



Using PL/SQL Records in SQL Statements

White Paper

© Copyright Quest® Software, Inc. 2005. All rights reserved.

The information in this publication is furnished for information use only, does not constitute a commitment from Quest Software Inc. of any features or functions discussed and is subject to change without notice. Quest Software, Inc. assumes no responsibility or liability for any errors or inaccuracies that may appear in this publication.

Last revised: June 2005

TABLE OF CONTENTS

ABSTRACT.....	4
OVERVIEW.....	5
ORACLE9/ RELEASE 2 RECORD IMPROVEMENTS	8
SELECT WITH RECORD BIND.....	9
INSERT WITH RECORD BIND.....	12
UPDATE SET ROW WITH RECORD BIND	14
DELETE AND UPDATE WITH RETURNING WITH RECORD BIND.....	16
PERFORMANCE IMPACT OF RECORD BINDING.....	18
RECORDS: THE WAY TO GO	19
ABOUT THE AUTHORS.....	20
ABOUT QUEST SOFTWARE, INC.	21
CONTACTING QUEST SOFTWARE	21
TRADEMARKS.....	21

ABSTRACT

The material in this article is based on an Oracle Corporation white paper originally prepared by Bryn Llewellyn for OracleWorld Copenhagen in June 2002.

OVERVIEW

The PL/SQL RECORD datatype has been available for many years; however, its usefulness was limited because it wasn't possible to use records inside SQL statements (such as inserting a record directly into a table). Oracle9i Release 2 corrects this deficiency. It's now possible to employ records in UPDATE, INSERT, DELETE, and SELECT statements. In this article, Steven Feuerstein and Bryn Llewellyn demonstrate all of these new capabilities, as part of their continuing series on new PL/SQL features in Oracle9i Release 2.

A PL/SQL RECORD is a composite datatype. In contrast to a scalar datatype like NUMBER, a record is composed of multiple pieces of information, called fields. Records can be declared using relational tables or explicit cursors as "templates" with the %ROWTYPE declaration attribute. You can also declare records based on TYPEs that you define yourself. Records are very handy constructs for PL/SQL developers.

The easiest way to define a record is by using the %ROWTYPE syntax in your declaration. For example, the following statement:

```
DECLARE
    bestseller books%ROWTYPE;
```

creates a record that has a structure corresponding to the books table; for every column in the table, there's a field in the record with the same name and datatype as the column. The %ROWTYPE keyword is especially valuable because the declaration is guaranteed to match the corresponding schema-level template and is immune to schema-level changes in definition of the shape of the table. If we change the structure of the books table, all we have to do is recompile the preceding code and bestseller will take on the new structure of that table.

A second way to declare a record is to define your own RECORD TYPE. One advantage of a user-defined TYPE is that you can take advantage of native PL/SQL datatypes as well as derived values in the field list, as shown here:

```
DECLARE
    TYPE extra_book_info_t
        IS RECORD (
            title books.title%TYPE,
            is_bestseller BOOLEAN,
            reviewed_by names_list
        );
    first_book extra_book_info_t;
```

Notice that the preceding user-defined record datatype includes a field ("title") that's based on the column definition of a database table, a field ("is_bestseller") based on a scalar data type (PL/SQL Boolean flag), and a collection (list of names of people who reviewed *Oracle PL/SQL Programming, 3rd Edition*, due out in September 2002).

Next, we can declare a record based on this type (you don't use %ROWTYPE in this case, because you're already referencing a type to perform the declaration). Once you've declared a record, you can then manipulate the data in these fields (or the record as a whole) as you can see here:

```
DECLARE
    bestseller books%ROWTYPE;
    required_reading books%ROWTYPE;
BEGIN
    -- Modify a field value
    bestseller.title :=
        'ORACLE PL/SQL PROGRAMMING';

    -- Copy one record to another
    required_reading :=
        bestseller;
END;
```

Note that in the preceding code we've used the structure of the books table to define our PL/SQL records, but the assignment to the title field didn't in any way affect data inside that table. You should also be aware that while you can assign one record to another, you couldn't perform comparisons or computations on records. Neither of these statements will compile:

```
BEGIN
    IF bestseller =
        required_reading
    THEN ...
```

```
BEGIN
    left_to_read :=
        bestseller -
        required_reading;
```

You can also pass records as arguments to procedures and functions. This technique allows you to shrink down the size of a parameter list (pass a single record instead of a lengthy and cumbersome list of individual values). And if you're using %ROWTYPE to declare the argument, the "shape" of the record (numbers and types of fields) will adjust automatically with changes to the underlying cursor or table. Here's an example of a function with a record in the parameter list:

```
CREATE OR REPLACE PROCEDURE
    calculate_royalties (
        book_in IN books%ROWTYPE,
        quarter_end_in IN DATE
    )
IS ...
```

Prior to Oracle9i Release 2, it was only possible to use a record in conjunction with a SQL statement in one way: on the receiving end of a SELECT INTO or FETCH INTO statement. For example:

```
DECLARE
    bestseller books%ROWTYPE;
BEGIN
    SELECT *
        INTO bestseller
        FROM books
        WHERE title =
            'ORACLE PL/SQL PROGRAMMING';
END;
```

This is very convenient syntax, but it unfortunately just leaves us all hungry for the full range of record-smart SQL, most importantly the ability to perform INSERT and UPDATE operations with a record (as opposed to having to "break out" all the individual fields of that record). In summary, before Oracle9i Release 2, records offered significant advantages for developers, but also left us frustrated because of the limitations on their usage. Oracle9i Release 2 goes a long way in relieving (but not completely curing us of) our frustrations.

ORACLE9/ RELEASE 2 RECORD IMPROVEMENTS

In response to developer requests, Oracle has now made it possible for us to do any of the following with static SQL (i.e., SQL statements that are fully specified at the time your code is compiled):

- Use collections of records as the target in a BULK COLLECT INTO statement. You no longer need to fetch into a series of individual, scalar-type collections.
- Insert a row into a table using a record. You no longer need to list the individual fields in the record separately, matching them up with the columns in the table.
- Update a row in a table using a record. You can now take advantage of the special SET ROW syntax to update the entire row with the contents of a record with a minimum of typing.
- Use a record to retrieve information from the RETURNING clause of an UPDATE, DELETE, or INSERT.

Some restrictions do remain at Version 9.2.0 for records in SQL, including:

- You can't use the EXECUTE IMMEDIATE statement (Native Dynamic SQL) in connection with record-based INSERT, UPDATE, or DELETE statements. (It's supported for SELECT, as stated earlier.)
- With DELETE and UPDATE...RETURNING, the column-list must be written explicitly in the SQL statement.
- In the bulk syntax case, you can't reference fields of the in-bind table of records elsewhere in the SQL statement (especially in the where clause).

But why dwell on the negative? Let's explore this great new functionality with a series of examples, all of which will rely on the employees table, defined in the hr schema that's installed in the seed database. The script to create this schema is demo/schema/human_resources/hr_cre.sql under the Oracle Home directory.

The samples also rely on common features such as an index-by-*_integer table, records of employees%rowtype and a procedure to show the rows of such a table. These are implemented in the Emp_Util package. The scripts to create the working tables and the utility package are supplied in the [Download file](#).

SELECT WITH RECORD BIND

As we noted earlier, while it was possible before 9.2.0 to SELECT INTO a record, you couldn't BULK SELECT INTO a collection of records. The resulting code was often very tedious to write and not as efficient as would be desired. Suppose, for example, that we'd like to retrieve all employees hired before June 25, 1997, and then give them all big, fat raises. A very straightforward way to write the logic for this is shown in *Listing 1*.

Listing 1. Give raises to employees using single row fetches.

```
DECLARE
    v_emprec      employees%ROWTYPE;
    v_emprecs    emp_util.emprec_tab_t;

    CURSOR cur
    IS
        SELECT *
          FROM employees
         WHERE hire_date < TO_DATE(
              '25-JUN-1997', 'DD-MON-YYYY');

    i BINARY_INTEGER := 0;
BEGIN
    OPEN cur;

    LOOP
        FETCH cur INTO v_emprec;
        EXIT WHEN cur%NOTFOUND OR cur%ROWCOUNT > 10;
        i := i + 1;
        v_emprecs (i) := v_emprec;
    END LOOP;

    emp_util.give_raise (v_emprecs);
END;
```

There's no problem understanding this logic, but depending on the quantity of data involved, this could be a very inefficient implementation. We'd really love to take advantage of the recent (Oracle8i) addition of the BULK COLLECT syntax (allowing us to fetch multiple rows with a single pass to the database); we might see an order of magnitude improvement.

To use BULK COLLECT with records prior to Oracle9i Release 2, however, we'd need to select each element in the select list into its own collection; this technique is shown in *Listing 2*. The complete code for this block may be seen in `bulkcollect8i.sql` and is more than 80 lines long! It's approaching what is feasible to maintain, and feels especially uncomfortable because of the artificial requirement to compromise the natural modeling approach by slicing the desired table of records vertically into N tables of scalars.

Listing 2. BULK COLLECT into separate collections.

```
DECLARE
  TYPE employee_ids_t IS
    TABLE OF employees.employee_id%TYPE
    INDEX BY BINARY_INTEGER;
  ...
  v_employee_ids  employee_ids_t;
  ...
  v_emprecs emp_util.emprec_tab_t;

  CURSOR cur
  IS
    SELECT employee_id,
           ...
    FROM employees
    WHERE hire_date >= TO_DATE(
      '25-JUN-1997', 'DD-MON-YYYY');

BEGIN
  OPEN cur;
  FETCH cur BULK COLLECT
    INTO v_employee_ids,
         ...
    LIMIT 10;
  CLOSE cur;

  FOR j IN 1 .. v_employee_ids.LAST
  LOOP
    v_emprecs (j).employee_id :=
      v_employee_ids (j);
    ...
  END LOOP;

  emp_util.give_raise (v_emprecs);
END;
```



The clause limit 10 is equivalent to where rownum <= 10.

With Oracle9i Release 2, our program becomes much shorter, intuitive, and maintainable. What you see here is all we need to write to take advantage of BULK COLLECT to populate a single associative array of records:

```
DECLARE
    v_emprecs
        emp_util.emprec_tab_t;

    CURSOR cur
    IS
        SELECT *
            FROM employees
            WHERE hire_date < '25-JUN-97';
BEGIN
    OPEN cur;
    FETCH cur BULK COLLECT
        INTO v_emprecs LIMIT 10;
    CLOSE cur;
    emp_util.give_raise (v_emprecs);
END;
```



Once again, the clause *limit 10* is equivalent to *where rownum <= 10*.

Even more wonderful, we can now combine BULK COLLECT fetches into records with native dynamic SQL. Here's an example, in which we give raises to employees for a specific schema:

```
CREATE OR REPLACE PROCEDURE
    give_raise (schema_in IN VARCHAR2)
IS
    v_emprecs
        emp_util.emprec_tab_t;

    cur SYS_REFCURSOR;
BEGIN
    OPEN cur FOR
        'SELECT * FROM ' ||
        schema_in || '.employees' ||
        'WHERE hire_date < :date_limit'
        USING '25-JUN-97';

    FETCH cur BULK COLLECT
        INTO v_emprecs LIMIT 10;

    CLOSE cur;
    emp_util.give_raise (
        schema_in, v_emprecs);
END;
```

SYS_REFCURSOR is a pre-defined weak REF CURSOR type that was added to the PL/SQL language in Oracle9i Release 1.

INSERT WITH RECORD BIND

PL/SQL developers are demanding, no doubt about that. Even though Oracle can add all sorts of cool, new functionality into PL/SQL, we'll still find something missing, something else we so dearly need. For years, one of our favorite "wish-we-had's" was the ability to insert a row into a table using a record. Prior to Oracle9i Release 2, if we had put our data into a record, it would then be necessary to "explode" the record into its individual fields when performing the insert, as in:

```
DECLARE
    v_emprec employees%ROWTYPE
        := emp_util.get_one_row;
BEGIN
    INSERT INTO employees_retired (
        employee_id,
        last_name,
        ...)
    VALUES (
        v_emprec.employee_id,
        v_emprec.last_name,
        ...);
END;
```

This is very cumbersome coding; it certainly is something we would have liked to avoid. In Oracle9i Release 2, we can now take advantage of simple, intuitive, and compact syntax to bind an entire record to a row in an insert. This is shown here:

```
DECLARE
    v_emprec employees%rowtype
        := Emp_Util.Get_One_Row;
BEGIN
    INSERT INTO employees_retired
        VALUES v_emprec;
END;
```

Notice that we don't put the record inside parentheses. You are, unfortunately, not able to use this technique with Native Dynamic SQL. You can, on the other hand, insert using a record in the highly efficient FORALL statement. This technique is valuable when you're inserting a large number of rows.

Take a look at the example in *Listing 3. Table 1* explains the interesting parts of the retire_them_now procedure (written and run at a low-tech company that never went public nor saw its value crash, enabling them to now offer early, paid retirement to everyone over 40 years of age!).

Listing 3. Bulk INSERTing with a record.

```

1 CREATE OR REPLACE PROCEDURE retire_them_now
2 IS
3     bulk_errors    EXCEPTION;
4     PRAGMA EXCEPTION_INIT (bulk_errors, -24381);
5     TYPE employees_t IS TABLE OF employees%ROWTYPE
6         INDEX BY PLS_INTEGER;
7     retirees      employees_t;
8 BEGIN
9     FOR rec IN (SELECT *
10                FROM employees
11                WHERE hire_date < ADD_MONTHS (SYSDATE, -1 * 18 * 40))
12     LOOP
13         retirees (SQL%ROWCOUNT) := rec;
14     END LOOP;
15     FORALL indx IN retirees.FIRST .. retirees.LAST
16         SAVE EXCEPTIONS
17         INSERT INTO employees
18             VALUES retirees (indx);
19 EXCEPTION
20     WHEN bulk_errors
21     THEN
22         FOR j IN 1 .. SQL%BULK_EXCEPTIONS.COUNT
23         LOOP
24             DBMS_OUTPUT.PUT_LINE ( 'Error from element #' ||
25                 TO_CHAR(SQL%BULK_EXCEPTIONS(j).error_index) || ': ' ||
26                 SQLERRM(SQL%BULK_EXCEPTIONS(j).error_code));
27         END LOOP;
28* END;
```

Table 1. Description of the retire_them_now procedure.

LINE(S)	DESCRIPTION
3-4	Declare an exception, enabling us to trap by name an error that occurs during the bulk insert.
5-7	Declare an associative array, each row of which contains a record having the same structure as the employees table.
9-14	Load up the array with the information for all employees who are over 40 years of age.
15-18	The turbo-charged insert mechanism, FORALL, that includes a clause to allow FORALL to continue past errors and references a record (the specified row in the array).
20-26	Typical code you'd write to trap any error that was raised during the bulk insert and display or deal with each error individually.

Prior to Oracle9i Release 2, you could use the FORALL syntax, but it would have been necessary to create and populate a separate collection for each column, and then reference individual columns and collections in the INSERT statement. To get a sense of the somewhat exhausting nature of this code, see the bulkforall9i1.sql file in the [Download](#).

UPDATE SET ROW WITH RECORD BIND

Oracle9i Release 2 now gives you an easy and powerful way to update an entire row in a table from a record: the SET ROW clause. The ROW keyword is functionally equivalent to *. It's most useful when the source of the row is one table and the target is a different table with the same column specification—for example, in a scenario where rows in an application table are updated once or many times and may eventually be deleted, and where the latest state of each row (including when it's been deleted) must be reflected in an audit table. (Ideally we'd use MERGE with a RECORD bind, but this isn't supported yet.)

The new syntax for the Static SQL, single row case is obvious and compact:

```
DECLARE
    v_emprec employees%ROWTYPE
        := emp_util.get_one_row;
BEGIN
    v_emprec.salary
        := v_emprec.salary * 1.2;

    UPDATE employees_2
        SET ROW = v_emprec
        WHERE employee_id =
            v_emprec.employee_id;
END;
```

Prior to Oracle9i Release 2, this same functionality would require listing the columns explicitly, as shown in *Listing 4*.

Listing 4. Pre-Oracle9i Release 2 update of entire row.

```
DECLARE
    v_emprec    employees%ROWTYPE    := emp_util.get_one_row;
BEGIN
    v_emprec.salary := v_emprec.salary * 1.2;

    UPDATE employees
        SET first_name = v_emprec.first_name,
            last_name = v_emprec.last_name,
            email = v_emprec.email,
            phone_number = v_emprec.phone_number,
            hire_date = v_emprec.hire_date,
            job_id = v_emprec.job_id,
            salary = v_emprec.salary,
            commission_pct = v_emprec.commission_pct,
            manager_id = v_emprec.manager_id,
            department_id = v_emprec.department_id
        WHERE employee_id = v_emprec.employee_id;
END;
```

Now, it would certainly be nice to be able to use the SET ROW syntax in a FORALL statement, as follows:

```
DECLARE
    v_emprecs emp_util.emprec_tab_t
        := emp_util.get_many_rows;
BEGIN
    -- This will not work, due to:
    -- PLS-00436:
    --   implementation restriction:
    --   cannot reference fields of
    --   BULK In-BIND table of records.
    FORALL j IN
        v_emprecs.FIRST .. v_emprecs.LAST
    UPDATE employees
        SET ROW = v_emprecs (j)
        WHERE employee_id =
            v_emprecs (j).employee_id;
END;
```

Sadly, this code fails to compile with the error: "PLS-00436: implementation restriction: cannot reference fields of BULK In-BIND table of records." Instead, we must write:

```
DECLARE
    v_emprecs emp_util.emprec_tab_t
        := emp_util.get_many_rows;

    TYPE employee_id_tab_t IS
        TABLE OF employees.employee_id%TYPE
        INDEX BY PLS_INTEGER;

    v_employee_ids    employee_id_tab_t;
BEGIN
    -- Transfer just the IDs into their own
    -- collection for use in the WHERE clause
    -- of the UPDATE statement.
    FOR j IN v_emprecs.FIRST .. v_emprecs.LAST
    LOOP
        v_employee_ids (j) :=
            v_emprecs (j).employee_id;
    END LOOP;

    FORALL j IN
        v_emprecs.FIRST .. v_emprecs.LAST
    UPDATE employees
        SET ROW = v_emprecs (j)
        WHERE employee_id = v_employee_ids (j);
END;
```

If you're not yet fortunate enough to be on Oracle9i Release 2, you can see the workaround necessary to achieve this same functionality in the bulkupdate-pre9i2.sql file contained in the [Download](#).

DELETE AND UPDATE WITH RETURNING WITH RECORD BIND

You can also take advantage of rows when using the RETURNING clause in both DELETES and UPDATES. The RETURNING clause allows you to retrieve and return information that's processed in the DML statement without using a separate, subsequent query. Record-based functionality for RETURNING means that you can return multiple pieces of information into a record, rather than individual variables. An example of this feature for DELETES is shown in *Listing 5*.

Listing 5. RETURNING into a record from a DELETE statement.

```
DECLARE
    v_emprec    employees%ROWTYPE;
BEGIN
    DELETE FROM employees
        WHERE employee_id = 100
    RETURNING employee_id, first_name, last_name, email, phone_number,
        hire_date, job_id, salary, commission_pct, manager_id,
        department_id
        INTO v_emprec;

    emp_util.show_one (v_emprec);
END;
```

You can also retrieve less than a full row of information by relying on programmer-defined record types, as this next example shows:

```
DECLARE
    TYPE key_info_rt IS RECORD (
        id    NUMBER,
        nm    VARCHAR2 (100)
    );

    v_emprec key_info_rt;
BEGIN
    DELETE FROM employees
        WHERE employee_id = 100
    RETURNING employee_id, first_name
        INTO v_emprec;

    ...
END;
```

You must still list the individual columns or derived values in the RETURNING clause, making the integration a bit less than ideal (for example, Oracle *could* and perhaps will some day allow us to write RETURNING ROW INTO v_emprec). Nevertheless, this is a significant improvement over Version 9.0.1, where a RECORD could not be used as the target for INTO, requiring us to provide a long list of individual variables to hold the values returned from the DML statement.

Next, suppose that we execute a DELETE or UPDATE that modifies more than one row. In this case, we can use the RETURNING clause to obtain information from each of the individual rows modified by using BULK COLLECT to populate a collection of records! This technique is shown in *Listing 6*.

Listing 6. RETURNING multiple rows of information from an UPDATE statement.

```

DECLARE
    v_emprecs    emp_util.emprec_tab_t;
BEGIN
    UPDATE      employees
        SET salary = salary * 1.1
        WHERE hire_date < = '25-JUN-97'
    RETURNING employee_id, first_name, last_name, email, phone_number,
        hire_date, job_id, salary, commission_pct, manager_id,
        department_id
        BULK COLLECT INTO v_emprecs;
    emp_util.show_all (v_emprecs);
END;
```

Again, this is a significant improvement over Version 9.0.1, in which you would have to declare a separate collection for each value specified in the RETURNING clause, and then populate each separately. A fragment of this approach is shown in *Listing 7*; you'll find a complete implementation (74 lines!) in the bulkreturning9i1.sql script of the [Download file](#). Not pretty.

Listing 7. RETURNING multiple rows of data from an UPDATE statement in Version 9.0.1.

```

DECLARE
    TYPE employee_ids_t IS TABLE OF employees.employee_id%TYPE
        INDEX BY BINARY_INTEGER;
    ...
    v_employee_ids    employee_ids_t;
    ...
    v_emprecs        emp_util.emprec_tab_t;
BEGIN
    UPDATE      employees
        SET salary = salary * 1.1
        WHERE hire_date < = '25-JUN-97'
    RETURNING employee_id, first_name, last_name, email,
        phone_number, hire_date, job_id, salary,
        commission_pct, manager_id, department_id
        BULK COLLECT INTO v_employee_ids, v_first_names, v_last_names, v_emails,
            v_phone_numbers, v_hire_dates, v_job_ids, v_salaries,
            v_commission_pcts, v_manager_ids, v_department_ids;
    FOR j IN 1 .. v_employee_ids.LAST
    LOOP
        v_emprecs (j).employee_id := v_employee_ids (j);
        ...
    END LOOP;
    emp_util.show_all (v_emprecs);
END;
```

PERFORMANCE IMPACT OF RECORD BINDING

There's no doubt that using records in DML statements results in greatly reduced code volume and therefore increased productivity. Is there, however, a penalty to be paid in runtime execution of this leaner code? Our tests (see *Table 2*) show for the most part that there's no measurable difference between field and record-based operations.

Table 2. Scripts to examine performance impact of record binding.

SCRIPT NAME	WHAT IS TESTED?
insrec1.tst	Insert with record for table with sequence-generated primary key.
insrec2.tst	Insert with record on table with non-sequence primary key.
insrec3.tst	Insert with record on table with many columns with non-sequence primary key.
insrec4.tst	Bulk insert with record on table with non-sequence primary key.

Depending on extenuating circumstances, however, you can see more of a differential. For example, any one of the following situations could impact negatively on record-based DML processing time:

- *Use of a sequence to generate a primary key.* You can't include `<sequence>.NEXTVAL` in your record specification, so you must execute an "external" query (usually against the Oracle "dual" table) prior to the INSERT itself, to obtain the primary key value and assign it to the appropriate field in the record. See the `insrec1.tst` script for a demonstration of the impact of this step. One must conclude that a record-based INSERT is simply not a good fit for this scenario.
- *Update triggers on individual columns of the table.* An update with a record updates all columns of the table. To avoid this problem, make sure that you include a WHEN clause on your triggers to avoid extraneous execution (when NEW and OLD values are the same). See the `genwhen.sql` script for a utility that will generate the appropriate WHEN clause for each column of a table.
- If, on the other hand, you take advantage of Oracle9i Release 2's ability to perform bulk collect operations with records (see `insrec4.tst`), you'll find that record-based operations are consistently and noticeably faster than those relying on individual fields (requiring a separate collection for each field).

Record-based DML was added to the PL/SQL language primarily as a "usability" feature, rather than one related to performance. Part of the challenge of integrating new features into your "box of tricks" is that you need to know when *not* to use them. In general, if you're already working with and populating records (particularly if you're transferring data from one table to another using records), you'll find this feature to be a wonderful enhancement.

RECORDS: THE WAY TO GO

Records have always been a very powerful programming construct for PL/SQL developers. Use of records reduces code volume and also increases the resiliency of one's code, since a record defined using %ROWTYPE automatically (upon recompilation of the program) adapts to the current structure of the base cursor or table.

The inability to utilize records within SQL DML statements in a PL/SQL program has long been a frustration to developers. With Oracle9i Release2, another barrier between SQL and PL/SQL has been removed, allowing for ever-smoother programming efforts, higher productivity, and more easily maintained applications.

Download [FEUER.ZIP](#)

ABOUT THE AUTHORS

Steven Feuerstein is considered one of the world's leading experts on the Oracle PL/SQL language. He's the author or co-author of six books on PL/SQL, including the now-classic *Oracle PL/SQL Programming* and *Oracle PL/SQL Best Practices* (all from O'Reilly & Associates). Steven is a Senior Technology Advisor with Quest Software, has been developing software since 1980, and worked for Oracle Corporation from 1987 to 1992. Steven is president of the Board of Directors of the Crossroads Fund, which makes grants to Chicagoland organizations working for social, racial, and economic justice (www.CrossroadsFund.org). steven.feuerstein@quest.com.

Bryn Llewellyn is PL/SQL Product Manager, Database and Application Server Technologies Development Group, at Oracle Corporation Headquarters. Bryn has worked in the software field for 25 years. He joined Oracle UK in 1990 at the European Development Center in the Oracle Designer team. He transferred to the Oracle Text team and from there moved into Consulting as EMEA's Text specialist. He relocated to Redwood Shores in 1996 to join the Text Technical Marketing Group. His brief has recently been extended to cover Product Manager responsibility for PL/SQL. Bryn.Llewellyn@oracle.com.

ABOUT QUEST SOFTWARE, INC.

Quest Software, Inc. delivers innovative products that help organizations get more performance and productivity from their applications, databases and infrastructure. Through a deep expertise in IT operations and a continued focus on what works best, Quest helps more than 18,000 customers worldwide meet higher expectations for enterprise IT. Quest Software, headquartered in Irvine, Calif., can be found in offices around the globe and at www.quest.com.

Contacting Quest Software

Mail:	Quest Software, Inc. World Headquarters 8001 Irvine Center Drive Irvine, CA 92618 USA
Web site	www.quest.com
Email:	info@quest.com
Phones:	1.800.306.9329 (Inside U.S.) 1.949.754.8000 (Outside U.S.)

Please refer to our Web site for regional and international office information. For more information on Quest Software solutions, visit www.quest.com.

Trademarks

All trademarks and registered trademarks used in this guide are property of their respective owners.