You wrote *WHAT?*

An examination of common coding mistakes made by PL/SQL developers (like yours truly!) and how you can avoid them yourself.

Steven Feuerstein
steven.feuerstein@quest.com
Love those cursor FOR loops!

• The cursor FOR loop is a very handy construct.
  – Need to iterate through all the rows identified by a cursor?
  – The cursor FOR loop takes care of that for you, with an absolute minimum of effort on your part.

With an implicit cursor....

BEGIN
  FOR rec IN (SELECT * FROM employee)
  LOOP
    process_employee (rec);
  END LOOP;
END;

With an explicit cursor....

DECLARE
  CURSOR emps_cur IS
    SELECT * FROM employee;
BEGIN
  FOR rec IN emps_cur
  LOOP
    process_employee (rec);
  END LOOP;
END;
Hey, I can even fetch one row with CFL!

• I could also use the cursor FOR loop to fetch just a single row.
  – Then I do not have to write the INTO clause, worrying about NO_DATA_FOUND, etc.

```
BEGIN
  FOR rec IN (SELECT * FROM employee
               WHERE employee_id = employee_id_in)
  LOOP
    IF rec.salary > 10000 THEN ...
    ELSE ...
    END IF;
  END LOOP;
END;
```
But, really, why would you do that?

• Cursor FOR loops are very nice constructs, but they have two problems:
  – The row by row processing inherent in a cursor FOR loop is a relatively slow way to retrieve data.
  – The very fact that it does so much for us appeals to our lazy side.

• May I suggest that you....
  – *Never* use a cursor FOR loop to retrieve a single row.
  – Generally consider the cursor FOR loop to be an "old-fashioned" way of doing things, something to be generally avoided.
Never use a CFL for a single row fetch.

- If we *know* we are fetching a single row of data, we should not use a cursor FOR loop.
  - The code *works*, but it is very misleading. There really isn't any loop processing going on. Let's face it - we're just being lazy!

If you are only grabbing a single row, then make sure your code says that.

Otherwise, you are complicating the life of anyone assigned to maintain your code.

```sql
BEGIN
  SELECT * INTO l_employee
  FROM employee
  WHERE employee_id = employee_id_in;

  IF l_employee.salary > 10000 THEN ...
  ELSE ...
  END IF;
  EXCEPTION
  WHEN NO_DATA_FOUND ...
END;
```
You Can Expect More.

Go Modern...Go with BULK COLLECT!

- Generally, if you are running Oracle8i and above, you should strongly consider replacing any and all cursor FOR loops with the BULK COLLECT query.
  - It will be significantly faster.
- And if you are executing any DML inside your loop, you will replace those statements with their FORALL equivalent, also providing a big boost in performance.
- Let's take a look at how you go from the old-fashioned CFL code to bulk processing.
**What is a collection?**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>22</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>def</td>
<td>sf</td>
<td>q</td>
<td>rrr</td>
<td>swq</td>
</tr>
</tbody>
</table>

- A collection is an "ordered group of elements, all of the same type."
  - That's a very general definition; lists, sets, arrays and similar data structures are all types of collections.
  - Each element of a collection may be addressed by a unique subscript, usually an integer but in some cases also a string.
  - Collections are single-dimensional, but you can create collections of collections to emulate multi-dimensional structures.
Three Types of Collections

• Associative arrays (aka index-by tables)
  – Similar to hash tables in other languages, allows you to access elements via arbitrary subscript values.

• Nested tables
  – Can be defined in PL/SQL and SQL. Use to store large amounts of persistent data in the column of a table.
  – Required for some features, such as table functions

• Varrays (aka variable size arrays)
  – Can be defined in PL/SQL and SQL; useful for defining small lists in columns of relational tables.
CREATE OR REPLACE PROCEDURE upd_for_dept ( 
    dept_in IN employee.department_id%TYPE,
    newsal IN employee.salary%TYPE)
IS 
    CURSOR emp_cur IS 
        SELECT employee_id, salary, hire_date 
            FROM employee 
                WHERE department_id = dept_in;
BEGIN 
    FOR rec IN emp_cur 
    LOOP 
        INSERT INTO employee_history 
            (employee_id, salary, hire_date) 
            VALUES (rec.employee_id, rec.salary, rec.hiredate);
        UPDATE employee 
            SET salary = newsal 
                WHERE employee_id = rec.employee_id;
    END LOOP;
END upd_for_dept;
Step 1. Declare a bunch of collections.

A single associative array TYPE and variable for each column selected.

```sql
CREATE OR REPLACE PROCEDURE upd_for_dept (
    dept_in IN employee.department_id%TYPE,
    newsal IN employee.salary%TYPE
) IS
    TYPE employee_tt IS TABLE OF employee.employee_id%TYPE INDEX BY BINARY_INTEGER;
    employees    employee_tt;

    TYPE salary_tt IS TABLE OF employee.salary%TYPE INDEX BY BINARY_INTEGER;
    salaries     salary_tt;

    TYPE hire_date_tt IS TABLE OF employee.hire_date%TYPE INDEX BY BINARY_INTEGER;
    hire_dates   hire_date_tt;
```
Step 2. Replace CFL with BULK COLLECT.

BULK COLLECT the rows for this department into the individual collections

```sql
BEGIN
    SELECT employee_id, salary, hire_date
    BULK COLLECT INTO employees, salaries, hire_dates
    FROM employee
    WHERE department_id = dept_in FOR UPDATE;
END;
```
Step 3. Write one FORALL for each DML.

Use FORALL for each, distinct DML statement to process rows quickly.

BEGIN
    SELECT ... (see previous page) ;

    FORALL indx IN employees.FIRST .. employees.LAST
        INSERT INTO employee_history
            (employee_id, salary, hire_date )
        VALUES (employees (indx), salaries (indx), hire_dates (indx) ) ;

    FORALL indx IN employees.FIRST .. employees.LAST
        UPDATE employee
            SET salary = newsal ,
                hire_date = hire_dates ( indx )
        WHERE employee_id = employees ( indx ) ;

END upd_for_dept;
SQL is generally the key to optimized code.

• I have demonstrated one particular transformation of "old-fashioned" code built around a cursor FOR loop to BULK COLLECT and FORALL.

• Oracle has recently enhanced its SQL language in many ways to improve performance and maintainability.
  – They are outside of the scope of this presentation (and my expertise) and can be overwhelming to keep up with.

• Toad's automated tuning and analysis functionality can help you get up to speed and leverage these new capabilities.
A string is a string is a string? Not quite....

• Actually there are variable length and fixed length, single-byte and multi-byte strings, but let's not quibble.
  – I will assume that you are at least avoiding the use of the CHAR datatype.

• That's good, but perhaps you write code that looks like this:

```sql
DECLARE
  l_last_name VARCHAR2 (100);
  l_full_name VARCHAR2 (500);
  l_big_string VARCHAR2 (32767);
BEGIN
  SELECT last_name, last_name || ', ' || first_name
  INTO l_last_name, l_full_name
  FROM employee
  WHERE employee_id = 1500;
  ...
```
Don't hard-code VARCHAR2 declarations.

- Establish "source definitions" for all your VARCHAR2 declarations and then reference those when declaring your local variables.
- What that code could look like:

```sql
DECLARE
    l_last_name employee.last_name%TYPE;
    l_full_name employee_rp.fullname_t;
    l_big_string plsql_limits.maxvarchar2_t;
BEGIN
    SELECT last_name,
           employee_rp.fullname (first_name, last_name)
    INTO l_last_name, l_full_name
    FROM employee
    WHERE employee_id = 1500;
    ...
```
Supporting code for datatype sources

- Package for employee rules, formulae, related types:

```sql
CREATE OR REPLACE PACKAGE employee_rp
AS
    SUBTYPE fullname_t IS VARCHAR2 (200);

    FUNCTION fullname (
        employee_id_in IN
        employee.employee_id%TYPE
    ) RETURN fullname_t;
END;
```

- Separate package of PL/SQL limits:

```sql
CREATE OR REPLACE PACKAGE plsql_limits
IS
    -- Maximum size for VARCHAR2 in PL/SQL
    SUBTYPE maxvarchar2_t IS VARCHAR2 (32767);

    END plsql_limits;
```

Objective:

Never declare with hard-coded VARCHAR2(N) type... unless it is the "original."
For many years, Oracle "gurus" urged everyone to use explicit cursors all the time, and never, ever use implicit cursors.

```sql
DECLARE
  CURSOR onerow_cur IS
    SELECT * FROM EMPLOYEE
    WHERE EMPLOYEE_ID = employee_id_in;
  l_employee EMPLOYEE%ROWTYPE;
BEGIN
  OPEN onerow_cur;
  FETCH onerow_cur INTO l_employee;
  IF onerow_cur%FOUND THEN ...
  ELSE ...
  END IF;
  CLOSE onerow_cur;
END or_EMPLOYEE;
```

Wrong!
Implicit one row queries are usually faster.

Lessons to learn:

• Don't take "our" word for it. Test claims yourself.
• Assume things will be changing. Don't expose your queries. Hide them behind functions.

```
CREATE OR REPLACE FUNCTION or_employee (employee_id_in IN employee.employee_id%TYPE) RETURN employee%ROWTYPE IS
    retval employee%ROWTYPE;
BEGIN
    SELECT * INTO retval FROM employee WHERE employee_id = employee_id_in;
    RETURN retval;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN retval;
END or_employee;
```
I take exception to (some of) your exceptions.

Exception handling is flexible, powerful -- and vulnerable to abuse.

- Here's a good rule: write well-structured code.
- The exception: aw, what the heck - who's going to notice?

```sql
CREATE OR REPLACE FUNCTION matching_row (
    list_in IN strings_nt, value_in IN VARCHAR2
) RETURN PLS_INTEGER IS
    exit_function EXCEPTION;
BEGIN
    FOR j IN list_in.FIRST .. list_in.LAST LOOP
        IF list_in (indx) = value_in THEN
            RETURN indx;
        END IF;
    END LOOP;
    RAISE exit_function;
EXCEPTION
    WHEN exit_function THEN RETURN NULL;
END;
```
Raise exceptions, never actions!

• Examine the names of user-defined exceptions.
• If they sound like actions ("return value" "calculate total", etc.) then the programmer is very likely abusing the exception handling mechanism of PL/SQL.
• So remember....

No GOTOs

No exceptions like GOTOs
Don't assume you haven't made assumptions

- I am using collections - how exciting!
- I need to do a "full collection scan".
- No problem - here comes the FOR loop.

CREATE OR REPLACE FUNCTION display_contents (collection_in IN my_pkg.collection_type) IS
  indx PLS_INTEGER;
BEGIN
  FOR indx IN collection_in.FIRST .. collection_in.LAST LOOP
    -- Display contents of a row.
    DBMS_OUTPUT.PUT_LINE (collection_in (indx).name));
    ...
  END LOOP;
END display_contents;

What assumptions am I making in this program?
Things to keep in mind with collections...

• Touch a row that doesn't exist and Oracle raises the NO_DATA_FOUND exception.
• Associative arrays may be sparse (gaps between defined rows).
• FOR loops aren't smart about collections.
• And some non-collection issues...
  – If low or high range values are NULL, then Oracle raises VALUE_ERROR exception.
  – Don't declare a local variable for the FOR loop index. It's done for you. This extra code can allow errors to creep into code later.
Assumption-less code (more or less)

- Now it is harder for the next coder to accidentally introduce bugs into the application.

```sql
CREATE OR REPLACE FUNCTION display_contents (collection_in IN my_pkg.collection_type) IS
  l_row PLS_INTEGER;
BEGIN
  l_row := collection_in.FIRST;
  WHILE (l_row IS NOT NULL) LOOP
    -- Display contents of a row.
    DBMS_OUTPUT.PUT_LINE (collection_in (l_row).name));
    ...
    l_row := collection_in.NEXT (l_row);
  END LOOP;
END display_contents;
```

Replace FOR loop with WHILE loop. Only touch defined rows.
Cut-and-paste - down the slippery slope.

• Cut-and-paste sure is a handy feature of a Windows and other GUIs.
  – But C-A-P can also lead to truly awful code.
  – Like cursor FOR loops, just because it is easy and saves some key strokes, does not make it better.

```sql
PROCEDURE show_percentages (sales_in IN sales$%ROWTYPE, total_in IN NUMBER)
IS
BEGIN
  food_sales_stg :=
    TO_CHAR ((sales_in.food_sales / total_in) * 100, ' $999,999');
  service_sales_stg :=
    TO_CHAR ((sales_in.service_sales / total_in) * 100, ' $999,999');
  toy_sales_stg :=
    TO_CHAR ((sales_in.toy_sales / total_in) * 100, ' $999,999');
END show_percentages;
```
• Set a very simple rule for yourself: No executable section will have more than 50 lines of code.
  – Use local modules and packaged code to keep program units small, testable and easy to maintain.

```sql
PROCEDURE show_percentages (sales_in IN sales%ROWTYPE, total_in IN NUMBER)
IS
  FUNCTION formatted_pct (val_in IN NUMBER)
  RETURN VARCHAR2
  IS
    BEGIN
      RETURN TO_CHAR ((val_in / total_in) * 100, '$999,999');
    END;
    BEGIN
      food_sales_stg := formatted_pct (sales_in.food_sales);
      service_sales_stg := formatted_pct (sales_in.service_sales);
      toy_sales_stg := formatted_pct (sales_in.toy_sales);
    END show_percentages;
```
Making mistakes is a part of the game.

- As long as there are programmers and programs, we will make mistakes and have to fix bugs.
  - All we can do is keep them to a minimum.
- So keep the following in mind....
  - Don't repeat things.
  - Your code is your legacy, and your offspring may have to maintain your code.
  - Concentrate on readability, not cleverness.
- Visit www.oracleplsqlprogramming.com to download any and all of my training materials and accompanying scripts.

AND TEST OUR CODE.
Check out utplsql via www.ounit.com.

AND sign up for OPP/News.
Oracle PL/SQL Programming conference!

- A two-day conference packed with intensive trainings on the PL/SQL language.
- A celebration of the 10th anniversary of the publication of Oracle PL/SQL Programming:

Sponsored by Quest Software. More information available at:

www.oracleplsqlprogramming.com