and Updating with Records**

As you have seen, PL/SQL makes it very easy to populate a record from a row in a table. But what if you want to change the contents of a row in a table by using a record? PL/SQL offers special syntax in both the INSERT and UPDATE statements so that you can easily use records to perform those data manipulation language (DML) operations as well.

The most common form of an INSERT statement is

INSERT INTO table\_name (column\_list)

VALUES (expression\_list)

where column\_list is the list of columns that will be populated on insert and expression\_list is the list of expressions that will be assigned to their respective columns.

If I want to provide a value for each column in a table that has, say, 500 columns, writing and managing that code can become quite tedious. Inserting with a record comes in very handy in such a scenario.

**Code Listing 1:** Insert of a single row with each column specified

DECLARE

l\_employee\_id omag\_employees.employee\_id%TYPE

:= 500;

l\_last\_name omag\_employees.last\_name%TYPE

:= 'Mondrian';

l\_salary omag\_employees.salary%TYPE

:= 2000;

BEGIN

INSERT

INTO omag\_employees (employee\_id,

last\_name,

salary)

VALUES (

l\_employee\_id,

l\_last\_name,

l\_salary);

END;

To perform a record-level insert, simply leave off the parentheses around the record in the VALUES clause. Listing 1 demonstrates an insert of a single row into the omag\_employees table that specifies each column individually. The following demonstrates the same insert, using a record:

DECLARE

l\_employee omag\_employees%ROWTYPE;

BEGIN

l\_employee.employee\_id := 500;

l\_employee.last\_name := ‘Mondrian’;

l\_employee.salary := 2000;

INSERT

INTO omag\_employees

VALUES l\_employee;

END;

/

So if you ever find yourself typing what feels like an endless list of variables in the VALUES clause of your INSERT statement, try using a record instead.

For updates, use SET ROW to update all the columns in a row from the record:

DECLARE

l\_employee omag\_employees%ROWTYPE;

BEGIN

l\_employee.employee\_id := 500;

l\_employee.last\_name := 'Mondrian';

l\_employee.salary := 2000;

UPDATE omag\_employees

SET ROW = l\_employee

WHERE employee\_id = 100;

END;

Remember: this UPDATE sets the value of every column in the table, including your primary key, so you should use the SET ROW syntax with great care.

**User-Defined Record Types**

So far you’ve seen how to declare a record variable based on a table or a cursor by using the %ROWTYPE attribute. You can also declare your own, user-defined record types by using the TYPE. . . RECORD statement.

User-defined record types come in handy when you find yourself declaring “sets” of individual variables, such as

DECLARE

l\_name1 VARCHAR2 (100);

l\_total\_sales1 NUMBER;

l\_deliver\_pref1 VARCHAR2 (10);

--

l\_name2 VARCHAR2 (100);

l\_total\_sales2 NUMBER;

l\_deliver\_pref2 VARCHAR2 (10);

Instead, why not create your own record type and then declare two records:

DECLARE

TYPE customer\_info\_rt IS RECORD

(

name VARCHAR2 (100),

total\_sales NUMBER,

deliver\_pref VARCHAR2 (10)

);

l\_customer1 customer\_info\_rt;

l\_customer2 customer\_info\_rt;

(Note that when I declare types, I use a root “t” suffix and then add the “type of type.” Here I added “\_rt” for ***record type***.)

With this approach, you do more than avoid writing repetitive statements. You also document that those three pieces of information are all related to a customer. And once you’ve “moved up” to using a record, you can pass that record as an argument or perform record-level operations, further reducing the volume of code needed to implement your requirements.

|  |
| --- |
| **Next Steps**  **DOWNLOAD** [**Oracle Database 11*g***](http://www.oracle.com/technetwork/database/enterprise-edition/downloads/index.html)  **TEST** [**your PL/SQL knowledge**](http://www.plsqlchallenge.com/)  **READ** [**PL/SQL 101, Parts 1-6**](http://www.oracle.com/technetwork/issue-archive/index-087690.html) |

Another excellent time to use the TYPE . . . RECORD statement to create your own record type is when a field of your record needs to be a PL/SQL-specific type, such as BOOLEAN. If you use %ROWTYPE, the datatypes of all the fields will be constrained to SQL types.

Here’s an example of a record type that contains two Boolean fields:

DECLARE

TYPE user\_preferences\_rt IS RECORD

(

show\_full\_name BOOLEAN,

autologin BOOLEAN

);

l\_user user\_preferences\_rt;

Records are, themselves, PL/SQL-specific datatypes, so another nice feature of user-defined record types is that you can define a record type as a field in ***another*** record type. In the declaration section below, I have created one record type that holds the different numeric elements that make up a telephone number. I then create another record to hold the various telephone numbers for a salesperson:

DECLARE

TYPE phone\_rt IS RECORD

(

area\_code PLS\_INTEGER,

exchange PLS\_INTEGER,

phn\_number PLS\_INTEGER,

extension PLS\_INTEGER

);

TYPE contact\_rt IS RECORD

(

day\_phone# phone\_rt,

eve\_phone# phone\_rt,

cell\_phone# phone\_rt

);

l\_sales\_rep contact\_rt;

**Composite Datatypes to the Rescue!**

PL/SQL’s support for records, one of several composite datatypes, enables you to write code that is simple, clean, and easy to maintain. Rather than work with long lists of variables or parameters, you can work with a record that ***contains*** all that information. User-defined records offer the flexibility to construct your own composite datatype, reflecting program-specific requirements that may not be represented by a relational table.

In the next article in this PL/SQL 101 series, I will explore another key composite datatype, the collection. Collections, PL/SQL’s implementation of arraylike structures, are used in some of the most important performance-related PL/SQL features, including FORALL and BULK COLLECT.

|  |
| --- |
| **Pseudorecords in Database Triggers**    Row-level triggers defined on tables can reference ***pseudorecords*** named NEW and OLD (you can override these default names with the REFERENCING clause of the trigger). They are called ***pseudo***records because they are similar in structure to a record defined on a table with %ROWTYPE but are restricted in their usage.  Both of the pseudorecords contain a field for every column in the table on which the trigger is defined. When you execute an INSERT or UPDATE statement, the NEW pseudorecord’s fields contain the “post” values of the columns (the values after the INSERT or UPDATE has taken place).  When you execute a DELETE or UPDATE statement, the OLD pseudorecord’s fields contain the “pre” values of the columns—how the row looks before the statement executes.  I can, for example, use pseudorecords to validate business rules, determine whether a column value has changed, and more. In the following trigger, I enforce a salary freeze; no one is allowed to get a raise during these tough economic times:    CREATE OR REPLACE TRIGGER  omag\_employees\_freeze\_trg  BEFORE INSERT  ON omag\_employees  FOR EACH ROW  DECLARE  BEGIN  IF :NEW.salary > :OLD.salary  THEN  RAISE\_APPLICATION\_ERROR (  -20000,  'Salary freeze in effect: '||  ' no increases allowed!');  END IF;  END omag\_employees\_freeze\_trg;  There are, however, record features that do not apply to pseudorecords. I cannot, for example, pass a pseudorecord as an argument to a subprogram, ***even*** if the parameter for that subprogram is defined as tablename%ROWTYPE, where tablename is the name of the table that causes the trigger to be fired. |

|  |
| --- |
| **Take the Challenge!**  Each PL/SQL 101 article offers a quiz to test your knowledge of the information provided in the article. The quiz questions appear below and also at PL/SQL Challenge ([plsqlchallenge.com](http://www.plsqlchallenge.com/)), a Website that offers online quizzes on the PL/SQL language. You can read and take the quiz here in ***Oracle Magazine*** and then check your answers in the next issue. If, however, you take the quiz at PL/SQL Challenge, you will be entered into a raffle to win an e-book from O’Reilly Media ([oreilly.com](http://www.oreilly.com/)).    I create and populate this table:  CREATE TABLE plch\_parts  (  partnum INTEGER PRIMARY KEY,  partname VARCHAR2 (100) UNIQUE  )  /  BEGIN  INSERT INTO plch\_parts  VALUES (100, 'Keyboard');  COMMIT;  END;  /  **Question**  Which choices contain code I can use in place of the /\*DECLARE\*/ comment in the following block so that after the resulting block executes, “Keyboard” will be displayed?    DECLARE  /\*DECLARE\*/  BEGIN  SELECT \*  INTO l\_part  FROM plch\_parts  WHERE partnum = 100;  DBMS\_OUTPUT.put\_line  (l\_part.partname);  END;  /     1. l\_part plch\_parts%TYPE; 2. \_part plch\_parts; 3. l\_part plch\_parts%ROWTYPE; 4. >CURSOR parts\_cur 5. IS 6. SELECT \* FROM plch\_parts; 7. l\_part parts\_cur%ROWTYPE; |