**On Cursors, Context Switches, and Mistakes**

*By Steven Feuerstein Oracle ACE Director*

**There are good ways, and other ways, to get data from tables.**

|  |
| --- |
| **Answer to Last Issue’s Challenge**    The PL/SQL Challenge quiz in last issue’s “[Dynamically Dangerous Code](http://www.oracle.com/technetwork/issue-archive/2015/15-may/o35plsql-2541606.html)” article created and populated two tables, presented a code block, offered four different procedures, and asked which procedures would display both “Oak” and “Sam.”  All four answers—A, B, C, and D—are correct, but only D addresses basic SQL injection concerns and avoids unnecessary dynamic PL/SQL execution. |

Things have recently settled down at extremememe.info. Rather than work with the development team side by side, I have shifted to serving as a sounding board for developers. I stop by once a week, hang out my shingle (“The Man Who Knows Only PL/SQL,” aka TMWKOP) at an available cubicle, and see what the day brings me.

Usually it brings me PL/SQL code, which is a good thing, because I am not to be trusted with anything else. Sometimes that code is bright and shiny: the developer is there to essentially show off some nice code.

Sometimes, however, that code is a twisted pretzel of good intentions warped by tough deadlines that send developers scurrying for shortcuts that will, they think, save them a little time. And every second counts.

It’s really tough when I am presented with a twisted pretzel of code that the developer believes is actually bright and shiny. Welcome to the world of Coding Diplomacy.

“Check this out, Steven,” said Marsha. “I figured out a way to reuse existing code and avoid declaring variables when I can get PL/SQL to do it for me.”

I did not like the sound of that at all. I am all for code reuse and all for getting Oracle Database technology to do as much heavy lifting as possible. I just don’t think that being able to avoid declaring a variable should be something to get excited about. It’s likely an indication of deeper dysfunction.

But I smiled. A big smile. Marsha is a new developer, both at extremememe.info and to PL/SQL. The fact that she’s excited about her code means that she *cares* about her code. That’s always a good thing and always to be encouraged.

“Lay it on me, Marsha. Show me what you’ve got,” I said. And she showed me the em\_meme\_reporter procedure (in Listing 1).

**Code Listing 1:** Marsha’s em\_meme\_reporter procedure

CREATE OR REPLACE PROCEDURE em\_meme\_reporter (

meme\_id\_in IN em\_memes.meme\_id%TYPE)

IS

BEGIN

FOR one\_meme

IN (SELECT title,

em\_meme\_summary (meme\_id) summary,

em\_meme\_history (meme\_id) history

FROM em\_memes

WHERE meme\_id = meme\_id\_in)

LOOP

DBMS\_OUTPUT.put\_line (one\_meme.title);

DBMS\_OUTPUT.put\_line (one\_meme.summary);

DBMS\_OUTPUT.put\_line (one\_meme.history);

END LOOP;

END;

/

I instantly saw what she meant about avoiding a declaration—and I didn’t like it one bit. But I have learned that before going into critique mode, it is best to gather all the facts.

“I can see that you are reusing a couple of functions,” I told Marsha in a neutral voice. “Could you show those to me as well?”

“Sure,” she said, adding, “I didn’t write these.” I wasn’t sure if I was supposed to take that as a warning or as an expression of humility.

She showed me the em\_meme\_summary function and the em\_meme\_history function (in Listings 2 and 3, respectively).

**Code Listing 2:** The em\_meme\_summary function (called by em\_meme\_reporter)

CREATE OR REPLACE FUNCTION em\_meme\_summary (

meme\_id\_in IN em\_memes.meme\_id%TYPE)

RETURN VARCHAR2

IS

CURSOR summary\_cur

IS

SELECT stats\_summary

FROM em\_meme\_stats\_mv

WHERE meme\_id = meme\_id\_in;

l\_return VARCHAR2 (32767);

BEGIN

OPEN summary\_cur;

FETCH summary\_cur INTO l\_return;

CLOSE summary\_cur;

RETURN l\_return;

END;

/

**Code Listing 3:** The em\_meme\_history function (called by em\_meme\_reporter)

CREATE OR REPLACE FUNCTION em\_meme\_history (

meme\_id\_in IN em\_memes.meme\_id%TYPE)

RETURN VARCHAR2

IS

l\_return VARCHAR2 (32767);

BEGIN

SELECT LISTAGG (a.activity\_date || ' - ' || a.description,

CHR (10))

WITHIN GROUP (ORDER BY a.activity\_date)

INTO l\_return

FROM em\_meme\_activities a

WHERE meme\_id = meme\_id\_in;

RETURN l\_return;

END;

/

“I especially like the em\_meme\_history function,” said Marsha. “That LISTAGG feature is way cool.”

I kept a pleasant smile on my face and nodded as I mentally formulated my list of concerns. I encourage my readers to do the same-—do not read further until you review the three programs in Listings 1, 2, and 3 and sort out what you think could be improved. Then compare your list with mine.

**Criticize Constructively**

There’s no shortage of ways to improve Marsha’s program—just as there is no shortage of ways to improve many of the packages and subprograms that I (TMWKOP) have written over the years.

For Marsha’s program, I put together two lists in my head: first, the list of things Marsha has done well and for which she should be complimented and, second, the list of areas of concern, of what could be improved. I then carefully filtered and organized the latter to maximize the impact.

“Marsha,” I started, “I love your programmer instincts. They are on target and will serve you well. You recognize that keeping code volume to a minimum is generally a solid goal to aim for. Acting on that principle, you found a way to avoid a declaration by using the cursor FOR loop. Before writing all the logic in your program, you also checked around to see if someone else had beaten you to it: was the algorithm or the information you needed already implemented? I wish that more PL/SQL developers carried that attitude into their coding sessions.”

Marsha beamed at me. She was happy and receptive. Now it was time to focus on areas of improvement. I organized the list in my head, and sorted out the best way to take Marsha through them:

* PL/SQL is often not the best reporting mechanism.
* Cursor FOR loops for single-row fetching is “bad lazy.”
* Don’t hide possible bugs.
* Avoid unnecessary context switches.
* Separate data access from data formatting.

**PL/SQL as a Reporting Mechanism?**

Of course, I am biased, but I believe that PL/SQL is the best database programming language in the world, in no small part because the original designers of PL/SQL were smart enough to use the Ada programming language as the “template” for PL/SQL.

But notice that I did not say PL/SQL is the “best programming language.” It certainly is not. It is a special-purpose language designed primarily to make it as easy as possible to write efficient, secure, maintainable applications on top of Oracle Database (that is, on “top” of SQL).

I asked Marsha, “Who will be using this report, and how do they call it?”

“Right now, it’s just for the dev team, so we can quickly and easily see the status of a meme. But my manager did say that maybe later we will offer the report to users.”

“Well, we all know what ‘maybe later’ means, right, Marsha?” I smiled at her, and she smiled back and shrugged.

“Here’s the thing,” I said. “Developers use the DBMS\_OUTPUT package to send output to the screen, so they are inclined to use it for reports. That might be OK for relatively simple reports for internal developer use.

“For reports that involve any amount of complex formatting and will be user-facing, though, I suggest that you look at alternatives, such as [Oracle Business Intelligence Publisher](http://www.oracle.com/technetwork/middleware/bi-publisher/overview/index.html).

“So let’s continue looking at your program, but I do suggest that you check in with your manager about her vision of where this program will end up. If it really is going to end up as a significant user report, I’d consider switching the ‘delivery’ mechanism sooner rather than later.”

**Cursor FOR Loops for Single-Row Fetches: Bad Lazy**

“Marsha, you were happy to avoid declaring a variable,” I continued, “Please explain.”

“You bet!” Marsha started. “I just *love* the cursor FOR loop. The compiler implicitly declares the iterator record of the right type. I don’t have to open, fetch, figure out when to exit, and close. So I used the cursor FOR loop to find the row of information I needed for the meme and then displayed that info. No local variables declared!”

“Gotcha,” I said. “I appreciate the cursor FOR loop as well. To me, it speaks to how well-designed and implemented PL/SQL is as a database programming language. It’s a high-level declarative construct, like SELECT itself. We ask PL/SQL to fetch each row and apply the code in the body of the loop to that row. We don’t tell PL/SQL how to do the fetching. And that means, among other things, that the PL/SQL optimizer is free to change the way it does its fetching to improve performance.

“So that’s all great—*but* I think you are making a big mistake by using the cursor FOR loop here.”

“Huh?” said Marsha, the smile fading from her face as she absorbed my last sentence.

To soften the blow, I said, while typing, “I would say the same thing if you showed me this numeric FOR loop. Tell me what *you* think of it.”

BEGIN

FOR indx IN 1 .. 1

LOOP

DBMS\_OUTPUT.put\_line (indx);

END LOOP;

END;

“Well. . .” said Marsha slowly, “that’s just silly code. Why wouldn’t you just call DBMS\_OUTPUT.put\_line to display 1?”

“Exactly!” I replied. “That is exactly the problem with that loop—and with yours as well. Can you see why?”

Marsha looked at her program, frowning. Then her expression cleared. “I get it! I am producing a report for a single meme, so only one row is being fetched from the em\_memes table. The body of the loop always executes only once.”

“Ideally, yes, once and only once,” I said. “A cursor FOR loop that fetches just one row is like a numeric FOR loop that goes from N to N. And neither of them should be loops.

“Let’s face it, Marsha: you were just being lazy. And,” I continued before she could take offense, “I think that being lazy is *not* a character flaw for a programmer. But you have to be lazy in the right—that is, *smart*—way.

“Good lazy: hide a SELECT behind a function to avoid having to type it more than once. Bad lazy: abusing a language construct so that Oracle Database will declare a variable for you.” Marsha looked a little embarrassed.

“Oh, and by the way,” I told Marsha, “there’s another problem with using a cursor FOR loop this way: you might hide a bug.”

**Don’t Hide Possible Bugs**

Every program a developer writes makes an assumption. When that assumption fails, the program usually fails with an error. That’s bad enough. Even worse, however, is when the assumption fails but no error reflecting that failure is communicated to the user or to the error log.

I told Marsha, “In em\_meme\_reporter, you assume that there is just one row in em\_memes for a given meme ID.”

“Well, sure I do,” she said. “There’s a primary key on the table. There couldn’t possibly be more than one row in the table with the same meme ID.”

“You mean that you *assume* that the primary key is enabled on the table, right?”

Marsha looked at me like I was crazy. “I guess so,” she said slowly. “But why wouldn’t it be enabled?”

“Because someone disabled it by accident?” I gestured to indicate the entire extremememe.info workplace. “What? You think your coworkers don’t ever make mistakes? Accidents happen all the time. Things change, and no one notices—until the consequence of a change smacks us in the face.”

Marsha looked distinctly unconvinced.

“Sure, it’s a long shot to think that the primary key would be disabled on the em\_memes table—in production,” I said. “But in development and testing, controls are usually looser. Someone might, for example, decide to turn off the primary key constraint to make it easier to refresh test data.”

Marsha looked over her code. Then she said, “OK, I can see that in terms of the cursor FOR loop. But I call em\_meme\_summary inside the query. If there were multiple rows with the same meme ID, the function would fail with TOO\_MANY ROWS errors, right?”

I smiled. Marsha was a fast thinker. “You’d like to think so, right? But in fact, no. Notice that em\_meme\_summary [Listing 2] relies on an explicit cursor rather than a SELECT-INTO to fetch that one row. Using SELECT-INTOs is, generally, going to be the fastest way to fetch a single row, so the explicit cursor is not only a bad idea for performance but is also a mistake, because it will hide TOO\_MANY\_ROWS errors.

“That’s one of the other reasons—besides performance—why you should always use a SELECT-INTO—and certainly not a cursor FOR loop—when fetching what you believe should always be a single row: the PL/SQL runtime engine will raise TOO\_MANY ROWS.”

Because Marsha had mentioned the em\_meme\_summary function she was calling inside her loop’s query, it was time to move on to my next concern.

“Oh, and by the way, you should not be reusing those functions—em\_meme\_summary and em\_meme\_history—inside your query. I suppose,” I finished with a sigh, “that’s another—and worse—example of bad lazy, and it could affect both the performance and correctness of your program.”

**Avoid Unnecessary Context Switches**

In my trainings at extremememe.info and other locations, I urge developers to hide SELECT statements behind functions and also to reuse code whenever possible. Given that, it is not surprising that Marsha took the approach she did. But any piece of advice must always be applied with common sense and a broader understanding of your objectives.

“It’s great that you wanted to reuse existing code, Marsha,” I explained. “You’ve just got to keep in mind that reuse—along with a reduced volume of code to maintain—is only one objective, among many, for a successful application. I hate to say it, but sometimes pursuing one objective might interfere with another. In this case, you prioritized reuse, but at the expense of performance and increased complexity of code.

“So,” I continued, “let’s talk about context switches.”

The PL/SQL language is tightly integrated *syntactically* with SQL: you can write native SQL statements inside PL/SQL, and almost all built-in SQL functions are also available as native PL/SQL functions.

But when it comes to executing PL/SQL code and executing SQL statements, you are looking at two different engines. The PL/SQL runtime engine executes PL/SQL blocks and the statements within them. The SQL engine executes SQL statements.

“That’s straightforward enough, right?” I asked Marsha. She nodded. “But then it gets tricky. What happens when the PL/SQL engine encounters a SQL statement smack-dab in the middle of the PL/SQL block?”

Marsha thought about it for a moment. “I guess it must hand over the SQL statement to the SQL engine for processing.”

“Exactly right! It’s called a *context switch*. The PL/SQL engine passes the SQL statement, along with bind variable values, over to the SQL engine. The SQL engine does its thing and then passes the results back to the PL/SQL engine, which continues right along.

“Next question: what happens when the SQL engine is executing a query and it comes across a user-defined function, such as em\_meme\_history?”

Marsha was clearly no dummy. She answered in an instant: “The same in reverse. The SQL engine passes the function call, along with the values from the SQL statement—passed as arguments—to the PL/SQL engine.”

I beamed at Marsha. I like smart students, and I like smart PL/SQL developers even better.

“Final question: what happens when a PL/SQL block executes a query that calls a user-defined function that, in turn, executes a SQL statement?” I asked.

“Lots and lots of context switches?”

“Yes, indeed, a whole mess of context switches, and each switch is a delay. So the more you switch, the slower your code goes. Remember FORALL from that training I did a few months ago? Reduction of context switches was the primary design goal of that feature.”

Marsha looked over her code. She nodded. “Yep, so that’s exactly what’s going on here. I call those functions in my SELECT because—well, heck, why write that logic again? But they, in turn, each execute their own SELECTs. OK, I can see the problem. What do you think I should do about it?”

“Before I get to that,” I said, “I have one more piece of advice: it is critical to hide SELECTs behind functions, especially in those queries that are commonly used throughout your PL/SQL code. It’s really the only way to avoid copy/pastes of queries and the only way to ensure that your code is maintainable. But even more important is that you do as much work as you can in *native* SQL.

“And that advice can be hard to follow—or you can be tempted *not* to follow it—when you have at hand a whole bunch of functions that fetch data for a variety of common scenarios. That’s the situation you ran into, Marsha. OK, so what should you do? I suggest restructuring your em\_meme\_reporter procedure [Listing 4].”

**Code Listing 4:** The revised em\_meme\_reporter procedure

CREATE OR REPLACE PROCEDURE em\_meme\_reporter (

meme\_id\_in IN em\_memes.meme\_id%TYPE)

IS

l\_title em\_memes.title%TYPE;

l\_stats\_summary em\_meme\_stats\_mv.stats\_summary%TYPE;

BEGIN

SELECT m.title, s.stats\_summary

INTO l\_title, l\_stats\_summary

FROM em\_memes m, em\_meme\_stats\_mv s

WHERE m.meme\_id = em\_meme\_reporter.meme\_id\_in

AND m.meme\_id = s.meme\_id;

DBMS\_OUTPUT.put\_line (l\_title);

DBMS\_OUTPUT.put\_line (l\_stats\_summary);

FOR activity\_rec

IN ( SELECT a.activity\_date || ' - ' || a.description

one\_activity

FROM em\_meme\_activities a

WHERE a.meme\_id = em\_meme\_reporter.meme\_id\_in

ORDER BY a.activity\_date)

LOOP

DBMS\_OUTPUT.put\_line (activity\_rec.one\_activity);

END LOOP;

END;

/

I then went over the changes I made to the em\_meme\_reporter procedure.

|  |
| --- |
| **Next Steps**    **DOWNLOAD** [**Oracle Database 12*c***](http://www.oracle.com/technetwork/database/enterprise-edition/downloads/index.html) [**PL/SQL code for this article**](http://www.oracle.com/technetwork/issue-archive/2015/15-jul/o45plsql-2606901.zip)  **TEST your**[**PL/SQL knowledge**](http://www.plsqlchallenge.com/)  **READ more Feuerstein** [**bit.ly/omagplsql**](http://www.oracle.com/technetwork/issue-archive/index-087690.html) [**stevenfeuersteinonplsql.blogspot.com**](http://stevenfeuersteinonplsql.blogspot.com/)  **READ more about**[**PL/SQL**](http://www.oracle.com/plsql) |

“First, I execute a SELECT-INTO to retrieve the title from em\_ememes and the statistics summary from the em\_emem\_stats\_mv materialized view. No need to call a separate function. Instead, I join the two tables together on the meme\_id and, with one context switch, get all that data.

“Notice that I do *not* also include a join to em\_meme\_activities. Instead, in the em\_meme\_reporter procedure, I rely on the cursor FOR loop construct we both know and love to simply retrieve each row of activity for that meme and then display it. Very straightforward, easy to understand and maintain, avoiding unnecessary context switches. Again, this is assuming that you want to stick with a PL/SQL procedure as your reporting mechanism.”

**Separate Data Access from Data Formatting**

Marsha seemed OK with the em\_meme\_reporter procedure rewrite, so I moved on to my final piece of advice: “Even if you are going to use PL/SQL as a reporting engine,” I told her, “you should clearly separate the code used to retrieve data and the code used to format that data in the report.”

Using LISTAGG is a nice way to aggregate data from multiple rows into a single “list.” In the em\_meme\_history function, the author specifies CHR(10) as the list separator. CHR(10) is the line-feed character, so what this function really does is return multiple rows of data on multiple lines in the report.

Developers should take care to keep user interface rules and code separate from business rules and data. And the same rule applies to formatting for a report: put all such logic in the report generator, and keep it out of the code used to retrieve the data for the report.

Marsha grinned widely. “I’m going to be sure to tell that to Sam, who wrote the em\_meme\_history function. He thinks he’s such a hotshot.”

I figured it would be smarter for Marsha to first find out how the function is currently being used: never rush to judgment! But the best way to learn how to write code—*and* how to relate to other humans—is to make mistakes and be corrected. So I left her to it.

**Key Lessons Learned**

As is so often the case, the smallest and, at first glance, reasonable program can offer many lessons for code optimization. Marsha and I walked through em\_meme\_reporter and came away with the following checklist, from which I think all PL/SQL developers can benefit:

* Never use a cursor FOR loop to fetch a single row.
* Never use an explicit cursor to fetch a single row.
* Avoid writing code that hides bugs.
* User-defined functions executed inside a SQL statement should never contain their own SQL statements.
* Keep user interface and report formatting logic separate from data retrieval and business rule logic.

|  |
| --- |
| **Take the Challenge**    Each PL/SQL article offers a quiz to test your knowledge of the information provided in that article. The quiz appears below and also at PL/SQL Challenge ([plsqlchallenge.com](http://www.plsqlchallenge.com/)), a website that offers online quizzes on PL/SQL as well as on SQL, Oracle Application Express, database design, and deductive logic.  Here is your quiz for this article:  I execute the following statements:  CREATE TABLE plch\_memes  (  meme\_id INTEGER,  title VARCHAR2 (100),  description VARCHAR2 (4000)  )  /  BEGIN  INSERT INTO plch\_memes  VALUES (  1,  'Cats doing somersaults',  'When that cat flips, the world flips with it.');  INSERT INTO plch\_memes  VALUES (  1,  'Cats being the boss',  'Who really “owns” whom?');  COMMIT;  END;  /  Which of the following choices create a function named PLCH\_MEME\_TITLE so that “Cats” and some other text is displayed on your screen after execution of this block?  BEGIN  DBMS\_OUTPUT.put\_Line (plch\_meme\_title (1));  END;  /  **a.**  CREATE OR REPLACE FUNCTION plch\_meme\_title (  meme\_id\_in IN plch\_memes.meme\_id%TYPE)  RETURN VARCHAR2  IS  l\_return plch\_memes.title%TYPE;  BEGIN  SELECT title  INTO l\_return  FROM plch\_memes  WHERE meme\_id = meme\_id\_in;  RETURN l\_return;  END;  /  **b.**  CREATE OR REPLACE FUNCTION plch\_meme\_title (  meme\_id\_in IN plch\_memes.meme\_id%TYPE)  RETURN VARCHAR2  IS  CURSOR title\_cur  IS  SELECT title  FROM plch\_memes  WHERE meme\_id = meme\_id\_in;  l\_return plch\_memes.title%TYPE;  BEGIN  OPEN title\_cur;  FETCH title\_cur INTO l\_return;  CLOSE title\_cur;  RETURN l\_return;  END;  /  **c.**  CREATE OR REPLACE FUNCTION plch\_meme\_title (  meme\_id\_in IN plch\_memes.meme\_id%TYPE)  RETURN VARCHAR2  IS  BEGIN  FOR rec IN (SELECT title  FROM plch\_memes  WHERE meme\_id = meme\_id\_in)  LOOP  RETURN rec.title;  END LOOP;  END;  / |