LESS PAIN, MORE GAIN
USE THE NEW PL/SQL FEATURES IN ORACLE9i DATABASE TO GET BETTER PROGRAMS BY WRITING FEWER CODE LINES

Bryn Llewellyn, PL/SQL Product Manager, Oracle Corp
Delivered as paper #30720 at OracleWorld, Copenhagen, Tue 25-June-2002

INTRODUCTION
Both Oracle9i Database Release 1 and Release 2 introduced powerful new PL/SQL language features. These allow you to implement your required functionality using compact syntax backed by an efficient implementation. This paper will examine some very common programming tasks. We'll remind you how you would have had to implement them in versions prior to Oracle9i Database. And we'll show you the new recommended approach. You'll see that you need fewer lines of code. We'll explain why the program will be more efficient.

Another reason you often used to write more code than you wanted was to call Dbms Output.Put_Line, the de facto debugging tool of choice. We'll demonstrate how you can now forget that and debug your programs interactively using Oracle9i JDeveloper's graphical PL/SQL development environment.

SUMMARY OF ENHANCEMENTS
The new PL/SQL features in Oracle9i Database Release 1 were described in Paper 129 delivered at Oracle OpenWorld, San Francisco, Tue 4-Dec-2001. This is posted on the Oracle Technology Network website at…

otn.oracle.com/tech/pl_sql

Oracle9i Database Release 2 has added new functionality in four major areas. Two (associative arrays and the ability to use RECORDs in DML and in BULK SELECTs) are language features. The third is enhancements to the Utl_File supplied package. And the fourth exposes the industry standard JDWP protocol to enable a tools vendor to build a PL/SQL debugger. This has already been exploited by Oracle's JDeveloper in its Version 9.0.3 (soon to be available in beta for download from OTN).

ONLINE CODE SAMPLES
All the code samples used in this paper are available for easy copy&paste to SQL*Plus via the PL/SQL section on OTN. Click on Sample Code and then on New Look Samples, or go direct to…

otn.oracle.com/sample_code/tech/pl_sql/htdocs/x/start.htm

1. ASSOCIATIVE ARRAYS
1.1. OVERVIEW
The functionality of PL/SQL index-by tables has been enhanced by adding two new possibilities: index-by-pls_integer; and index-by-varchar2. We've adopted the term associative array for all these variants of index-by tables in line with common usage in other 3GLs for similar functionality.

The introduction of index-by-pls_integer means that the older datatype binary_integer need no longer be used in any new code.

The introduction of index-by-varchar2 provides a compact and easy-to-program technique for scenarios where values
need to be accessed by non-numeric key.

Previously, as the following examples show, the functionality would have been explicitly programmed, for example by using `Dbms_Utility.Get_Hash_Value` to provide index values for an `index-by-binary_integer` table.

**1.2. SCENARIO**

The requirement to look up a value via a unique non-numeric key is a generic computational problem. Of course the Oracle9i Database provides a solution with SQL and a B*-tree index. But there’s a set of scenarios where considerable performance improvement can be obtained by instead using an explicit PL/SQL implementation. This was true even before the new features discussed in this paper were available. These scenarios are characterized by very frequent lookup in a relatively small set of values, usually in connection with flattening a relational representation for reporting or for UI presentation. In order not to complicate the following examples with distracting detail, we’ll use a simple neutral scenario.

Suppose we have a set of English-French vocabulary pairs stored persistently in the obvious way in a schema-level table...

```
select * from translations;
ENGLISH              FRENCH
-------------------- ----------
computer             ordinateur
                     arbre
book                 livre
                     chou
country              pays
vehicle              voiture
garlic               ail
apple                pomme
desk                 écrivain
furniture            meubles
...
```

Our task is to allow lookup from French to English, and to allow efficient addition of new vocabulary pairs. This is abstracted as the Vocab package, thus...

```
package Vocab is
  function Lookup ( p_english in varchar2 ) return varchar2;
  procedure New_Pair ( p_english in varchar2, p_french in varchar2 );
end Vocab;
```

And we’ll use this test...

```
begin
  Dbms_Output.Put_Line ( Vocab.Lookup ( 'tree' ) );
  Vocab.New_Pair ( 'garden', 'jardin' );
  Dbms_Output.Put_Line ( Vocab.Lookup ( 'garden' ) );
end;
```

Of course all implementations satisfy the test. We’ll examine some alternative implementations, leaving the new – and best – approach to last.

**1.3. NAIVE PURE SQL APPROACH**

Here we’re concerned only about correctness of behavior and ease of algorithm design. We’ll accept the performance we get.
package body Vocab is
  function Lookup ( p_english in varchar2 ) return varchar2
  is
    v_french translations.french%type;
  begin
    select french into v_french from translations
      where english = p_english;
    return v_french;
  end Lookup;

  procedure New_Pair ( p_english in varchar2, p_french in varchar2 ) is
  begin
    insert into translations ( english, french ) values ( p_english, p_french );
  end New_Pair;
end Vocab;

The algorithm is of course trivial and is due entirely to native SQL language features. Each time Lookup is invoked, we make a round trip between PL/SQL and SQL. The associated expense of this frequently repeated context switch could become very significant.

1.4. LINEAR SEARCH IN INDEX-BY-BINARY_INTEGER TABLE
Here we recognize that we can improve performance by eliminating the round trip between PL/SQL and SQL for lookup by caching all the target values in an appropriate PL/SQL structure. Pre Oracle9i Database Release 2, the natural choice is an index-by table, since we are happy with the persistent representation as a relational table and do not want to maintain any persistent denormalization.

Note: the following implementation was coded to run in Oracle9i Database Version 9.0.1. It also runs in Oracle8i Database Version 8.1.7.

package body Vocab is
  type word_list is table of translations%rowtype
    index by binary_integer /* can't use pls_integer pre-9.2 */;
  g_english_french word_list;

  function Lookup ( p_english in varchar2 ) return varchar2
  is
    begin
      for j in 1..g_english_french.Last()
        loop
          if g_english_french(j).english = p_english
            then
              return g_english_french(j).french;
          end if;
        end loop;
    end Lookup;

  procedure New_Pair ( p_english in varchar2, p_french in varchar2 ) is
    begin
      idx := g_english_french.Last() + 1;
      g_english_french(idx).english := p_english;
      g_english_french(idx).french := p_french;
      insert into translations ( english, french ) values ( p_english, p_french );
    end New_Pair;
begin /* package initialization */
Oracle9i Database

```
declare idx binary_integer := 0;
begin
  for j in ( select english, french from translations ) loop
    idx := idx+1;
    g_english_french(idx).english := j.english;
    g_english_french(idx).french := j.french;
  end loop;
end; end Vocab;
```

The algorithm is still trivial, but does require some explicit coding.

The entire table contents are loaded into a PL/SQL index-by-binary_integer table when the package is initialized. We cannot use the more efficient and compactly written BULK SELECT to initialize the index-by table. The attempt causes compilation error PLS-00597 in Version 9.0.1 and earlier. This restriction is lifted in Oracle9i Database Release 2 enabling us to write the initialization thus...

```
declare cursor cur is select english, french from translations;
begin
  open cur; fetch cur bulk collect into g_english_french; close cur;
end;
```

We’ll discuss enhancements to RECORD functionality in the next section.

1.5. HASH-BASED LOOKUP IN INDEX-BY-BINARY_INTEGER TABLE

The linear search algorithm suffers from the well-known disadvantage that on average we’ll examine half the elements in the index-by table before we find a match. A possible improvement is to maintain the elements in lexical sort order and to use a binary chop algorithm: compare the search target to the half-way element to determine which half it’s in; repeat this test recursively on the relevant half. This requires more elaborate coding – and testing – and poses a design problem for the New_Pair procedure. Either the array must be opened at the insertion point copying all later elements to the next slot, or it must be created sparse. Neither of these approaches is very comfortable, and the latter is not complete until the corner case where a gap fills up is catered for.

A more popular approach therefore is to determine the element index for storing the current value by hashing that value.

Note: the following implementation was coded to run in Oracle9i Database Version 9.0.1. It also runs in Oracle8i Database Version 8.1.7.

```
package body Vocab is
  /* can't use pls_integer pre-9.2 */
  hash               binary_integer;
  g_hash_base constant number := 100 /* care must be given to... */;
  g_hash_size constant number := 100 /* ...the choice of these values */;
  type word_list is table of translations.french%type
    index by binary_integer;
  g_english_french word_list;

  function Lookup ( p_english in varchar2 ) return varchar2 is
    begin
      hash := Dbms_Utility.Get_Hash_Value ( p_english, g_hash_base, g_hash_size );
      return g_english_french(hash);
    end Lookup;

  procedure New_Pair ( p_english in varchar2, p_french in varchar2 ) is
    begin
```

4 Paper # 30720
The above algorithm is too naïve for real-world use. There is no guarantee that two distinct values for the `name IN parameter to `Get_Hash_Value` will always produce distinct hash values. Thus to be robust, a collision avoidance scheme must be implemented. This is a non-trivial design, implementation and testing task. (Oracle provides no specific support for this.)

It’s probable that the resulting `index-by-binary_integer` table will be quite sparse. This point will be discussed after describing the new `index-by-varchar2` table.

### 1.6. Direct Lookup in `index-by-varchar2` Table

Of course a yet more elaborate ambition pre-Oracle9i Database Release 2 would be – after studying the relevant computer science textbooks – to implement a B*-tree structure in PL/SQL, horror of wheel re-invention notwithstanding!

The datastructure might this…

```plsql
type Node_t is record (
  value varchar2(20),
  left_child binary_integer /* refer to the array index... */,
  right_child binary_integer /* ...of the... */,
  parent binary_integer /* ...relevant element. */
);
type Btree_t is table of Node_t index by binary_integer;
the_tree Btree_t;
```

However, the implementation would be very far from trivial, and is certainly too long and complex for inclusion in this paper.

A far better approach – newly possible in Oracle9i Database Release 2 – is to use precisely the B*-tree organization of the values but to do so implicitly via a language feature, the `index-by-varchar2` table.

You can think of the `index-by-varchar2` table as the in-memory PL/SQL version of the schema-level index organized table. Here’s the syntax…

```plsql
declare
  idx varchar2(20);
  type word_list is table of varchar2(20)
    index by idx%type;
the_list word_list;
begin
  the_list ('computer') := 'ordinateur';
  ...
  the_list ('tree') := 'arbre';
  /* loop in lexical sort order */
```
idx := the_list.First();
while idx Is not null
loop
   Dbms_Output.Put_line ( idx || ' : ' || the_list(idx) );
   idx := the_list.Next(idx);
end loop;
end;

The index-by-varchar2 table is optimized for efficiency of lookup on a non-numeric key, where the notion of sparseness is not really applicable. The index-by-*_integer table (where now *_integer can be either pls_integer or binary_integer) in contrast is optimized for compactness of storage on the assumption that the data is dense.

This implies that there might be cases where, even though the key is inherently numeric, it's better to represent it as a index-by-varchar2 table via a To_Char conversion.

This new construct now allows the following very natural implementation of the Vocab body...

package body Vocab is
   type word_list is table of translations.french%type
     index by translations.english%type;
   g_english_french word_list;

   function Lookup ( p_english in varchar2 ) return varchar2
     is
      begin
         return g_english_french( p_english );
      end Lookup;

   procedure New_Pair ( p_english in varchar2, p_french in varchar2 ) is
      begin
         g_english_french( p_english ) := p_french;
         insert into translations ( english, french ) values ( p_english, p_french );
      end New_Pair;

begin /* package initialization */
   for j in ( select english, french from translations )
     loop
         g_english_french( j.english ) := j.french;
     end loop;
end Vocab;

Note that it is not yet possible to use a index-by-varchar2 table as the target or source of a bulk binding construct.

The body of function Lookup is now so trivial that you might decide to dispense with it altogether (suppressing a mild qualm about violating some rule of best practice) and use this stripped down implementation...

package Vocab is
   type word_list is table of translations.french%type
     index by translations.english%type;
   lookup word_list;
   procedure New_Pair ( ... );
end Vocab;

package body Vocab is

   procedure New_Pair ... end New_Pair;
begin /* package initialization */
   for j in ( select english, french from translations )
     loop
         lookup ( j.english ) := j.french;
end Vocab;
end loop;
end Vocab;

1.7. PERFORMANCE COMPARISON: SCHEMA-LEVEL TABLE VS INDEX-BY-VARCHAR2 TABLE

Since index-by tables are pure memory structures, they are much faster than schema-level tables. This was of course the motivation behind the previous discussion. As a sanity check, Appendix A.1 shows a trivial test to compare populating an index-by-varchar2 table with populating a schema-level table with the same data. This simulates the requirement to cache intermediate results, for example for sorting.

1.8. ASSOCIATIVE ARRAYS: SUMMARY

The new ability to use an index-by-varchar2 table is a major enhancement that allows a class of very common programming tasks to be implemented much more performatantly than was possible pre Version 9.2.0, with very much less design and testing, and in dramatically fewer lines of code.

The new ability to use an index-by-pls_integer table is a minor enhancement that allows a clear coding guideline for all new projects: never use BINARY_INTEGER except where it’s required to match the type in an existing API.

2. BINDING RECORDS TO SQL STATEMENTS

2.1. OVERVIEW OF FUNCTIONALITY

A PL/SQL RECORD is the datatype that corresponds to a row in a schema-level table. It is the natural construct to use when manipulating table rows programmatically, especially when a row is read (via SELECT, UPDATE...RETURNING or DELETE...RETURNING), manipulated programmatically, and then stored (via INSERT or UPDATE) in another table with the same shape. It has these advantages over representing the column values in individual variables...

- The declaration is compact, using mytable%rowtype.
- By using %rowtype, 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.
- The record variable can be used as a formal and actual parameter to a procedure or function giving compact and guaranteed correct notation and allowing optimizations in the implementation of parameter passing.
- The SQL-PL/SQL interface allows a syntax which does not list the columns of the source/target table explicitly, again allowing for robust code which has a greater degree of schema-independence.

These features not only reduce effort of writing the code, but also increase the chances of its correctness.

The use of RECORDs in the SQL-PL/SQL interface was greatly restricted at Oracle9i Database Version 9.0.1. The syntax was supported only for the single row SELECT case (but for that in both Static SQL and Native Dynamic SQL). Thus the advantages listed above were not yet capable of being realized. To put this another way: pre Version 9.2.0 PL/SQL coding standards, to ensure schema-independence, typically forbade the use of SELECT * with an explicit list of bind variables and forbid the use of INSERT without listing the target columns. The new Version 9.2.0 functionality, by using a RECORD bind, now allows both SELECT * and INSERT without listing the target columns (and thus a change to your coding standards).

A large number of restrictions have been lifted at Version 9.2.0. Full support is now provided for all flavors of SELECT. And support is provided (with some minor restrictions - see below) for all Static SQL flavors of INSERT, DELETE and UPDATE.

However, some restrictions do remain at Version 9.2.0…

- No use is supported of EXECUTE IMMEDIATE in connection with INSERT, UPDATE or DELETE. (It is supported for SELECT, as stated above.) In other words, RECORDs are not yet supported for DML using Native Dynamic SQL.
- With DELETE and UPDATE...RETURNING the column-list must be written explicitly in the SQL statement.
• In the bulk syntax case, you cannot reference fields of the in-bind table of records elsewhere in the SQL statement (e.g., in the where clause).

### 2.2. Overview of Examples

The following examples rely on the `employees` table. This is in the `hr` schema which is installed in the seed database. The script to create this schema is `demo/scherm/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 given in Appendixes A.2.1 and A.2.2.

### 2.3. Select with Record Bind

This example shows the syntax for BULK SELECT into an `index-by-*-integer` table `RECORDs` in Static SQL.

```sql
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.Show_All ( v_emprecs );
end;
```

(The clause `limit 10` is equivalent to where `rownum <= 10`.) The Native Dynamic SQL version follows the obvious way thus...

```sql
open cur for
  'select * from employees where hire_date >= :avg_hire_date'
using '25-JUN-97';
```

...where `cur` is now declared of type `sys_refcursor` and where the other code lines in the block are unchanged.

The above both fail to compile pre Version 9.2.0 with PLS-00597: expression 'V_EMPRECS' in the INTO list is of wrong type, so we must make an uncomfortable compromise: either give up the BULK SELECT and use an explicit loop...

```sql
declare
  v_emprec employees%rowtype;
  v_emprecs Emp_Util.emprec_tab_t;
cursor cur is
  select * from employees where hire_date >= '25-JUN-97';
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.Show_All ( v_emprecs );
end;
```

...or give up schema-independence and list the columns in order to retain the BULK SELECT ...
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 >= '25-JUN-97';
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.Show_All ( v_emprecs );
end;
The complete code for this block is shown in Appendix A.2.3, and is over 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.
The above two workarounds for the Static SQL case transform naturally for the Native Dynamic SQL case.

2.4. INSERT WITH RECORD BIND
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
  insert into employees_2 values v_emprec;
end;
The corresponding Native Dynamic SQL case (using EXECUTE IMMEDIATE) is not yet supported. The above fails to compile pre Version 9.2.0 with PL/SQL: ORA-00906: missing left parenthesis, so we must list the columns explicitly...

declare
  v_emprec employees%rowtype := Emp_Util.Get_One_Row;
begin
  insert into employees_2 ( employee_id,
    ...
  ) values ( v_emprec.employee_id,
    ...
  );
end;
The complete code for this block is shown in Appendix A.2.4. The new syntax for the Static SQL **BULK INSERT** case is also obvious…

```sql
declare
  bulk_errors exception;
  pragma exception_init ( bulk_errors, -24381 );
  v_emprecs Emp_Util.emprec_tab_t := Emp_Util.Get_Many_Rows;
begin
  v_emprecs(1).employee_id := /* contrive "unique constraint violated" error on INSERT */
  v_emprecs(2).employee_id;
  forall j in v_emprecs.first..v_emprecs.last save exceptions
  insert into employees_2 values v_emprecs(j);
exception when bulk_errors then
  for j in 1..sql%bulk_exceptions.count
  loop
    Dbms_Output.Put_Line ( 'Error from element #' ||
      To_Char(sql%bulk_exceptions(j).error_index) || ': ' ||
      Sqlerrm(-sql%bulk_exceptions(j).error_code) );
  end loop;
end;
```

The **SAVE EXCEPTIONS** syntax is illustrated to show that it’s consistent with **BULK INSERT** of **RECORDs** and because it represents best practice. This feature itself was new at **Version 9.0.1** and represents an important enhancement with respect to Oracle8i Database.

The above again fails to compile pre **Version 9.2.0** with **PL/SQL: ORA-00906: missing left parenthesis**. Moreover, the attempt to list the columns explicitly analogously to the single row case above causes **PLS-00436: implementation restriction: cannot reference fields of BULK In-BIND table of records**. (This restriction persists at **Version 9.2.0** but is less significant because of the new **RECORD** functionality.) So the workaround is both to slice the table of **RECORDs** vertically and to list the columns explicitly...

```sql
declare
  bulk_errors exception;
  pragma exception_init ( bulk_errors, -24381 );
  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 := Emp_Util.Get_Many_Rows;
begin
  v_emprecs(1).employee_id := /* contrive "unique constraint violated" error on INSERT */
  v_emprecs(2).employee_id;
  for j in 1..v_emprecs.Last
  loop
    v_employee_ids(j) := v_emprecs(j).employee_id;
    ...
  end loop;
  forall j in v_emprecs.first..v_emprecs.last save exceptions
  insert into employees_2 ( employee_id,
    ... )
  values ( v_employee_ids(j),
```
exception when bulk_errors then
   ...
end;

The complete code for this block is shown in Appendix A.2.5 and is about 90 lines long! Again, it’s approaching what is feasible to maintain. And again, it 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.

**INSERT** with a **RECORD** bind is not yet supported for Native Dynamic SQL.

### 2.5. DELETE AND UPDATE WITH RETURNING WITH RECORD BIND

The behavior for **DELETE** and **UPDATE** with **RETURNING** is symmetrical. **DELETE** is illustrated for the single row case and **UPDATE** for the **BULK** case. Support for this is not quite ideal, even at **Version 9.2.0**. We have to list the columns explicitly thus...

```sql
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;
```

Nevertheless this is a significant improvement over **Version 9.0.1** where a **RECORD** could not be used as the target for **INTO**, and where you had to write...

```sql
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.employee_id,
```
v_emprec.first_name,
v_emprec.last_name,
v_emprec.email,
v_emprec.phone_number,
v_emprec.hire_date,
v_emprec.job_id,
v_emprec.salary,
v_emprec.commission_pct,
v_emprec.manager_id,
v_emprec.department_id;
Emp_Util.Show_One ( v_emprec );
end;

Here is the corresponding *Version 9.2.0 BULK* flavor...

```
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* where you had to write...

```
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_employees,
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_employees.Last
loop
  v_emprcs(j).employee_id := v_employees(j);
  ...
end loop;
Emp_Util.Show_All ( v_emprcs );
end;

The complete code for this block is shown in Appendix A.2.6.

**UPDATE** and **DELETE** with a **RECORD** bind are not yet supported for Native Dynamic SQL.

### 2.6. UPDATE SET ROW WITH RECORD BIND

The **ROW** keyword is functionally equivalent to * and is new at *Version 9.2.0*. It would be most useful where 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 on or many times and may eventually be deleted, and where the latest state of each row (including when it has 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_emprc employees%rowtype := Emp_Util.Get_One_Row;
begin
  v_emprc.salary := v_emprc.salary * 1.2;
  update employees_2
    set row = v_emprc
    where employee_id = v_emprc.employee_id;
end;

Pre *Version 9.2.0* this would require listing the columns explicitly...

declare
  v_emprc employees%rowtype := Emp_Util.Get_One_Row;
begin
  v_emprc.salary := v_emprc.salary * 1.2;
  update employees
    set
      first_name       = v_emprc.first_name,
      last_name        = v_emprc.last_name,
      email            = v_emprc.email,
      phone_number     = v_emprc.phone_number,
      hire_date        = v_emprc.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;

We would have liked to have written the BULK case thus…

declare
  v_emprecs Emp_Util.emprec_tab_t := Emp_Util.Get_Many_Rows;
begin
  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;

…but this fails to compile with 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
  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;

The pre Version 9.2.0 workaround for this ish shown in Appendix A.2.7.

UPDATE SET ROW is not yet supported for Native Dynamic SQL.

2.7. BINDING RECORDS TO SQL STATEMENTS: SUMMARY

This enhancement allows the value of the RECORD construct to be realized in several new classes of application programming. The value is mainly dramatically to reduce development effort, as the code samples showing the pre Version 9.2.0 workarounds illustrate vividly. These samples do show a certain amount of copying between alternative PL/SQL representations, and so the new functionality, by avoiding this, also represents a performance enhancement.

3. UTL_FILE ENHANCEMENTS

3.1. OVERVIEW

Pre Version 9.2.0, the way to denote the director(ies) for files was via the UTL_FILE_DIR initialization parameter. This suffered from the disadvantages that the instance had to be bounced to make changes to the list of directories and that there was no security scheme (all users could access files on all directories). Version 9.2.0 allows the same mechanism (the DIRECTORY schema object) to be used with Utl_File as is used for BFILEs. You should consider the UTL_FILE_DIR initialization parameter as slated for deprecation.

The line length limit for Utl_File.Get_Line and Utl_File.Put_Line has been increased from 1K to 32K.
The following new API primitives have been introduced…

For operating system file management

- procedure `Fgetattr`
- procedure `Fccopy`
- procedure `Fremove`
- procedure `Frename`

For handling RAW data

- procedure `Fseek`
- function `Fgetpos`
- procedure `Get_Raw`
- procedure `Put_Raw`

In addition, `Get_Line` and `Get_Line_Nchar` have acquired a new defaulted `binary_integer` parameter `len`. And `Put_Line` has acquired a new defaulted boolean parameter `autoflush`. `Utl_File` performance is improved via transparent internal reimplementation.

3.2. ALWAYS USE `MAX_LINESIZE => 32767`

You now have functionality to allow you to read from character data files with arbitrarily long lines. (These might occur as mechanically generated output from an unfriendly program.) And together with the brand new functionality to handle raw data, you can read from files of arbitrary size.

Oracle recommends that you always use `max_linesize => 32767` in the call to `Utl_File.Fopen`. This is a “magic number” which prevents error if the linesize exceeds `max_linesize` (for a character file) or if the file size exceeds `max_linesize` (for a binary file).

If the file is treated as character data, then `max_linesize` must be no smaller than the longest line. Else an exception is raised when a line is attempted read whose length exceeds `max_linesize` unless `max_linesize` is 32767.

If the file is treated as binary data, then `max_linesize` must be no smaller than the size of the file unless `max_linesize` is 32767.

3.3 COPYING, DELETING AND RENAMING FILES

The APIs for these new features are unsurprising and so are illustrated only in Appendix 3.2. (Appendix 3.1 sets up the Oracle `USER`.) `Utl_File.Fgetattr` returns the blocksize and the size in bytes of an existing file. Of course pre `Version 9.2.0` you had to do some research and some slightly tricky implementation – costing more lines of code – to achieve this functionality via call-out to Java or to C.

3.4. HANDLING RAW DATA

Again the APIs for this are unsurprising. See Appendix 3.3. And again, pre `Version 9.2.0`, you had to achieve this functionality via call-out to Java or to C.

3.4. UTL_FILE ENHANCEMENTS: SUMMARY

Functionality is now provided natively for a number of common file i/o tasks. This both dramatically reduces the amount of code you need to write and delivers better performance. The performance improvement is further enhanced by some transparent internal changes. Robustness is improved by removing some uncomfortable limits, and
security is improved by adopting the BFILE model.

It should be noted that several general areas of functionality that historically have required the use of Utl_File now have explicit, and better, solutions within Oracle9i Database Release 2. The most obvious example is of course the BFILE, which allows an external bytestream to be visible for Oracle processing (but which famously does not support writing to the external bytestream). The external table feature supercedes the use of Utl_File for record-oriented data held on external files. Utl_File has also been used to write messages to a directory which is polled by an operating system daemon which sends email. This approach is superceded by the Utl_Smtp package. And it has been used to read in XML bytestreams for parsing. This approach is superceded by the new XML DB feature. Nevertheless, there remain cases where an explicit approach is required. Perhaps the most obvious is logging of progress in some long-running program, and subsequent programmatic analysis of the log. (This functionality can be obtained with a logging table and autonomous transactions, but Utl_File is more performant.)

4. GUI DEBUGGING VIA JDEVELOPER VERSION 9.0.3

4.1. OVERVIEW

JDeveloper Version 9.0.3 will soon (after the time of writing this paper, June 2002) be available for download from OTN. Its PL/SQL IDE subcomponent is extended to provide support for graphical debugging of PL/SQL stored subprograms. This includes support for all the new features discussed in this paper.

4.2. THE CONNECTION MODEL

For editing and compiling of code read from and stored in the database and for general schema-browsing, JDeveloper connects to the database via Oracle Net in classical two-tier client-server style. As part of the development cycle, you might like to execute a procedure or a function. JDeveloper uses a dedicated client-server connection for this. (A GUI allows you to supply parameter values, and the log window shows any results from Dbms_Output.)

In contrast, debugging is initiated from the Oracle shadow process, in response to executing this call from the client…

Dbms_Debug_Jdwp.Connect_Tcp ( :the_node, :the_port, ... );

…which causes the shadow process to attempt to create a debugging session to a JDWP server. JDWP (the Java Debugging Wire Protocol) is an industry standard which, as the name suggests, was invented to support Java debugging. However, it's completely suitable for PL/SQL too, and was adopted by Oracle to make it easier for third party tools vendors to implement PL/SQL debugging. (JDWP is at the same protocol level in the discussion of distributed computing as say HTTP or Oracle Net and is typically implemented on top of TCP/IP.)

Note: In many cases, the calls to the Dbms_Debug_Jdwp API will be made transparently by the debugging tool or by some client or middle-tier infrastructure. This is covered in greater detail below.

A second benefit of using JDWP is (of course) that JDeveloper and other third party tools can support the debugging of database stored Java subprograms. And in an application which is implemented in a mix of Java and PL/SQL, the execution point moves seamlessly from one environment to the other during debugging, and the integrity of the call stack is maintained across the language boundary.

JDeveloper incorporates a JDWP listener which – just as can the database tnslistener – can be started on any port. And just as the tnslistener can, the JDWP listener can spawn any number of debugging sessions running concurrently. (These run as threads within JDeveloper.) In an exotic case, the Oracle shadow processes (aka Oracle server processes) which are the debugging clients (the terminology can get a bit confusing!) could be running on different machines, communicating with each other via say Oracle Advanced Queuing.
4.3. **Pre-conditions for Debugging**

The debugging Oracle user must have the `debug connect session` and `debug any procedure` system privileges and must have `execute` privilege on the subprograms of interest.

For a particular subprogram to be debuggable, its source code must not be wrapped, and it must have been compiled with debug information, thus...

```
alter session set plsql_debug=true;
create or replace [ package | package body | procedure | function ] P is
  ...
end P;
```

... or...

```
alter [ package | procedure | function ] P compile debug;
```

(If you use JDeveloper to do your compiling, this will happen automatically.) Moreover (at Version 9.2.0) it must not have been compiled **NATIVE**. You can check the last two conditions with this query...

```
select param_value from user_stored_settings
  where param_name = 'plsql_compiler_flags'
  and object_name = :my_object
  and object_type = :my_type;
```

... which must give `'INTERPRETED,DEBUG'`.

If any of these conditions is not satisfied for a particular subprogram, then the execution point will quietly step over that subprogram while debugging.

4.4. **Setting Up for a Debugging Exercise**

4.4.1 **Breakpoints**

You can choose either to break at the very first executable statement (not bothering to set up any breakpoints) or you
can set one or several breakpoints and choose to break at the first that is encountered. When paused, you can set and remove breakpoints in the normal way.

### 4.4.2. DEBUGGING MODES

#### 4.4.2.1. GUI-INITIATED DEBUGGING

A debugging exercise will always involve opening the source code read from the database for the subprograms of interest in the source window. This is how the current execution point is displayed and how you set breakpoints. In many scenarios it’s sufficient to initiate debugging directly from JDeveloper by nominating the top-level subprogram. (The GUI is a flavor of that used simply to execute a particular subprogram.) In this approach, you do not need to know about `Dbms_Debug_Jdwp.Connect_Tcp`, and can survive without understanding the connection model explained above. (The magic to control the JDWP listener is implicit, and in the default configuration it chooses the first available port automatically.)

If you want to debug code which is invoked when a trigger fires, the simplest way to do this is to encapsulate the SQL statement that fires the trigger in a procedure and to invoke this procedure from the JDeveloper GUI.

#### 4.4.2.2. REMOTE DEBUGGING

Sometimes though, the server-side PL/SQL which you want to debug is invoked from a client. This might be a Pro*C or Java application in a classical two-tier model. Or it might be the `mod_plsql` component of Oracle9iAS (or the equivalent) as the middle tier in a three-tier model. In all these cases, the parameters that define the debugging case of interest are set by application logic, and it would of course be undesirable and ineffective to change the application code to report the values of these parameters to allow calling the server-side PL/SQL manually from say SQL*Plus. Similarly it would be frustrating to have to change the application code to include calls to the `Dbms_Debug_Jdwp` APIs.

Oracle caters for this requirement, in the two-tier case, via the environment variable `ora_debug_jdwp`. Here is how it might be set…

```
set ora_debug_jdwp=host=lap99.acme.com;port=2125
```

This variable is read by the OCI layer and when non-null causes this layer to issue a call to `Dbms_Debug_Jdwp.Connect_Tcp` immediately on connection, parameterized as indicated. (This approach also allows you to debug subprograms called from an after logon trigger.)

In this case JDeveloper (or an equivalent third party tool) must be running on the indicated node, and its JDWP listener must have been started manually to listen on the indicated port. The user achieves this by choosing remote debugging rather than the default GUI-initiated debugging mode via an interface that allows selection of the port.

An example client application, written in Java and using SQLJ, which is sensitive to the `ora_debug_jdwp` environment variable is given in Appendix A.4.3.

If the client is `mod_plsql` then a more elaborate approach is needed, since the requirement is to turn on debugging for the server-side PL/SQL that supports a particular browser (pseudo) session (and of course not to turn it on for other concurrent browser sessions). In this case the browser sets a designated cookie in response to specific UI choice by the debugging user, and `mod_plsql` acknowledges this for each server call (corresponding to each successive delivered browser page) until the cookie is unset, by calling `Dbms_Debug_Jdwp.Connect_Tcp` before its normal calls and by calling `Dbms_Debug_Jdwp.Disconnect` after these. The cookie is parameterized with the node and port for the JDWP listener. Of course each successive browser page is typically delivered by a different Oracle session, but this does not impact the debugging flow since each new session kicks off a JDWP session to the same JDeveloper invocation.

If you use a `jdbc:thin` Java client or some middle tier infrastructure other than Oracle9iAS’s `mod_plsql`, you’ll need to make calls to the `Dbms_Debug_Jdwp` API in your production code.
4.5 Debugger features

JDeveloper implements the normal set of debugger features…

- GUI-initiated and remote debugging modes.
- Ability to support one or many concurrent debugging clients and thus to debug interacting processes.
- Current execution point displayed as highlighted line in the source window. This shows the source code of executing program unit, extracted dynamically from the database where the code is executing.
- Breakpoints can be set and unset before and during the debugging session. Displayed as highlighted line in source window.
- Can start debugging session with Step Into, Step Over or Run to First Breakpoint.
- When paused, can continue with Resume (aka Run to Next Breakpoint), Step Into or Step Over. Or Abort the current debugging session.
- A condition can be attached to a breakpoint so that execution stops there only if the condition is met.
- When paused, can view the values of variables visible in the current subprogram, and the values for all variables that are currently “alive” (even if they aren’t visible at the current execution point). See call stack display below.
- Intuitive display for collection objects and objects of user-defined abstract datatypes.
- Call stack display. Can click anywhere in the stack to view the values of variables that belong to the selected subprogram. Intuitive display for recursive calls.
- When paused, can modify the values of variables (this comes with the normal caveats regarding what you can then deduce about your program’s subsequent behavior).

4.6. Scenario

The screenshots below illustrate debugging the following scenario. The code is given in Appendix A.4.1.

- An index-by-pls_integer table of RECORDs of employees%rowtype is populated by BULK SELECT using Native Dynamic SQL.
- The table of RECORDs is inserted into a second table with the same shape with BULK INSERT using Static SQL with the save exceptions construct.
- This table has a trigger which examines the NEW values and raises an exception on conditions of these data.
- An exception handler traverses the index-by table of exception codes.
- The new contents of the target table are selected using the same subprogram as at the start, but now the Native Dynamic SQL uses a different table name. This confirms the successful insert of those rows for which an exception was not raised.

4.7. Screenshots

4.7.1. The code window

This shows the current execution point on the line close cur in function Select_Many and a breakpoint set on the line emprecs := empty_emprecs in procedure Insert_Many.
4.7.2. **THE DATA WINDOW**

This shows the state of the variables of `function Select_Many` when paused as in the previous screenshot. The second element of the `emprec` index-by table is expanded to show the values of each field in the `RECORD`…
4.7.3. THE STACK

As execution moves on to the BULK INSERT into employees_2, which the debugger shows as a single statement, the before insert trigger is seen to fire once for each row as expected. Here is how the call stack looks when the execution point is in the trigger...

The procedure Jdev_Tmp_Proc_1 was created dynamically by JDeveloper as the harness to launch Emp_Util_920.Insert_many with the user-supplied value for its p_limit input parameter. The following two
screenshots show the execution point at \texttt{Raise\_Application\_Error} in the trigger and the values of the fields in the \textbf{NEW} row, with the value of salary that causes the exception to be raised highlighted.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{screenshot.png}
\caption{Oracle 9i Database Screenshot}
\end{figure}

\textbf{4.8. DEBUGGING IN VERSION 9.0.1 AND EARLIER}

JDeveloper \textit{Version 9.0.3} also implements PL/SQL debugging via the earlier \texttt{Dbms\_Debug} API to allow it to be used against database versions earlier than \textit{Version 9.2.0}. Appendix \textit{A.4.2} shows the scenario re-implemented to run in \textit{Version 8.1.7} so that you can try it. Of course since the scenario was designed to highlight features introduced in
Version 9.0.1 and Version 9.2.0, the code to implement the same functionality in Version 8.1.7 is hugely more voluminous, echoing the title of this paper! (This has nothing to do with the debugger API per se.)

4.9. GUI DEBUGGING VIA JDeveloper: SUMMARY
The value of a GUI debugger is self-evident. This is illustrated vividly by the code samples that use the RECORD construct, especially when index-by tables of RECORDs are used. PL/SQL of course does not provide syntax to display all fields in each row of an index-by tables of RECORDs, and the effort required to program a loop to do this via Dbms_Output, though not rocket science, is time-consuming and potentially error-prone. The last thing one wants to do while debugging is spend time on debugging one’s ad hoc debugging instrumentation! JDeveloper provides ready-made intuitive mechanisms for displaying the values of arbitrarily complex structures.

5. ORACLE9i IN ACTION
170 Systems, Inc (www.170Systems.com) have been an Oracle Partner for eleven years and participated in the Beta Program for the Oracle9i Database Version 9.2.0. They have now certified their 170 MarkView Document Management and Imaging System™ against Version 9.2.0.

The 170 MarkView Document Management and Imaging System provides Content Management, Document Management, Imaging and Workflow solutions – all tightly integrated with the Oracle9i Database, Oracle9i Application Server and the Oracle E-Business Suite. Enabling businesses to capture and manage all of their information online in a single, unified system – regardless of original source or format – the 170 MarkView solution provides scalable, secure, production-quality Internet B2B and intranet access to all of an organization’s vital information, while streamlining the associated business processes and maximizing operational efficiency.

A large-scale multi-user, multi-access system, 170 MarkView supports the very large numbers of documents, images, concurrent users, and the high transaction rates required by 170 Systems customers. Therefore performance and scalability are especially important. 170 Systems customers include organizations such as British Telecommunications, E*TRADE Group, the Apollo Group and the University of Pennsylvania.

170 Systems are planning to take advantage of many of the new PL/SQL features that are available in Version 9.2.0, for example, Associative Arrays to improve performance in an application that performs complex stack manipulations. Values are taken from the stack using random access based on a identifying character name. Previously, this was simulated using hashing routines written using PL/SQL, therefore a performance improvement is expected by using Associative Arrays implemented natively in Version 9.2.0.

170 Systems tested the JDeveloper PL/SQL debugging environment extensively. They like it, and plan to adopt it as their PL/SQL debugging environment of choice for their developers.

CONCLUSION
All the enhancements discussed in this paper – associative arrays, use of RECORD binds, Utl_File enhancements, and PL/SQL debugging via JDeveloper Version 9.0.3 – dramatically reduce the programming effort required for applications with matching requirements. And the first three also deliver more performant programs.

APPENDIX
A.1. PERFORMANCE COMPARISON: SCHEMA-LEVEL TABLE VS INDEX-BY-VARCHAR2 TABLE
This following procedure compares the time to populate a schema-level table with that to populate an index-by-varchar2 table with the same data. This simulates the requirement to cache programmatically generated intermediate results, for example for sorting. It is assumed that the scenario will involve some random access to the data, and so the time to create a B*-tree index on the schema-level table is included. (Note that because of this requirement we cannot use a temporary table effectively.) Recall that the index-by-varchar2 table is implicitly created and maintained as an index-organized structure.
procedure P ( nof_rows in pls_integer ) is
  -- assumes...
  -- create table t ( key varchar2(30), val integer );
index_doesnt_exist exception;
pragma exception_init ( index_doesnt_exist, -1418);
t0 number;
t_schema_level number;
t_index_by number;
type tab_t is table of integer index by varchar2(30);
my_tab tab_t;
my_empty_tab tab_t;
begin
  execute immediate 'truncate table t' /* for clean restart of test */;
t0 := Dbms_Utility.Get_time;
  begin
    execute immediate 'drop index t_key';
    exception when index_doesnt_exist then null;
  end;
  for j in 1..nof_rows
    loop
      insert into t ( key, val ) values ( To_Char(j), j );
    end loop;
  execute immediate 'create index t_key on t(key)';
t_schema_level := Dbms_Utility.Get_time - t0;
Dbms_Output.Put_Line ( 'Schema-level table: ' || To_Char ( t_schema_level/100, 9999.9 ) );

  my_tab := my_empty_tab
  /* in general, it would be non-empty from previous use */;
t0 := Dbms_Utility.Get_time;
  for j in 1..nof_rows
    loop
      my_tab ( To_Char(j) ) := j;
    end loop;
t_index_by := Dbms_Utility.Get_time - t0;
Dbms_Output.Put_Line ( 'Index-by table: ' || To_Char ( t_index_by/100, 999.9 ) );
Dbms_Output.Put_Line ( 'Ratio: ' || To_Char ( t_schema_level/t_index_by, 9999.9 ) );
end P;

For tests up to 1,000,000 rows the index-by-varchar2 table is about 20-30 times faster than the schema-level table.

A.2. RECORD BIND CODE SAMPLES

A.2.1. CREATE THE TABLES

Run the following in any schema that has select access to hr.employees.

create table employees_2 as select * from hr.employees where 1=2;
alter table employees_2
  add constraint employees_2_pk primary key (employee_id)
using index;
A.2.2. CREATE THE **Emp_Util** SUPPORTING PACKAGE

Note: this package is written using syntax that is supported at Version 9.0.1 to allow the remaining samples to be attempted there to demonstrate their compilation errors pre Version 9.2.0 without distracting knock-on errors. The function **Get_Many_Rows** would be better written using **BULK SELECT** at Version 9.2.0.

```sql
package Emp_Util is
  type emprec_tab_t is table of employees%rowtype index by binary_integer;
  procedure Show_One ( p_emprec employees%rowtype );
  procedure Show_All ( p_emprecs emprec_tab_t );
  function Get_One_Row return employees%rowtype;
  function Get_Many_Rows return emprec_tab_t;
end Emp_Util;

package body Emp_Util is
  g_table_name    varchar2(30);
  g_initial_count integer;
  g_current_count integer;

  procedure Show_Header
  is
    begin
      Dbms_Output.Put_Line (
        Lpad ( 'employee_id',        15, ' ' ) ||
        Lpad ( 'first_name',         15, ' ' ) ||
        Lpad ( 'last_name',          15, ' ' ) ||
        Lpad ( 'salary',             15, ' ' )
      );
    end Show_Header;

  procedure Show_One ( p_emprec in employees%rowtype )
  is
    begin
      Show_Header;
      Dbms_Output.Put_Line (
        Lpad ( p_emprec.employee_id, 15, ' ' ) ||
        Lpad ( p_emprec.first_name, 15, ' ' ) ||
        Lpad ( p_emprec.last_name, 15, ' ' ) ||
        Lpad ( p_emprec.salary, 15, ' ' )
      );
    end Show_One;

  procedure Show_All ( p_emprecs in emprec_tab_t )
  is
    begin
      Show_Header;
      for j in p_emprecs.First..p_emprecs.Last
        loop
          Dbms_Output.Put_Line (
            Lpad ( p_emprecs(j).employee_id, 15, ' ' ) ||
            Lpad ( p_emprecs(j).first_name, 15, ' ' ) ||
            Lpad ( p_emprecs(j).last_name, 15, ' ' ) ||
            Lpad ( p_emprecs(j).salary, 15, ' ' )
          );
        end loop;
    end Show_All;

  function Get_One_Row return employees%rowtype is
    v_emprec employees%rowtype;
  begin
    select * into v_emprec from employees where employee_id=100;
  end Get_One_Row;
end Emp_Util;
```
function Get_Many_Rows return emprec_tab_t is  
n binary_integer := 0;  
v_avg_hire_date employees.hire_date%type := '25-JUN-97';  
v_emprecs emprec_tab_t;  
begin  
   for j in ( select * from employees where hire_date >= v_avg_hire_date )  
   loop  
     n:=n+1; v_emprecs(n) := j; end loop;  
   return v_emprecs;  
end Get_Many_Rows;  

end Emp_Util;  

-- Sanity check  
begin Emp_Util.Show_All ( Emp_Util.Get_Many_Rows ); end;  

A.2.3. PRE VERSION 9.2.0 WORKAROUND TO POPULATE A TABLE OF RECORDS WITH BULK SELECT  
declare  
   type employee_ids_t is table of employees.employee_id%type  
                 index by binary_integer;  
   type first_names_t is table of employees.first_name%type  
                 index by binary_integer;  
   type last_names_t is table of employees.last_name%type  
                 index by binary_integer;  
   type emails_t is table of employees.email%type  
                 index by binary_integer;  
   type phone_numbers_t is table of employees.phone_number%type  
                 index by binary_integer;  
   type hire_dates_t is table of employees.hire_date%type  
                 index by binary_integer;  
   type job_ids_t is table of employees.job_id%type  
                 index by binary_integer;  
   type salarys_t is table of employees.salary%type  
                 index by binary_integer;  
   type commission_pcts_t is table of employees.commission_pct%type  
                 index by binary_integer;  
   type manager_ids_t is table of employees.manager_id%type  
                 index by binary_integer;  
   type department_ids_t is table of employees.department_id%type  
                 index by binary_integer;  

   v_employee_ids     employee_ids_t;  
v_first_names      first_names_t;  
v_last_names       last_names_t;  
v_emails           emails_t;  
v_phone_numbers    phone_numbers_t;  
v_hire_dates       hire_dates_t;  
v_job_ids          job_ids_t;  
v_salarys          salarys_t;  
v_commission_pcts  commission_pcts_t;  
v_manager_ids      manager_ids_t;  
v_department_ids   department_ids_t;  

   v_emprecs Emp_Util.emprec_tab_t;  

   cursor cur is  
   select
Oracle9i Database

employee_id,
first_name,
last_name,
email,
phone_number,
hire_date,
job_id,
salary,
commission_pct,
manager_id,
department_id
from employees where hire_date >= '25-JUN-97';
begin
open cur;
fetch cur 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
limit 10;
close cur;
for j in 1..v_employee_ids.Last
loop
  v_emprecs(j).employee_id     := v_employee_ids(j);
  v_emprecs(j).first_name      := v_first_names(j);
  v_emprecs(j).last_name       := v_last_names(j);
  v_emprecs(j).email           := v_emails(j);
  v_emprecs(j).phone_number    := v_phone_numbers(j);
  v_emprecs(j).hire_date       := v_hire_dates(j);
  v_emprecs(j).job_id          := v_job_ids(j);
  v_emprecs(j).salary          := v_salaries(j);
  v_emprecs(j).commission_pct  := v_commission_pcts(j);
  v_emprecs(j).manager_id      := v_manager_ids(j);
  v_emprecs(j).department_id   := v_department_ids(j);
end loop;
Emp_Util.Show_All ( v_emprecs );
end;

A.2.4 PRE VERSION 9.2.0 WORKAROUND TO INSERT A SINGLE RECORD
declare
  v_emprec employees%rowtype := Emp_Util.Get_One_Row;
bEGIN
  insert into employees_2 (employee_id,
  first_name,
  last_name,
  email,
  phone_number,
  hire_date,
  job_id,
salary,
commission_pct,
manager_id,
department_id )
values (  
v_emprec.employee_id,
v_emprec.first_name,
v_emprec.last_name,
v_emprec.email,
v_emprec.phone_number,
v_emprec.hire_date,
v_emprec.job_id,
v_emprec.salary,
v_emprec.commission_pct,
v_emprec.manager_id,
v_emprec.department_id )
end;

A.2.5. PRE VERSION 9.2.0 WORKAROUND TO BULK INSERT A RECORD

declare  
bulk_errors exception;
pragma exception_init ( bulk_errors, -24381 );
type employee_ids_t     is table of employees.employee_id%type  
                          index by binary_integer;
type first_names_t      is table of employees.first_name%type  
                          index by binary_integer;
type last_names_t       is table of employees.last_name%type  
                          index by binary_integer;
type emails_t           is table of employees.email%type  
                          index by binary_integer;
type phone_numbers_t    is table of employees.phone_number%type  
                          index by binary_integer;
type hire_dates_t       is table of employees.hire_date%type  
                          index by binary_integer;
type job_ids_t          is table of employees.job_id%type  
                          index by binary_integer;
type salarys_t          is table of employees.salary%type  
                          index by binary_integer;
type commission_pcts_t  is table of employees.commission_pct%type  
                          index by binary_integer;
type manager_ids_t      is table of employees.manager_id%type  
                          index by binary_integer;
type department_ids_t   is table of employees.department_id%type  
                          index by binary_integer;
  
v_employee_ids     employee_ids_t;
v_first_names      first_names_t;
v_last_names       last_names_t;
v_emails           emails_t;
v_phone_numbers    phone_numbers_t;
v_hire_dates       hire_dates_t;
v_job_ids          job_ids_t;
v_salarys          salarys_t;
v_commission_pcts  commission_pcts_t;
v_manager_ids      manager_ids_t;
v_department_ids   department_ids_t;
  
v_emprecs Emp_Util.emprec_tab_t := Emp_Util.Get_Many_Rows;
begin
v_emprecs(1).employee_id :=
    /* contrive "unique constraint violated" error on INSERT */
    v_emprecs(2).employee_id;

for j in 1..v_emprecs.Last loop
    v_employee_ids(j)     := v_emprecs(j).employee_id;
    v_first_names(j)      := v_emprecs(j).first_name;
    v_last_names(j)       := v_emprecs(j).last_name;
    v_emails(j)           := v_emprecs(j).email;
    v_phone_numbers(j)    := v_emprecs(j).phone_number;
    v_hire_dates(j)       := v_emprecs(j).hire_date;
    v_job_ids(j)          := v_emprecs(j).job_id;
    v_salaries(j)         := v_emprecs(j).salary;
    v_commission_pcts(j)  := v_emprecs(j).commission_pct;
    v_manager_ids(j)      := v_emprecs(j).manager_id;
    v_department_ids(j)   := v_emprecs(j).department_id;
end loop;

forall j in v_emprecs.first..v_emprecs.last save exceptions
insert into employees_2 ( employee_id, first_name, last_name, email, phone_number, hire_date, job_id, salary, commission_pct, manager_id, department_id )
values ( v_employee_ids(j), v_first_names(j), v_last_names(j), v_emails(j), v_phone_numbers(j), v_hire_dates(j), v_job_ids(j), v_salaries(j), v_commission_pcts(j), v_manager_ids(j), v_department_ids(j) );
exception when bulk_errors then
    for j in 1..sql%bulk_exceptions.count loop
        Dbms_Output.Put_Line ('Error from element #' ||
            To_Char(sql%bulk_exceptions(j).error_index) || ': ' ||
            Sqlerrm(-sql%bulk_exceptions(j).error_code) );
    end loop;
end;

A.2.6. PRE VERSION 9.2.0 WORKAROUND FOR BULK UPDATE RETURNING A RECORD

declare
    type employee_ids_t     is table of employees.employee_id%type
        index by binary_integer;
    type first_names_t      is table of employees.first_name%type
        index by binary_integer;

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);
  v_emprecs(j).first_name      := v_first_names(j);
  v_emprecs(j).last_name       := v_last_names(j);
  v_emprecs(j).email           := v_emails(j);
  v_emprecs(j).phone_number    := v_phone_numbers(j);
  v_emprecs(j).hire_date       := v_hire_dates(j);
  v_emprecs(j).job_id          := v_job_ids(j);
  v_emprecs(j).salary          := v_salarys(j);
  v_emprecs(j).commission_pct  := v_commission_pcts(j);
  v_emprecs(j).manager_id      := v_manager_ids(j);
  v_emprecs(j).department_id   := v_department_ids(j);
end loop;

Emp_Util.Show_All ( v_emprecs );
end;

A.2.7. PRE VERSION 9.2.0 WORKAROUND FOR BULK UPDATE SET ROW = A RECORD

declare
  v_emprecs Emp_Util.emprec_tab_t :=
      Emp_Util.Get_Many_Rows;

  type employee_ids_t     is table of employees.employee_id%type
      index by binary_integer;
  type first_names_t      is table of employees.first_name%type
      index by binary_integer;
  type last_names_t       is table of employees.last_name%type
      index by binary_integer;
  type emails_t           is table of employees.email%type
      index by binary_integer;
  type phone_numbers_t    is table of employees.phone_number%type
      index by binary_integer;
  type hire_dates_t       is table of employees.hire_date%type
      index by binary_integer;
  type job_ids_t          is table of employees.job_id%type
      index by binary_integer;
  type salarys_t          is table of employees.salary%type
      index by binary_integer;
  type commission_pcts_t  is table of employees.commission_pct%type
      index by binary_integer;
  type manager_ids_t      is table of employees.manager_id%type
      index by binary_integer;
  type department_ids_t   is table of employees.department_id%type
      index by binary_integer;

  v_employee_ids     employee_ids_t;
  v_first_names      first_names_t;
  v_last_names       last_names_t;
  v_emails           emails_t;
  v_phone_numbers    phone_numbers_t;
  v_hire_dates       hire_dates_t;
  v_job_ids          job_ids_t;
  v_salarys          salarys_t;
  v_commission_pcts  commission_pcts_t;
  v_manager_ids      manager_ids_t;
  v_department_ids   department_ids_t;

begin
for j in 1..v_emprecs.Last
loop
  v_employee_ids(j) := v_emprecs(j).employee_id;
  v_first_names(j) := v_emprecs(j).first_name;
  v_last_names(j) := v_emprecs(j).last_name;
  v_emails(j) := v_emprecs(j).email;
  v_phone_numbers(j) := v_emprecs(j).phone_number;
  v_hire_dates(j) := v_emprecs(j).hire_date;
  v_job_ids(j) := v_emprecs(j).job_id;
  v_salaries(j) := v_emprecs(j).salary * 1.2;
  v_commission_pcts(j) := v_emprecs(j).commission_pct;
  v_manager_ids(j) := v_emprecs(j).manager_id;
  v_department_ids(j) := v_emprecs(j).department_id;
end loop;

forall j in v_emprecs.first..v_emprecs.last
update employees
set
  first_name = v_first_names(j),
  last_name = v_last_names(j),
  email = v_emails(j),
  phone_number = v_phone_numbers(j),
  hire_date = v_hire_dates(j),
  job_id = v_job_ids(j),
  salary = v_salaries(j),
  commission_pct = v_commission_pcts(j),
  manager_id = v_manager_ids(j),
  department_id = v_department_ids(j)
where employee_id = v_employee_ids(j);
end;

A.3. UTL_FILE SAMPLES

A.3.1. SET UP THE DIRECTORY AND THE USER

You'll need the cooperation of the DBA and access to a nominated directory on the database machine to try these samples. If you control your own test machine, here is a convenient way to do the setup...

Connect sys/p@9i as sysdba

grant
  resource,
  connect
to programmer identified by p;
grant select on v_$parameter to programmer;

declare v_stmt varchar2(4000);
begin
  -- use the udump dir for convenience
  select value
    into v_stmt
    from v$parameter where name = 'user_dump_dest';
  v_stmt :=
    'create or replace directory UTL_FILE_TEST as ''' || v_stmt || '''';
  execute immediate v_stmt;
end;

grant read on directory UTL_FILE_TEST to programmer;
Connect programmer/p@9i

```
select directory_path from all_directories
   where directory_name = 'UTL_FILE_TEST';
```

Else, of course, any directory you can access will do. It’s always nice to be able to query `v$parameter` (on a test system).

More exhaustive samples are provided online in the PL/SQL section on OTN.

### A.3.2. COPYING, DELETING AND RENAMING FILES

```plsql
declare
    v_line         varchar2(32767);
    c_location     constant varchar2(80) := 'UTL_FILE_TEST';
    c_source_filename  constant varchar2(80) := 'test.txt';
    c_copy_filename    constant varchar2(80) := 'copy.txt';
    c_ren_filename     constant varchar2(80) := 'ren.txt';

procedure Show_Attr ( p_filename in varchar2 ) is
    v_exists            boolean;
    v_file_length       number;
    v_block_size        binary_integer;
begin
    Utl_File.Fgetattr (
        location    => c_location,
        filename    => p_filename,
        fexists     => v_exists,
        file_length => v_file_length,
        block_size  => v_block_size);
    -- Bug #2240685. v_exists is always returned TRUE
    -- But non-existent file has ZERO file_length and block_size
    if not v_exists then Raise_Application_Error ( -20000,
        'Bug #2240685 is fixed' ); end if;
    Dbms_Output.Put_Line ( p_filename             || ' : ' ||
        To_Char(v_file_length) || ' : ' ||
        To_Char(v_block_size ) );
end Show_Attr;

procedure Remove ( p_filename in varchar2 ) is
begin
    Utl_File.Fremove ( location    => c_location,
                       filename    => p_filename );
    exception
    when Utl_File.Delete_Failed then
        Utl_File.Fclose_All;
        Raise_Application_Error ( -20000,
            'Fremove: Delete_Failed trapped' );
    when Utl_File.Invalid_Operation then
        Utl_File.Fclose_All;
        Raise_Application_Error ( -20000,
            'Fremove: Invalid_Operation trapped' );
    end exception;
end Remove;

procedure Full_Copy ( p_source in varchar2, p_dest in varchar2 ) is
begin
    Utl_File.Fcopy ( src_location => c_location,
                       src_filename => c_source_filename,
                       dest_location => c_location,
                       dest_filename => p_dest );
end Full_Copy;
```
procedure Rename ( p_source in varchar2, p_dest in varchar2 ) is
begin
    Utl_File.Frename (
        src_location  => c_location,
        src_filename  => p_source,
        dest_location => c_location,
        dest_filename => p_dest,
        overwrite     => false);
exception when Utl_File.Rename_Failed then
    Utl_File.Fclose_All;
    Raise_Application_Error ( -20000, 'Frename: Rename_Failed trapped' );
end Rename;

begin
    Show_Attr ( c_source_filename );
    Show_Attr ( c_copy_filename );

    Full_Copy ( c_source_filename, c_copy_filename );
    --Full_Copy ( 'not_there.txt', c_copy_filename ) /* Uncomment to see
    --    Utl_File.Invalid_Operation */;
    Show_Attr ( c_copy_filename );

    -- Create ren.txt before running this block to see Utl_File.Rename_Failed
    Rename ( c_copy_filename, c_ren_filename );
    Show_Attr ( c_ren_filename );

    Remove ( c_ren_filename );
    -- Create protected.txt by hand and do "chmod a-x protected.txt"
    --Remove ( 'protected.txt' ) /* Uncomment to see Utl_File.Invalid_Operation */;
end;

A.3.3. HANDLING RAW DATA

This sample demonstrates random piecewise read from a binary file using Fseek and Get_Raw.
We create a file with Put_Raw, writing a buffer of constant length with each successive call. This is conveniently done
for this example by casting a CHAR to a RAW.
Then we re-open the file for reading and use Fseek to position at the start of a record and Get_Raw to read one
record.

declare
    v_record        char(80);
    c_record_size   constant binary_integer := 80;
    v_raw           raw(32767);
    v_raw_size      constant binary_integer := 32767;
    v_handle        Utl_File.File_Type;
    c_location      constant varchar2(80) := 'UTL_FILE_TEST';
c_filename      constant varchar2(80) := 'dest.jpg';

procedure Put_Raw ( p_text in varchar2 ) is begin
  v_record := p_text;
  v_raw := Utl_Raw.Cast_To_Raw ( v_record );
  Utl_File.Put_Raw (  
    file   => v_handle,
    buffer => v_raw,
    autoflush => false );
end Put_Raw;

procedure Get_Raw ( p_offset in binary_integer ) is begin
  Utl_File.Fseek (  
    file            => v_handle,
    absolute_offset => p_offset );

  Utl_File.Get_Raw (  
    file   => v_handle,
    buffer => v_raw,
    len    => c_record_size );

  Dbms_Output.Put_Line (  
    Utl_Raw.Cast_To_Varchar2 ( v_raw ) );
end Get_Raw;

begin
  v_handle := Utl_File.Fopen (  
    location    => c_location,
    filename    => c_filename,
    open_mode   => 'w' /* Create for writing */,
    max_linesize => v_raw_size );

  Put_Raw ( 'Mary had a little lamb' );
  Put_Raw ( 'The cat sat on the mat' );
  Put_Raw ( 'The quick brown fox jumped over the lazy dog' );

  Utl_File.Fclose ( file => v_handle );

  v_handle := Utl_File.Fopen (  
    location    => c_location,
    filename    => c_filename,
    open_mode   => 'r' /* Re-open for reading */,
    max_linesize => v_raw_size );

  Get_Raw ( 2*c_record_size-1 );
  Get_Raw ( 0 );

  Utl_File.Fclose ( file => v_handle );
end;

A.4. PL/SQL DEBUGGING WITH JDEVELOPER VERSION 9.0.3

A.4.1. SCENARIO FOR THE SCREENSHOTS, DATABASE VERSION 9.2.0 IMPLEMENTATION

create table employees as select * from hr.employees;
create table employees_2 as select * from employees
  where l=2;
alter session set plsql_debug=true;
create or replace trigger Emp_Trg
before insert on employees_2
for each row
begin
  if :new.salary < 8000
  then
    Raise_Application_Error ( -20001, 'salary much too low' );
  elsif :new.salary < 10000
  then
    Raise_Application_Error ( -20002, 'salary too low' );
  end if;
end Emp_Trg;
/
create or replace package Emp_Util_920 is
  procedure Insert_Many ( p_limit in natural );
end Emp_Util_920;
/
create or replace package body Emp_Util_920 is
  type emprec_tab_t is table of employees%rowtype index by binary_integer;

  function Select_Many ( p_limit in natural, p_table in varchar2 )
  return emprec_tab_t is
    emprec_tab_t; cur sys_refcursor;
  begin
    open cur for 'select * from ' || p_table;
    fetch cur bulk collect into emprec_tab_t limit p_limit;
    close cur;
    return emprec_tab_t;
  end Select_Many;

  procedure Insert_Many ( p_limit in natural ) is
    bulk_errors exception;
    pragma exception_init ( bulk_errors, -24381 );
    emprec_tab_t := Select_Many ( p_limit, 'employees' );
    empty_emprec_tab_t;
    exception_count pls_integer;
    exception_index pls_integer;
    exception_code pls_integer;
  begin
    delete from employees_2;
    begin
      forall j in emprec_tab_t.First()..emprec_tab_t.Last() save exceptions
      insert into employees_2 values emprec_tab_t(j);
      exception when bulk_errors then
        exception_count := sql%bulk_exceptions.count;
        for j in 1..exception_count loop
          exception_index := sql%bulk_exceptions(j).error_index;
          exception_code := sql%bulk_exceptions(j).error_code;
        end loop;
      end;
      emprec_tab_t := empty_emprec_tab_t;
      emprec_tab_t := Select_Many ( p_limit, 'employees_2' );
    end Insert_Many;
  end Insert_Many;
end Emp_Util_920;
/
A.4.2 SCENARIO FOR THE SCREENSHOTS, DATABASE VERSION 8.1.7 IMPLEMENTATION

Note the much greater code volume and awkwardness of constructs that used to be necessary before the new Version 9.2.0 RECORD bind functionality was introduced. (This difference has nothing to do with the differences in the debugging APIs between Version 8.1.7 and Version 9.2.0, but is due entirely to the differences for RECORDs.)

```
alter session set plsql_debug=true;

create or replace package Emp_Util_817 is
    procedure Insert_Many ( p_limit in natural );
end Emp_Util_817;
/

create or replace package body Emp_Util_817 is
    type emprec_tab_t is table of employees%rowtype index by binary_integer;

    function Select_Many ( p_limit in natural, p_table in varchar2 )
    return emprec_tab_t is
        emprecs emprec_tab_t;
        type cur_t is ref cursor;
        cur cur_t;
        i binary_integer := 1;
        v_stmt varchar2(4000) :=
            'select' || Chr(10) ||
            '  employee_id,' || Chr(10) ||
            '  first_name,' || Chr(10) ||
            '  last_name,' || Chr(10) ||
            '  email,' || Chr(10) ||
            '  phone_number,' || Chr(10) ||
            '  hire_date,' || Chr(10) ||
            '  job_id,' || Chr(10) ||
            '  salary,' || Chr(10) ||
            '  commission_pct,' || Chr(10) ||
            '  manager_id,' || Chr(10) ||
            '  department_id'
        from ' || p_table
        where rownum <= ' || To_Char(p_limit);

    begin
        open cur for v_stmt;
        -- Can't use BULK select with Dynamic SQL before 9.0.1
        loop
            fetch cur into
                emprecs(i).employee_id,
                emprecs(i).first_name,
                emprecs(i).last_name,
                emprecs(i).email,
                emprecs(i).phone_number,
                emprecs(i).hire_date,
                emprecs(i).job_id,
                emprecs(i).salary,
                emprecs(i).commission_pct,
                emprecs(i).manager_id,
                emprecs(i).department_id;
            exit when cur%NotFound;
            i := i + 1;
        end loop;
        close cur;
        return emprecs;
    end Select_Many;
```
procedure Insert_Many ( p_limit in natural ) is
  emprecs          emprec_tab_t  := Select_Many ( p_limit, 'employees' );
  empty_emprecs    emprec_tab_t;
  exception_code  pls_integer;
begin
  delete from employees_2;
  /*
   Though the following syntax compiles and runs OK at 8.1.7, it's no good for the task in hand since the 1st exception raised by the trigger causes a rollback of ALL rows inserted...
   
   forall j in emprecs.First()..emprecs.Last()
     insert into employees_2 values (employee_ids(j),
      first_names(j),
      last_names(j),
      email(j),
      phone_numbers(j),
      hire_dates(j),
      job_Id(j),
      salarys(j),
      commission_pcts(j),
      manager_ids(j),
      department_ids(j));
   
   It would also be uncomfortable because we'd need to define index-by tables for employee_ids, etc and copy them to the target emprec table of records.
   So we need to use an explicit loop rather than the BULK syntax */

  for j in emprecs.First()..emprecs.Last()
    loop
      begin
        insert into employees_2 values (emprecs(j).employee_id,
         emprecs(j).first_name,
         emprecs(j).last_name,
         emprecs(j).email,
         emprecs(j).phone_number,
         emprecs(j).hire_date,
         emprecs(j).job_id,
         emprecs(j).salary,
         emprecs(j).commission_pct,
         emprecs(j).manager_id,
         emprecs(j).department_id);
        exception
          when others then exception_code := SqlCode;
        end;
    end loop;

  emprecs := empty_emprecs;
  emprecs := Select_Many ( p_limit, 'employees_2' );
end Insert_Many;
end Emp_Util_817;
/
A.4.3. EXAMPLE CLIENT APPLICATION WRITTEN IN JAVA

This application calls the procedure illustrated in Appendix A.4.1.

```java
import oracle.sqlj.runtime.Oracle;

public class x {
    public static void main (String args[]) throws java.sql.SQLException {
        // setup the JDBC default connection
        Oracle.connect(x.class, "x_connect.properties");

        #sql { call Emp_Util_920.Insert_Many (5) };
    }
}
```

Instead of editing this source code to add calls to the Dbms_Debug_Jdwp API, presumably to be executed or not according to further custom logic (in response say to a custom environment variable), you can use Oracle’s pre-packaged approach. Leave your application code intact and just set the ora_debug_jdwp environment variable. The above code depends on the x_connect.properties file. Here’s an example...

```
# Users should uncomment one of the following URLs or add their own.
# (If using Thin, edit as appropriate.)
sqlj.url=jdbc:oracle:oci8:@

# User name and password here (edit to use different user/password)
sqlj.user=some_user
sqlj.password=p
```

Note: the ora_debug_jdwp environment variable mechanism requires the OCI driver. If you use the Thin driver, you’ll have to program your own calls to Dbms_Debug_Jdwp.