Recursive PL/SQL - or - Cut the Fat Out of SQL*Forms 3.0

Stanley J. Sewall, Internal Revenue Service

PL/SQL is the programming language for Triggers and Procedures for SQL*FORMS 3.0. By combining the recursive capabilities of programming languages such as C or Lisp, SQL*FORMS 3.0 augments PL/SQL's flexibility within the API (Application Programming Interface) SQL*FORMS 3.0. This paper describes how you can use PL/SQL to reduce procedures, thus increasing the performance of the API.

■ Explanation

One of the overall considerations for this presentation is dislodging the paradigm for PL/SQL. All the documentation that I have encountered for Oracle's PL/SQL Language illustrates how to separate distinct operations and programming routines into separate Oracle Procedures. This, of course, demands adding the number of Procedures in SQL*FORMS 3.0. Although this will improve the performance time of the application, this methodology can create a tremendous amount of Oracle Procedures, thus increasing the size of the .inp file.

This paper deals specifically with a PL/SQL Procedures to recursiveally call itself. The paper does not have any PL/SQL LOOP's. The primary purpose is to control the behavior of SQL*FORMS by reiterative calls to a single control procedure.

Any programmer can tell you about the nightmares of having to go through someone else's code, and how potentially difficult it is to correct problems, if the original creator is not there to assist with the maintenance of code. The actual consideration of recursion was brought forth by doing maintenance of existing SQL*FORMS 3.0 applications that required changes in the code.

The premise behind the implementation of this methodology is to centralize the conditional statements, reduce the querying of ORACLE RDBMS, and to allow an easier flow for the data structures in the SQL*FORMS. Since all computer hardware systems have limited resources, these considerations are taken with great painstaking care. (Yeh.. right). Programmers have their own individual style of programming in PL/SQL. Once a programmer inherits an application, one of the first tasks he begins doing is converting the code into a more recognizable style of his/her own. However, there is nothing wrong with doing it your own way. I have always taken the philosophy that one should try new implementations of methodologies and test the performance of the applications. Usually, a programmer would look at this and say, "That's not important, "it's trivial," or "it won't work with multi-page applications;" those are the usual responses for implementing a new methodology. Breaking the paradigm of this type of reaction demands a sometimes unorthodox approach, usually arising throughout the entire life cycle. More ideas are dismissed because they seem trivial or non-important. The recursion technique is not for the light-hearted or for a programmer who needs to rapidly build or modify an application. Once you "get it," future applications become easier to build and to maintain.

The programmer has to look at the recursive PL/SQL as a control data structure implemented in the fashion of a SQL*FORMS Procedure. This control structure contains the SELECT, conditional statements, and EXCEPTIONS to control a majority of SQL*FORMS application behaviors. The data structure is an organizational schema, applied to a record or an array, which causes data to be interpreted so that specific operations can be performed upon the data, and then the record is updated to the database.

The most important element to remember is that analyzing the current approach and researching involves a great deal of time and effort. In implementing recursive calls, I first studied Common-Lisp and C++, which allow for recursion. I noticed the similarities between both of the recursive implementations could use recursion for SQL*FORMS. Although Oracle PL/SQL does not fully have the support capabilities, the PL/SQL Language does facilitate the implementations of recursion.

The methodology of the consistency testing also facilitates the means of recursion of SQL*FORMS 3.0. At our develop site, one of the existing standards is the testing of the conditional statements. Another standard used at our site is screen-level testing, which allows the user to input data onto the screen. After all the data are entered onto the screen, we check those data on the screen, against the Oracle Procedure. This allows for faster input of the screens for the user. This methodology has a great deal to do with the development of recursive PL/SQL, because it allows for all the testing of the database values and the SQL*FORMS field on the API.

■ Recursion in SQL*FORMS 3.0.

As I stated previously, Oracle instructs programmers to construct groups of procedures to execute tasks in the application. Therefore, the first thing to remember is to throw out the old paradigm. That is, check all preconceived notions at the door.

The overall premise of this section is to reduce the amount of procedure calls and to get one procedure to act as a data structure in order to control a great deal of SQL*FORMS without sacrificing too much application performance.

In the first schema, we will use one screen with one data structure to control the data on the screen. In this case, it is usually an ORACLE Procedure. On top of the Oracle Procedure, we usually do a SELECT to OPEN a cursor to receive values from the database; second is the conditional constraints that are in the program for field value error detection.

We will use the screen-level methodology to test constraints for the sake of this illustration. The screen-level methodology prompts the user to input all the data onto the screen; then, at the last field, the data on the screen are all checked at one time. The exception is for special circumstances, where tests have to be run at the field level. The condition where we have to pull some data from the database to test the conditional statements, we first have a SELECT statement to pull the data from the database, based upon a key index. Then, the procedure would contain the conditional statements, to check the constraints against the SQL*FORMS PL/SQL. If an exception is encountered in the conditional statements, we perform an EXCEPTION within the procedure, and send the cursor to the field in error. Sounds simple, a pretty routine. I think I would agree, too.

After sending the cursor to a field that has an error, you have several nice options. You can prompt to make a change in a field, then continue editing, or press a "hot" key to go down to the end of the screen and through the test again. Sounds boring. Well fortunately, we are only limited by our imaginations.

The SQL*FORMS behavior control is paramount, because, at this time, the programmer has sent the cursor through the same PROCEDURE. At the pop-up window, if the user decides to accept the computed value of the pop-up window, the programmer executes the same Oracle Procedure and passes the function different parameters or sets of the GLOBALs or Control fields to check for conditional statements. Thus, by setting the parameters to different numbers, the Oracle Procedure can do different operations.

Another example of recursively calling the same PROCEDURE to perform different task is passing the PROCEDURE parameter(s) to perform certain tasks or to bypass a conditional statement that may have already been executed within the PROCEDURE. Within a PROCEDURE a wide variety of functions can be performed that can be localized by the programmer. Most of the time, programmers like to increase the appeal of the application by applying pop-up windows, help messages, and automatic settings to fields that may compute values for the screen. But the overall effectiveness is with the Oracle pop-up window. When the PROCEDURE is executed, the conditional statement which contains GO_FIELD function, is executed and the cursor is sent to the pop-up window. At the pop-up window, the user is given a choice to accept the computed value or to go back to the field to re-enter the field on the screen. What if we used a pop-up page to force the user to input an alternative value. How would that look? The initialization of the field would be performe;, then the Oracle Procedure would be executed again. By using a "hot" key to initialize the GLOBAL or control field to a different value, the programmer can send the cursor into the Oracle Procedure, therefore, performing a different part of the Oracle Procedure.

Controlling the behavior of a SQL*FORMS 3.0 applications relates to reseting the :GLOBALs or Control fields in the KEY-STARTUP. Each GLOBAL or Control field will be set depending on the conditional statements exhibited in SQL*FORMS. When the GLOBALs or Control fields are set to different numbers, the programmer executes the PROCEDURE to perform that part of the test or to bypass the test. This method is easier to implement, and you have greater control of SQL*FORMS. If there is any problem in the SQL*FORMS application, the programmer knows that he can start to trace the problem in the KEY-STARTUP trigger.

The Oracle Procedure in this methodology performs a SELECT to retrieve the same values from the database every instance the Procedure is initialized in the program. The only problem with this implementation, if you would like to minimize the amount of querying the database, is that it would be laborious. This is because every time the cursor goes through the Oracle Procedure, an Oracle Select must be performed. Therefore, another viewpoint would be to check conditional statements within the Oracle Procedure and to pass the parameters upon calling the function. After initializing parameters for the Procedure, the Oracle Procedure will then bypass the SELECT statement. If you break down what is actually saved by doing this operation, a programmer can alleviate doing another SELECT statement, just by doing a conditional check within the Oracle Procedure. Upon the first time through the Oracle Procedure, the SELECT statement can read values into hidden fields on page 0 of SQL*FORMS Screen Painter, thus storing the data from the database within the form. (I recommend this for small to medium applications.) This greatly increases the application performance. For larger SQL*FORMS applications, use a separate Oracle Procedure to read in values from the database into hidden fields.

****

An Example...Study Hard Grasshopper

```
DEFINE PROCEDURE

NAME = CONDITIONAL_TEST
DEFINITION = <<<
PROCEDURE CONDITIONAL_TEST IS
 HOLD_FIELD1   CONSTANT NUMBER := :BLOCK.FIELD1;
 HOLD_FIELD2   CONSTANT NUMBER := (NVL(:FORM8615_92.E73650,0 ) +
                                   NVL(:FORM8615_92.E74150,0 ));
 HOLD_FIELD3   CONSTANT NUMBER := (NVL(:FORM8615_92.E73100,0 ) +
                                   NVL(:FORM8615_92.E73200,0 ) +
                                   NVL(:FORM8615_92.E73300,0 ));
   TEST_1      EXCEPTION;
 BEGIN
/*****************************************************************/
IF :CONTROL.PF2 = 0 THEN
  IF ((HOLD_FIELD1 > (NVL(:BLOCK.FIELD1,0) + 10)) OR
     (HOLD_FIELD1 < (NVL(:BLOCK.FIELD1,0) - 10))) THEN
     :BLOCK.FIELD1 := 0.0;          /* Arbitary field */
     CONDITIONAL_TEST;              /* Recursive Call */
  END IF;
/*****************************************************************/
  IF :PAGE_0.FIELD_1 < HOLDER THEN
     :BLOCK.FIELD2 := 0.0;
     CONDITIONAL_TEST;              /* Recursive Call */
  END IF;
```

```
/*****************************************************************/
  IF ((HVALUE > (NVL(:BLOCK.FIELD3,0) + 10)) OR
     (HVALUE < (NVL(:BLOCK.FIELD3,0) - 10))) THEN
     RAISE TEST_1;
  END IF;
/** End of Consistency Test ***********************************/
/** Global Parameters ****** When the PF2 key is pressed *******/
ELSIF :CONTROL.PF2 = 1 THEN
    IF :BLOCK.FIELD1 IS NOT NULL THEN
      :CONTROL.PF2 := 0;    /* Reset the Oracle Procedure */
      MESSAGE('Amount already exists for line 6.');
      NEXT_FIELD;
      RAISE FORM_TRIGGER_FAILURE;
    ELSIF :BLOCK.FIELD1 IS NULL THEN
      IF HOLD_FIELD1  < 5370 THEN
        HOLD_FIELD1 := ROUND(((:BLOCK.FIELD2 / .15 ) +
                  NVL(:BLOCK.FIELD3,0)));
      ELSIF HOLD_FIELD2 >= 2000 AND HOLD_FIELD2 <= 10000 THEN
           :BLOCK.FIELD3 :=
                      ROUND((((HOLD_FIELD2 - 200 ) /
                      .30) + 35800
                      + NVL(:FORM8615_92.E73650,0)));
      ELSIF HOLD_FIELD3 > 20000 THEN
           :BLOCK.FIELD4 :=
                      ROUND(((HOLD_FIELD3 - 20000 ) /
                      .31)+ 86500
                      + NVL(:BLOCK.FIELD2,0));
      END IF;
     :CONTROL.PF2 := 0;
     CONSISTENCY_TEST;  /* Initialize variable and call */
    END IF;            /* recursively call the procedure */
ELSIF :CONTROL.PF2 = 2 THEN
    :CONTROL.PF2 := 0;
    GO_FIELD('COMPUTATION.HOLD_FIELD');
    RAISE FORM_TRIGGER_FAILURE;
END IF;
/*****************************************************************
COMMIT_FORM;
EXIT_FORM(NO_VALIDATE);
EXCEPTION
  WHEN TEST_1 THEN
    MESSAGE('Line 5 does not equal the smaller of (Ln 1 '||
         'minus Ln 2) or Line 4.');
    GO_FIELD('BLOCK.FIELD1');
    RAISE FORM_TRIGGER_FAILURE;
 END;
 >>>

 ENDDEFINE PROCEDURE
```

```
/*
 *   Main body of the Form.
 */

DEFINE TRIGGER

 NAME = KEY-STARTUP
 TRIGGER_TYPE = V3
 TEXT = <<<
 BEGIN
  /*
   * 1st STEP
   *
   * Set the Global parameter and initialize
   * the parameter to '0';
   */
   BEGIN
     SELECT NVL(TABLE.FIELD_1,0), NVL(TABLE.FIELD_2,0)
       INTO :PAGE_0.FIELD_1, :PAGE_0.FIELD_2
       FROM TABLE
      WHERE TABLE.NUMBER = :BLOCK.NUMBER;
     EXCEPTION
       WHEN NO_DATA_FOUND THEN
         MESSAGE('No data found for this record.');
         RAISE FORM_TRIGGER_FAILURE;
     END;
  :CONTROL.PF2 := 0;
  END;
  >>>

 ENDDEFINE TRIGGER

 DEFINE TRIGGER

  NAME = KEY-NXTFLD
  TRIGGER_TYPE = V3
  TEXT = <<<
  BEGIN
   /*
    *   The conditional statements are tested at the end of the
    *   screen. This is to allow for the user to rapidly input
    *   data onto the screen, and then check it against the
    *   other fields of the screen.
    */
   IF :SYSTEM.TRIGGER_FIELD = 'BLOCK.FIELD4' THEN
     CONDITIONAL_TEST;
   ELSE
     NEXT_FIELD;
   END IF;
 EXCEPTION
   WHEN OTHERS THEN
      RAISE FORM_TRIGGER_FAILURE;
 END;
>>>

ENDDEFINE TRIGGER

DEFINE BLOCK

 NAME = BLOCK
 DESCRIPTION = BLOCK
 TABLE = BLOCK
 ROWS_DISPLAYED = 1
 BASE_LINE = 1
 LINES_PER_ROW = 0
 ARRAY_SIZE = 0

 DEFINE FIELD

  NAME = FIELD1

   DEFINE TRIGGER

    NAME = KEY-CQUERY
     TRIGGER_TYPE = V3
     TEXT = <<<
       :CONTROL.PF2 := 1;
       CONDITIONAL_TEST;
   >>>

   ENDDEFINE TRIGGER

   DEFINE TRIGGER

     NAME = PRE-FIELD
     TRIGGER_TYPE = V3
     TEXT = <<<
     BEGIN
       MESSAGE('Press PF2 to calculate parents taxable ' ||
              'income.');
     END;
     >>>

    ENDDEFINE TRIGGER

  ENDDEFINE FIELD
```

```
DEFINE FIELD

 NAME = FIELD2

  DEFINE TRIGGER

   NAME = KEY-CQUERY
   TRIGGER_TYPE = V3
   TEXT = <<<
   :CONTROL.PF2 := 2;
   CONDITIONAL_TEST;
   >>>

  ENDDEFINE TRIGGER

  DEFINE TRIGGER

   NAME = PRE-FIELD
   TRIGGER_TYPE = V3
   TEXT = <<<
   BEGIN
    MESSAGE('Press PF2 to calculate tax, compare with '||
           'the amount on FIELD 4.');
   END;
   >>>

  ENDDEFINE TRIGGER

 ENDDEFINE FIELD

ENDDEFINE BLOCK
```

****

■ Advantages (Slicing off the Fat)

The primary advantage (to repeat what I said before) is that the control of SQL*FORMS Application is placed into one main procedure. As a matter of protocol, I usually keep one conditional Oracle Procedure routine per page. This testing methodology allows a great deal of flexibility for screen design and for naming efforts, such as CTEST_PG1, CTEST_PG2, and ......

Another advantage, and probably the most important, is the locality of the conditional statements within a central data structure. Although we are not supposed to do it, when a programmer uses an editor, such as vi, finding the conditional statement is rather easy to implement changes, by doing a global search on a string. If the programmer designs applications to be tested at the field level, the programmer has to go to each field and change the necessary code for the application. However, by using recursion, the cursor can be executing the Oracle Procedure, and the cursor can be instructed to go through the Oracle Procedure to check another part of the test. All this can be done be setting parameters or GLOBAL's to different settings within SQL*FORMS 3.0.

■ Disadvantages (Chewing Off the Fat)

There are probably many programmers who can offer several rebuttals to this methodology. But as I told you in the beginning of this paper, I was going against this paradigm.

One of the drawbacks is knowing enough is enough. Once you start programming recursive procedures, a programmer can get carried away with oneself. The trick to do this is to break down the operations of the procedure. Find out what is really necessary in the form to be programmed, and use generic coding to perform the operations. Then, set out to accomplish the task. This sounds easy; however, it is more difficult in application. The main disadvantage to this type of programming methodology is the size of the conditional Oracle Procedure. You have to be very careful about how large the procedure will get, and if your procedure is too long, then it would be difficult to debug and test the program, not to mention the time wasted sitting around and waiting for the SQL*FORMS to generate.

So you have to abstractly think about the program development well in advance, before you start to sit down and pound away at the program. You have to ask yourself, whether recursion can be used, and if so, will the SQL*FORMS 3.0. applications be easier to program, debug, and maintain.

When using the Oracle GLOBALS, it is important to remember to add a POST-FORM trigger to contain the ERASE function, to clear out the GLOBALS that were set in the KEY-STARTUP. On more than one instance, when I was going from form-to-form, the behavior of my form was demonstrating a strange phenomena. The GLOBAL variable was initialized earlier in the program, and I had used the variable without initialization in the existing SQL*FORMS KEY-STARTUP application, thus causing the conditional statement to always be evaluated to a certain condition. Using the POST-FORM trigger with the ERASE function can evaluate these minor problems.

■ Summary

Recursive PL/SQL is not that hard to understand and to use in SQL*FORMS 3.0. However, the key is to know when to use the recursive technique and to control the recursive behavior, and to remember the Recursive Creed and Oath:

I swear to:
1: Know when to stop.
2: Decide how to take the (first) step.
3: Break the Procedure down into separate steps and combine them together.
4. Use Generic Functions.

Although Recursive PL/SQL is not for everybody's shop, it does provide an interesting way to centralize Oracle Procedures and to consolidate repetitive coding.

■ Conclusion

So what if you don't read Oracle Reference Manuals on your leisure time. No Big Deal. Just remember that PL/SQL can be used more than just the plain old envelope procedural language. There are plenty of interesting features that you might want to investigate.

Don't be afraid to push the outer limits of PL/SQL. You may be surprised at what you can come up with.

Platform:

Hardware: Sequent s/2000
Software: Oracle RDBMS 6.0.34.3.1,
Oracle SQL*FORMS 3.0.16.11.01,
Oracle SQL*Plus 3.0.11.1.1,
Oracle PL/SQL 1.0.34.2.1. ■