Expanding the SQL Horizons: PL/SQL User-Defined Functions in the Real World

Michael Rosenblum, Dulcian, Inc.

The good old database half-wisdom/half-joke, often attributed to Tom Kyte, is still as valid as ever:

1. If you have a thing to do in the database, do it in SQL.
2. If you cannot do it in SQL, do it in PL/SQL.
3. If you cannot do it in either SQL or PL/SQL, do it in Java.
4. If you cannot do it in Java, do it in C.
5. If you cannot do it in C, are you sure that it needs to be done?

The reality is a bit more complicated because there are many dimensions involved when comparing different approaches such as performance, maintainability, available expertise, etc. The problem becomes even more challenging when the task crosses the boundaries of multiple languages. Overall, the rule of thumb states that working in the same environment is much better than jumping between multiple ones (of course, there are exceptions). You can pay a high price for unnecessary context switches, especially in terms of CPU costs. The key qualifier in the last sentence is "unnecessary." You should not try to stay within the same language environment simply for the sake of environmental consistency. Different tools are better suited for different purposes, but the goal of decreasing context switches is a sound one. Knowing how to appropriately select between SQL-based and PL/SQL-based solutions is one of the most important issues in Oracle database development. For the purposes of this paper, two less common examples of using PL/SQL in actual systems development were selected. The first illustrates the notion of performance tuning by going outside of the regular solution patterns, while the second one describes a case of extending standard SQL functionality. Both of these examples demonstrate the depth of PL/SQL language. It can do significantly more than you might imagine.

Making Life Simpler by Switching to PL/SQL

It is very common for SQL statements to grow more complex over time. If the system has been in production for a while, the once good decision to use SQL may eventually become less and less desirable. This is especially true when volumes change. Solutions that worked well in a smaller scope very often only scale so far, and can eventually lead to catastrophe. The following real world example demonstrates how switching from SQL to PL/SQL saved the day when the requirements went beyond the scope of the original implementation.

One of the most often encountered problems related to fluctuating between SQL and PL/SQL is the never-ending quest to find an efficient implementation of the "main search" functionality. About 90% of contemporary applications include some type of main screen with a number (usually a lot) of different filtering criteria that presents a grid with matching results. At first glance, this would seem to be a straightforward SQL implementation, especially if the search is limited to one table. But eventually, you will need to search using data from a group of sources, using multi-select, and/or a proximity search (LIKE, SOUNDEX, etc.). Gradually, the original simple query becomes so convoluted that any time you are asked to add an extra filter, you automatically budget at least a week of work time because you are not sure of the potential impact on other possible permutations.

Years ago, the author developed his own way of building a main search engine using Dynamic SQL. Obviously, Dynamic SQL allows you to build and execute SQL and PL/SQL on the fly. You can also use object collections in conjunction with Dynamic SQL. Combining these features means that the results of the search should be represented as object collections and should be built to match the specified search criteria. Instead of building a single generic SQL statement that can survive all possible search permutations, you should build customized SQL statements for each case.

The following is a basic example of such an implementation. Assume that there are requirements to filter employees by employee name, employee ID, and employee location (two filters are from the EMP table, while the last one is from the DEPT table). This requires an output structure that will represent your search results as shown here:

```
CREATE TYPE emp_search_ot AS OBJECT (empno_nr NUMBER,
                                     empno_dsp VARCHAR2(256),
                                     comp_nr  NUMBER);
CREATE TYPE emp_search_nt IS TABLE OF emp_search_ot;
```

It is critical to make EMP_SEARCH_NT a SQL type using a CREATE TYPE statement and not a part of any package. This is necessary because SQL object collection types can be converted into a regular SQL set using the built-in function TABLE (each object attribute becomes a column). Starting with Oracle 12c, you can use also package-defined collections for TABLE functions within PL/SQL program units. Even then, you will not be able to run direct SQL statements or make the TABLE function a part of a view if you don't have SQL type.
Now you can build a function that will return an object collection. Note that, in addition to filters, there is a default limit included. This should become a habit for anyone working with collections. You do not want to bring millions of rows into memory just because a user didn't specify any conditions.

```
CREATE FUNCTION f_search_tt
 (i_empno NUMBER:=NULL, i_ename_tx VARCHAR2:=NULL, i_loc_tx VARCHAR2:=null,
  i_limit_nr NUMBER:=50)
RETURN emp_search_nt IS
  v_out_tt emp_search_nt:=emp_search_nt(); -- output structure
  v_from_tx  VARCHAR2(32767):='emp';
  v_where_tx VARCHAR2(32767):='rownum<=v_limit_nr';
  v_plsql_tx VARCHAR2(32767);
BEGIN
  IF i_empno IS NOT NULL THEN
   v_where_tx:=v_where_tx||chr(10)||'and emp.empno=v_empno_nr';
  END if;
  IF i_ename_tx IS NOT NULL THEN
   v_where_tx:=v_where_tx||chr(10)||'and emp.ename like ''%''||v_ename_tx||''%''';
  END IF;
  IF i_loc_tx IS NOT NULL THEN
    v_from_tx:=v_from_tx||chr(10)||'join dept on (emp.deptno=dept.deptno)';
    v_where_tx:=v_where_tx||chr(10)||'and dept.loc=v_loc_tx';
  END IF;
  v_plsql_tx:=
    'declare '||chr(10)||
    'v_limit_nr number:=:1;'||chr(10)||
    'v_empno_nr number:=:2;'||chr(10)||
    'v_ename_tx varchar2(256):=:3;'||chr(10)||
    'v_loc_tx   varchar2(256):=:4;'||chr(10)||
    'begin '||chr(10)||
    'select emp_search_ot('||
           'emp.empno,emp.ename||''(''||emp.job||'')'','||
           'emp.sal+nvl(emp.comm,0))'||chr(10)||
    'bulk collect into :5'||chr(10)||
    'from '||v_from_tx||chr(10)||
    'where '||v_where_tx||';'||chr(10)||
    'end;';
    $IF $$MishaDebug $THEN
        dbms_output.put_line('<<Script that was executed>>'||chr(10)||v_plsql_tx);
    $END IF;
  EXECUTE IMMEDIATE v_plsql_tx USING
```

```
    IN i_limit_nr, IN i_empno, IN i_ename_tx, IN i_loc_tx,
    OUT v_out_tt;
  RETURN v_out_tt;
END;
```

The function above has a number of points requiring explanation:

- Table EMP is always used, while table DEPT is joined only when location is specified. ANSI SQL comes in very handy here because it allows for clearly split filtering and joining.
- You do not want to build permutations of EXECUTE IMMEDIATE to match different combinations of bind parameters that could be referenced. For this reason, it is much easier to generate anonymous blocks to contain all parameters and pass real values as defaults. Using this approach, you still have the full power of bind variables, but you do not have to worry about their order or number.
- The EMP_SEARCH_OT constructor must be included in the query because the output result is an object collection, and not a record.
- While building all portions of the queries, it is critical to keep the attributes fully qualified (TABLE.COLUMN).
- If you are using Dynamic SQL, always output it before execution. Doing this saves a lot of debugging time.

The most important idea behind the function above is to achieve the highest level of flexibility without losing performance or readability. Use the following code to a search and see what happens:

```
SQL> SELECT * FROM TABLE(f_search_tt(NULL,'A','CHICAGO',2));
  EMPNO_NR EMPNO_DSP               COMP_NR
---------- -------------------- ----------
      7499 ALLEN(SALESMAN)            1900
      7521 WARD(SALESMAN)             1750
<<Script that was executed>>
Declare
v_limit_nr number:=:1;
v_empno_nr number:=:2;
v_ename_tx varchar2(256):=:3;
v_loc_tx   varchar2(256):=:4;
begin
select
emp_search_ot(emp.empno,emp.ename||'('||emp.job||')',emp.sal+nvl(emp.comm,0))
bulk collect into :5
from emp
join dept on (emp.deptno=dept.deptno)
where rownum<=v_limit_nr
and emp.ename like '%'||v_ename_tx||'%'
and dept.loc=v_loc_tx;
end;
```

The join was built on the fly and used only when it was actually needed. This example illustrates the notion that the best tuning approach is to do nothing unless there is no other way. It also points out that constructing SQL statements dynamically can significantly shift the focus of all development efforts. Instead of trying to find a universal solution, you can divide this task into a set of smaller tasks and solve them one at a time. For example, depending upon the columns and tables involved, you can also add hints, change AND-conditions to OR-conditions, etc. As mentioned previously, you pay the price of overhead, but gain a lot of flexibility, which can often be more important.

Using PL/SQL to Fill Functionality Gaps

Many developers were relieved to learn that Oracle eventually created an official built-in function LISTAGG that provides a simple way of putting together a group of columns into a single text string. As long as you are working with a limited set of rows, it works like a charm:

```
SQL> SELECT deptno, listagg(ename,',') WITHIN GROUP(ORDER BY ename) list_tx
  2  FROM emp
  3  GROUP by deptno;
    DEPTNO LIST_TX
---------- ------------------------------------
        10 CLARK,KING,MILLER
        20 ADAMS,FORD,JONES,SCOTT,SMITH
        30 ALLEN,BLAKE,JAMES,MARTIN,TURNER,WARD
```

But there is a minor problem. LISTAGG is a regular SQL function, which means that it cannot return more than 4000 characters (even in Oracle 12c which can support columns up to VARCHAR2(32767)!). If the concatenated result exceeds this limit, it crashes and displays the error: "ORA-01489: result of string concatenation is too long".
The following shows a process for creating your own LISTAGG function that returns a CLOB. Currently, Oracle does not allow aggregate functions to have multiple parameters, but LISTAGG needs two: a value to aggregate and a separator. To overcome this restriction, you can create an object type with two attributes so you have a single parameter of composite nature:

```
CREATE OR REPLACE TYPE listAggParam_ot IS OBJECT
    (value_tx VARCHAR2(4000),
     separator_tx VARCHAR2(10))
```

The next step is to create a special object type required by the Oracle Extensibility Framework, or to be precise by its ODCIAggregate interface routines:

```
CREATE OR REPLACE TYPE ListAggCLImpl AS OBJECT (
  v_out_cl CLOB,
  v_defaultSeparator_tx VARCHAR2(10),
  STATIC FUNCTION ODCIAggregateInitialize(sctx IN OUT ListAggCLImpl)
    RETURN NUMBER,
  MEMBER FUNCTION ODCIAggregateIterate(self IN OUT ListAggCLImpl,
    value_ot IN listAggParam_ot) RETURN NUMBER,
  MEMBER FUNCTION ODCIAggregateTerminate(self IN ListAggCLImpl,
    returnValue OUT CLOB, flags IN NUMBER) RETURN NUMBER,
  MEMBER FUNCTION ODCIAggregateMerge(self IN OUT ListAggCLImpl,
    ctx2 IN ListAggCLImpl) RETURN NUMBER
  )
```

The structure of time definition is the following:

- a number of type attributes to serve as intermediate data storage
- method ODCIAggregateInitialize called once per group to initialize all required settings
- method ODCIAggregateIterate called once for every processed value. The second parameter's datatype must match the datatype of the input you are planning to process.
- method ODCIAggregateTerminate is called once at the end of each group. The second parameter's datatype should match the expected output of your aggregate function.

- method ODCIAggregateMerge is called in case your aggregate function is running in parallel. It is used to put together the results of different threads.

The following code is used to create the body of this type:

```
CREATE OR REPLACE TYPE BODY ListAggCLImpl is
STATIC FUNCTION ODCIAggregateInitialize(sctx IN OUT ListAggCLImpl)
RETURN NUMBER IS
BEGIN
  sctx := ListAggCLImpl(null,','); -- default constructor
  RETURN ODCIConst.Success;
END;
MEMBER FUNCTION ODCIAggregateIterate
   (self IN OUT ListAggCLImpl, value_ot IN listAggParam_ot)
RETURN NUMBER IS
BEGIN
    IF self.v_out_cl IS NULL THEN
        self.v_defaultSeparator_tx:=value_ot.separator_tx;
        dbms_lob.createtemporary(self.v_out_cl,true,dbms_lob.call);
        dbms_lob.writeappend
          (self.v_out_cl,length(value_ot.value_tx),value_ot.value_tx);
    ELSE
        dbms_lob.writeappend(self.v_out_cl,
                    length(value_ot.separator_tx||value_ot.value_tx),
                    value_ot.separator_tx||value_ot.value_tx);
    END IF;
    RETURN ODCIConst.Success;
END;
MEMBER FUNCTION ODCIAggregateTerminate
   (self IN ListAggCLImpl, returnValue OUT CLOB, flags IN NUMBER)
RETURN NUMBER IS
BEGIN
  returnValue := self.v_out_cl;
  RETURN ODCIConst.Success;
END;
MEMBER FUNCTION ODCIAggregateMerge
   (self IN OUT ListAggCLImpl, ctx2 IN ListAggCLImpl)
RETURN NUMBER IS
BEGIN
  IF ctx2.v_out_cl IS NOT NULL THEN
    IF self.v_out_cl IS NULL THEN
        self.v_out_cl:=ctx2.v_out_cl;
    ELSE
        dbms_lob.writeappend(self.v_out_cl,
                    length(self.v_defaultSeparator_tx),
                    self.v_defaultSeparator_tx);
        dbms_lob.append(self.v_out_cl,ctx2.v_out_cl);
    END IF;
  END IF;
  RETURN ODCIConst.Success;
END;
END;
```

As you can see, the code is unusual and requires some explanations:

- Method ODCIAggregateInitialize has a default constructor that specifies the initial values of two attributes of ListAggCLImpl type.
- Method ODCIAggregateIterate via DBMS_LOB APIs adds new values to the existing temporary storage V_OUT_CL
- Method ODCIAggregateTerminate returns the final value of V_OUT_CL as a formal result.
- Method ODCIAggregateMerge is used in case there are parallel executions merging two different V_OUT_CL values into a single output.

The final step is to define the function itself:

```
CREATE OR REPLACE FUNCTION ListAggCL (value_ot listAggParam_ot)
RETURN CLOB
PARALLEL_ENABLE
AGGREGATE USING ListAggCLImpl;
```

The way it is used is no different from any other aggregate function. It also can be used as an analytical function as shown in the following examples:

```
SQL> SELECT deptno, ListAggCL(listAggParam_ot(ename,',')) list_cl
  2  FROM emp
  3  GROUP BY deptno;
    DEPTNO LIST_cl
---------- ------------------------------
        10 CLARK,MILLER,KING
        20 SMITH,FORD,ADAMS,SCOTT,JONES
        30 ALLEN,JAMES,TURNER,BLAKE,MARTIN,WARD

SQL> SELECT empno, ename,
  2     ListAggCL(listAggParam_ot(ename,','))
  3     OVER(PARTITION BY deptno ORDER BY ename
  4     ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) list_cl
  5  FROM emp
  6  WHERE job = 'CLERK';
    EMPNO ENAME      LIST_CL
---------- ---------- --------------------------------------------------
      7934 MILLER     MILLER
      7876 ADAMS      ADAMS,SMITH
      7369 SMITH      ADAMS,SMITH
      7900 JAMES      JAMES
```

The first example above illustrates the usage of our ListAggCL as a pure aggregate function. All employees of each department were connected into a comma-separated list. The result is close to what you would expect from a regular built-in, as long as you are not looking for the sorted list. The real LISTAGG has a special WITHIN GROUP clause that currently cannot be directly replicated using ODCI interfaces. Of course, it is possible to make ListAggCL sort values by buffering them into a temporary collection and spinning that collection in ODCIAggregateTerminate. However, for the problems that the author was trying to solve, that sorting was not critical. Note that in 2004, there were good discussions on “AskTom” (http://tinyurl.com/AskTom-StrAgg) about different variations of STRAGG and many of these ideas are still applicable!
The second example shows how the same function could be used as an analytic. It prints out all clerks in the EMP table together with the comma-separated list of all clerks who work in the same department as in the processed record. This time, in the OVER clause, you can specify ordering and sort values in the list before putting it together. By default, analytic functions with the ORDER BY clause use a floating window BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW (the short form of it is RANGE UNBOUNDED PRECEDING), while you usually need the whole group to be evaluated. For this reason, the second part of the range was replaced with UNBOUNDED FOLLOWING.

User-defined aggregate functions are very handy as long as you understand what is going on. The author has seen a lot of interesting implementations utilizing the Extensible Framework. There are a significant number of cases when standard SUM or AVG functions were extended to return 0 instead of NULLs for the empty groups. There was even a system where people built their own MULT function to multiply all values in the set. This is not a bad idea to have as a standard built-in!

Managing User-Defined Functions

Historically, the most valuable role of PL/SQL was to provide user-defined functions to do things that could not be done in SQL (or could be done in SQL, but very inefficiently). Unfortunately, opening SQL to user-defined functions also opened some new performance-related danger areas. Complete coverage of potential pitfalls in all of the listed areas is impossible within a single paper, but it will introduce you to the most important ones.

Calling Functions within SQL

For too many database developers, the question of how many times their user-defined PL/SQL function is being called while processing the SQL statement is a mystery. Very often, a significant cause of performance problems is rooted in this area because user-defined functions are being called too many times. Each of those calls not only incurs a SQL-to-PL/SQL-and-back context switch, but also adds to the total cost when functions are called unnecessarily. Of course, there are special caching mechanisms. But before going to that level of optimization, it is very important to understand what happens under normal circumstances. For the purposes of performance tuning, it is important to remember the order of SQL statement execution:

1. JOIN
2. WHERE
3. GROUP BY
4. SELECT (including analytics functions)
5. HAVING
6. ORDER BY

The reason to consider the order is that you can eliminate some calls by applying conditions earlier in the process. It is very important to keep the list above in your head any time you are trying to analyze SQL statement internal logic.

Single Table Problems

Before discussing multi-table joins, you should first understand single-table activities that will make the whole analysis process much simpler. To illustrate different cases, create the following testing environment with a package to store a function counter and a testing procedure:

```
CREATE OR REPLACE PACKAGE counter_pkg IS
    v_nr NUMBER:=0;
    PROCEDURE p_check;
END;
CREATE OR REPLACE PACKAGE BODY counter_pkg IS
    PROCEDURE p_check is
    BEGIN
        dbms_output.put_line('Fired:'||counter_pkg.v_nr);
        counter_pkg.v_nr:=0;
    END;
END;
CREATE OR REPLACE FUNCTION f_change_nr (i_nr NUMBER) RETURN NUMBER IS
```

```
BEGIN
    counter_pkg.v_nr:=counter_pkg.v_nr+1;
    return return i_nr*(-1);;
END;
```

It is very difficult to create an exhaustive test of all possible permutations. For the purposes of this paper, the most important or counter-intuitive ones were selected.
Keep in mind that the same function put in different places within a SQL statement may not produce the same results. By default, in Oracle all of those calls are different as shown here:

```
SQL> SELECT empno, ename, f_change_nr(empno) change_nr
  2  FROM emp
  3  WHERE f_change_nr(empno) IS NOT NULL
  4  AND deptno = 20;
...
5 rows selected.

SQL> exec counter_pkg.p_check;
Fired:10
```

The example above demonstrates the following:

- The CBO tries to order predicates to decrease the total cost of the query. As a result, DEPTNO=20 was applied before F_CHANGE_NR(EMPNO) IS NOT NULL. This means that the second condition was checked for only 5 rows. It also means that it is very important to keep table statistics up to date, have proper indexes, and constraints etc. to help the CBO make the right ordering choices.
- The same functions in SELECT and WHERE clauses are being fired independently and cannot be reused (unless the results are cached in some way. That's why the total number of calls equals 10.

Overall, every time you fire a function anywhere, it requires a separate call. Even worse, sometimes might need multiple calls, depending upon how Oracle rewrites your code:

```
SQL> SELECT empno, ename, f_change_nr(empno) change_nr
  2  FROM emp
  3  WHERE deptno = 20
  4  ORDER BY 3;
5 rows selected.
...

SQL> exec counter_pkg.p_check;
Fired:5

SQL> SELECT empno, change_nr
  2  FROM (
  3    SELECT empno, ename, f_change_nr(empno) change
  4    FROM emp
  5    WHERE deptno = 20
  6    ORDER BY 3
  7    );
...
5 rows selected.
SQL> exec counter_pkg.p_check;
Fired:10
```

The two examples above differ only in that, in the second case, the main query was wrapped as an in-line view. Surprisingly, the second time the function F_CHANGE_NR was fired two times more, namely 10 instead of 5. If you generate 10053 trace (CBO), you will find that the following query was executed:

```
Final query after transformations:******* UNPARSED QUERY IS *******
SELECT "EMP"."EMPNO" "EMPNO","EMP"."ENAME" "ENAME",
       "SCOTT"."F_CHANGE_NR"("EMP"."EMPNO") "CHANGE_NR"
FROM "SCOTT"."EMP" "EMP"
WHERE "EMP"."DEPTNO"=20
ORDER BY "SCOTT"."F_CHANGE_NR"("EMP"."EMPNO")
```

The query seems suspicious because of two separate calls to F_CHANGE_NR (instead of referencing column position), but if you run that query directly, you will still get 5 executions. Something does not add up. After more digging into the 10053 trace, it is clear that the catch can be found elsewhere. Oracle has an internal CBO optimization feature called "ORDER BY elimination" (OBYE) that cuts unnecessary work from the ORDER BY. Unfortunately, it happens before the query transformation, so when the CBO evaluates the original call, it does not find anything to optimize in the root SELECT statement:

```
Order-by elimination (OBYE)
***************************
OBYE:     OBYE performed.
OBYE:     OBYE bypassed: no order by to eliminate.
```

This is why after the transformation, the ORDER BY clause suddenly appears. It is not eliminated and you end up with double the number of function calls. Interestingly enough, adding a /*+ NO_MERGE */ hint to the in-line view makes the double-fire problem disappear. It tells the CBO to keep in-line views instead of merging them with the main query. Although, remember that a hint is just a suggestion, and not a directive. There are known cases when the CBO transforms predicates and /*+ NO_MERGE */ could be ignored.

```
SQL> SELECT empno, change_nr
  2  FROM (
  3    SELECT /*+ NO_MERGE */empno, ename, f_change_nr(empno) change_nr
  4    FROM emp
  5    WHERE deptno = 20
  6    ORDER BY 3
  7    );
5 rows selected.
...

SQL> exec counter_pkg.p_check;
Fired:5
```

Let's take this example one step further and replace the in-line view with the real view. The chances are very good that the double-fire behavior will persist:

```
SQL> CREATE OR REPLACE VIEW v_emp AS
  2  SELECT empno, ename, f_change_nr(empno) change_nr
  3  FROM emp
  4  WHERE deptno = 20
  5  ORDER BY 3;
View created.

SQL> SELECT empno, change_nr
```

```
  2  FROM v_emp;
...
5 rows selected.

SQL> exec counter_pkg.p_check;
Fired:10
```

This is a real issue! In a lot of production systems, the author has often seen views with ORDER BY clauses referencing view columns created with PL/SQL functions. In turn, these views were used as parts of other more complex views. The result above demonstrates that such a coding style is a very bad idea because it could lead to doubling of the overhead incurred by firing those functions. More importantly, that overhead could just suddenly arise when the execution plan changes because of data growth or other reasons. In general, the ORDER BY clause should be applied at the highest level whenever possible. The lesson to be learned here is that even with a one-table query, PL/SQL functions can impact performance significantly, especially if used in multiple places. Be careful!

Multi-Table Problems

When PL/SQL functions are being called in multi-table joins, it is very important to keep in mind that you are operating on the merged sets. Here is a basic example:

```
SQL>   SELECT empno, f_change_nr(empno) change_nr, dname
  2    FROM emp,
  3         dept
  4    WHERE emp.deptno(+) = dept.deptno;
     EMPNO  CHANGE_NR DNAME
      7782       7782 ACCOUNTING
      ...
      7654       7654 SALES
                      OPERATIONS
15 rows selected.

SQL> exec counter_pkg.p_check;
Fired:15
```

Note the outer join between EMP and DEPT. This causes the query to return 15 rows: 14 from EMP plus one DEPT that does not have any employees. The function F_CHANGE_NR is also fired 15 times because it is being applied after the join. As a result, 1 out of 15 calls is unnecessary. The same 15 executions will occur even if you pass a column from the DEPT table into the function. This leads to even worse overhead because you have only four distinct departments. Anything with more than four calls is a waste of resources (11 extra calls!):

```
SQL>   SELECT empno, f_change_nr(dept.deptno) change_nr, dname
  2    FROM emp,
  3         dept
  4    WHERE emp.deptno(+) = dept.deptno;
      ...
15 rows selected.

SQL> exec counter_pkg.p_check;
Fired:15
```

The last example illustrates the most common mistake of using PL/SQL functions inside of SQL. If developers pass a column from the small table used in the join, they expect the total number of calls to this function to be relatively small. This is a mistake; however, there are special techniques to make Oracle aware that the total number of calls could be decreased. The majority of these techniques deal with caching and are described later in the paper.

Cost Management Using Oracle 12c-Only Features

As of this writing, Oracle 12c is very new. It was released only a couple of months ago. Understandably, this version does not have as much deep analysis available as Oracle 11g, but there are already some interesting features.

PRAGMA UDF

Oracle 11g introduced a special PL/SQL optimization level #3, where PL/SQL program units were "in-lined" into their callers. Starting with Oracle 12c, a similar "in-lining" notion could be applied to user-defined functions that are primarily called in SQL. The change is made by specifying the PRAGMA UDF clause in the function declaration. That clause tells Oracle to compile a PL/SQL function to make its usage by the SQL engine more efficient. On the other hand, it will somewhat slow down the same function in the context of PL/SQL, so do not rush to add that PRAGMA UDF clause everywhere. However, it really speeds up functions in SQL, as demonstrated by the following example.
To be able to see measurable performance changes, you need a table with enough rows. You also need an alternative to the function F_CHANGE_NR that would have PRAGMA UDF clause as shown here:

```
CREATE TABLE test_tab AS
SELECT *
FROM all_objects
WHERE ROWNUM <= 50000;

CREATE OR REPLACE FUNCTION f_change_udf_nr (i_nr NUMBER) RETURN NUMBER
IS
    PRAGMA UDF;
BEGIN
    counter_pkg.v_nr:=counter_pkg.v_nr+1;
    RETURN i_nr+1;
END;
The comparison is very simple – both functions will be executed 50000 times:
SQL> SELECT MAX(f_change_nr(object_id))
  2  FROM TEST_TAB;
MAX(F_CHANGE_NR(OBJECT_ID))
---------------------------
                      51485
Elapsed: 00:00:00.48
SQL> SELECT MAX(f_change_udf_nr(object_id))
  2  FROM TEST_TAB;
MAX(F_CHANGE_UDF_NR(OBJECT_ID))
-------------------------------
                         51485
Elapsed: 00:00:00.06
```

The difference in performance between two functions is definitely impressive: 0.48 seconds vs. 0.06! If you check 10046 trace, the time saving is associated with CPU time, so it is all about context switches. Of course, this is relevant for a very large number of iterations, but if you have a large number of light functions (for example, returning global variables or constants) that are being called thousands of times, the gain could be significant. There is also a counter-example when PRAGMA UDF causes performance degradation:

```
SQL> DECLARE
  2      v_out_nr NUMBER;
  3  BEGIN
  4      FOR i IN 1..1000000 loop
  5          v_out_nr:=f_change_nr(i)+f_change_nr(i+1);
  6      END LOOP;
```

```
  7  END;
  8  /
Elapsed: 00:00:01.39

SQL> DECLARE
  2      v_out_nr NUMBER;
  3  BEGIN
  4      FOR i IN 1..1000000 LOOP
  5          v_out_nr:=f_change_udf_nr(i)+f_change_udf_nr(i+1);
  6      END LOOP;
  7  END;
  8  /
Elapsed: 00:00:01.89
```

This time, including PRAGMA UDF caused the loss of 0.4 seconds for 2 million iterations. Although this is not a lot of extra time, it is important to balance SQL gains with PL/SQL losses if the same function is being used in two contexts. Overall, this feature shows very good potential, but it is new and may require additional testing.

Functions Inside the WITH Clause

The last few versions of the Oracle Database have consistently extended the functionality of the WITH clause. Oracle 12c includes the possibility of adding user-defined functions and procedures directly to the SQL statement, instead of creating them as separate objects:

```
SQL> WITH FUNCTION f_changeWith_nr (i_nr number) RETURN NUMBER IS
  2       BEGIN
  3          RETURN i_nr+1;|
  4       END;
  5  SELECT max(f_changeWith_nr(object_id))
  6  FROM test_tab
  7  /
MAX(F_CHANGEWITH_NR(OBJECT_ID))
-------------------------------
                          51485
Elapsed: 00:00:00.07
```

The goal of this approach is to decrease the number of context switches between SQL and PL/SQL, and it does up to a point. As of the initial release of Oracle 12c, there are some drawbacks:

- Coding fragmentation: The whole reason for using stored procedures is to have a single point of functionality. If you allow developers to create user-defined functions directly inside SQL statements, you may significantly complicate the whole code maintenance process.
- PL/SQL limitations: PL/SQL does not currently support SQL statements having functions in the WITH clause at all. Although the same call wrapped in Dynamic SQL will work just fine, this is a significant inconvenience.
- SQL limitations: If you like to use the WITH-clause with functions anywhere other than the top level query, you need to include a special hint /*+ WITH_PLSQL */ on that top level.
- Optimization limitations: The DETERMINISTIC clause is being ignored for WITH-clause functions. As a result, they may be even cheaper than standalone ones that could be fired much more often.
- Performance: Much to the surprise of a lot of Oracle 12c early adopters, adding the PRAGMA UDF clause to regular functions consistently outruns WITH-clause functions (see example above – 0.06 instead of 0.07), but not by a lot, compared with the original costs, although it does so very consistently.

The ink is still not dry on using functions inside the WITH clause functionality. It is worthwhile knowing that it exists, but unless something changes in later releases, its usability is somewhat questionable, especially when compared to the PRAGMA UDF alternative.

Introduction to Different Caching Techniques

There is a very interesting question about user-defined functions: Why do you need to do something that can be done once and preserved for future use multiple times? The answer to this question is "You don't!" Oracle contains a number of different caching techniques, each with own strengths and weaknesses. It would take hundreds of pages to cover them in detail. For the purposes, of this paper, only a brief introduction is included.

Deterministic Functions

If a user-defined function always does exactly the same thing for the specified input (both in terms of output and in terms of database activities), it can be defined with the special keyword DETERMINISTIC. Technically, this is also an internal caching technique because it lets Oracle reuse already known results for performance optimization purposes. Unfortunately, this clause is often misused, because Oracle does not have a way of checking whether your function is indeed deterministic. Although the following example violates the rules of the DETERMINISTIC clause, because you will be changing the packaged variable, it can serve as a good illustration of the effect of this clause:

```
-- basic function
CREATE OR REPLACE FUNCTION f_change_tx (i_tx VARCHAR2) RETURN VARCHAR2 IS
BEGIN
    counter_pkg.v_nr:=counter_pkg.v_nr+1;
    return lower(i_tx);
END;

-- optimized function
CREATE OR REPLACE FUNCTION f_change_det_tx (i_tx VARCHAR2) RETURN VARCHAR2
DETERMINISTIC
IS
BEGIN
    counter_pkg.v_nr:=counter_pkg.v_nr+1;
    RETURN lower(i_tx);
END;
```

This function will be applied to the column EMP.JOB that contains only five distinct values: PRESIDENT, MANAGER, CLERK, ANALYST, and SALESMAN.

```
SQL> SELECT empno, f_change_tx(job) FROM emp;
...

SQL> exec counter_pkg.p_check;
Fired:14

SQL> SELECT empno, f_change_det_tx(job) FROM emp;
...

SQL> exec counter_pkg.p_check;
Fired:5
```

In this case, the DETERMINISTIC function was called only five times, but it is important to remember that the DETERMINISTIC clause is a hint, and not a directive. Oracle can ignore it for reasons that may or may not be clear. This

unpredictability is sometimes the reason why developers prefer to stay away from this feature, even when the system could benefit from it.
It is important to mention that Oracle preserves the results of the deterministic function in the cache only for the duration of the current fetch operation and not for the duration of the entire query. Therefore, the total number of fetch operations could also impact the benefits realized from using this feature.
To examine the DETERMINISTIC clause in more detail, more data is needed. Table TEST_TAB, created in earlier in this paper as the first 50,000 rows from ALL_OBJECTS, is a good test set, especially if a few more columns are added. A special type STRINGLIST_TT will be required to obtain relevant results:

```
ALTER TABLE test_tab ADD
    (obj3_tx VARCHAR2(3),
     obj1_tx VARCHAR2(1));

UPDATE test_tab SET
    obj3_tx  = UPPER(SUBSTR(object_name,-3)), -- 3442 distinct values
    obj1_tx  = UPPER(SUBSTR(object_name,1,1)); -- 26 distinct values

CREATE TYPE stringList_tt IS TABLE OF VARCHAR2(256);
```

Now, by using the FETCH…BULK COLLECT LIMIT syntax, is it possible to check the impact of the DETERMINISTIC clause on the total execution count. This example also compares the total number of function calls with the number of distinct values in each fetch:

```
SQL> DECLARE
  2      v_obj_tt stringList_tt ;
  3      v_count_nr NUMBER;
  4      CURSOR c_rec is
  5      SELECT f_change_det_tx(obj1_tx) obj_tx
  6      FROM test_tab;
  7  BEGIN
  8      OPEN c_rec;
  9      FOR i IN 1..5 LOOP
 10          FETCH c_rec BULK COLLECT INTO v_obj_tt LIMIT 100;
 11          SELECT COUNT(DISTINCT column_value)
 12          INTO v_count_nr
 13          FROM TABLE(CAST (v_obj_tt AS stringList_tt));
 14          counter_pkg.p_check;
 15          dbms_output.put_line('-real count:'||v_count_nr);
 16      END LOOP;
 17      CLOSE c_REC;
 18  END;
 19  /
Fired:17
-real count:14

Fired:22
-real count:14

Fired:26
-real count:16

Fired:25
-real count:14

Fired:17
```

```
-real count:15
```

The results are mixed. The DETERMINISTIC clause did indeed reduce the total number of function calls from 200 to much lower numbers. However, it still fired more often than the distinct number of values in each fetch operation. The explanation for the last part is clear when you understand exactly how the DETERMINISTIC clause works. For the duration of the call, Oracle creates a hash table that stores the results of your function together with its corresponding IN-parameters, where hash-values of IN-parameters work as keys. By default, this hash table is of limited size (65536 bytes) and may be filled very quickly. As a result, after the hash table is full, extra IN/OUT combinations are ignored. The good news is that the size of this table can be changed. The bad news is that it is an "underscore" parameter, which means that you should not touch it unless told to do so by Oracle Support. For the sake of this discussion, let's multiply it by four and see what happens with the test:

```
SQL> ALTER SESSION SET "_query_execution_cache_max_size"=262144;

SQL> ... rerun the example from above …
Fired:17
-real count:14

Fired:16 [was 22]
-real count:14

Fired:18 [was 26]
-real count:16

Fired:24 [was 25]
-real count:14

Fired:17
-real count:15
```

The total number of function calls dropped significantly, not to the exact match, but enough to be aware of this tuning technique. It is important to note, that even if your IN-parameters are distinct, this does not mean that their Oracle hash-values are also distinct. It was proven that it is possible to get hash collisions in the memory table (the hash table). Such collisions also cause Oracle to ignore the DETERMINISTIC clause Overall, the DETERMINISTIC clause can be very useful, but only if you properly understand the data sets that are being processed.

Scalar Sub-Query Caching

Oracle introduced scalar sub-query caching a long time ago as a part of its internal SQL optimization mechanism. By definition, scalar sub-queries return a single column of a single row (or from the empty rowset), while caching in this context means that Oracle intermittently stores the results of such queries while processing more complex ones. Currently, this built-in feature is less well known than it should be and even less well understood. Unfortunately, there are good reasons for its lack of use and understanding since the feature is somewhat counter-intuitive from a PL/SQL developer's point of view. In order to apply it to user-defined functions, your code must be changed as shown here:

```
SQL> SELECT empno, f_change_tx(job) FROM emp;
...

SQL> exec counter_pkg.p_check;
Fired:14

SQL> SELECT empno, (SELECT f_change_tx(job) FROM dual) FROM emp;
...
```

```
SQL> exec counter_pkg.p_check;
Fired:5
```

Surprisingly enough, wrapping a function call into SELECT…FROM DUAL cuts the total number of calls from 14 (as the number of rows) to 5 (as the number of distinct values). The power of this technique is that it not only reuses existing data, but also drops the total number of SQL-to-PL/SQL context switches by internally managing results produced by user-defined functions.
Scalar sub-query caching looks a lot like the DETERMINISTIC clause: Oracle maintains a special memory-based hash table with cached values. It is even internally driven by the same "_query_execution_cache_max_size". However, there are some differences. First, the scope of the scalar sub-query caching is a query, not a fetch:

```
SQL> DECLARE
  2      v_obj_tt stringList_tt;
  3      v_count_nr NUMBER;
  4      CURSOR c_rec IS
  5      SELECT (SELECT f_change_tx(obj1_tx) FROM DUAL) obj_tx
  6      FROM test_tab;
  7  BEGIN
  8      OPEN c_rec;
  9      FOR i IN 1..5 LOOP
 10          FETCH c_rec BULK COLLECT INTO v_obj_tt LIMIT 100;
 11          SELECT COUNT(DISTINCT column_value)
 12          INTO v_count_nr FROM TABLE(CAST (v_obj_tt as stringList_tt));
 13          counter_pkg.p_check;
 14          dbms_output.put_line('-real count:'||v_count_nr);
 15      END LOOP;
 16      CLOSE c_rec;
 17  END;
 18  /
Fired:17
-real count:14

Fired:16
-real count:14

Fired:10
-real count:16

Fired:18
-real count:14

Fired:5
-real count:15
```

As you can see, in some fetches the total number of function calls is less than the number of distinct values. This happens because Oracle can reuse already calculated values from the previous fetch.
The second difference is a bit obscure. It has to do with what happens after the hash table is full. The DETERMINISTIC clause stops accepting new values, but scalar sub-query caching keeps the hash table plus one extra slot. That extra slot is being overwritten each time a new value comes in, but until then, it is preserved by Oracle. This means that if your dataset is ordered, scalar sub-query caching could benefit you even if you have a large number of distinct values in every fetch:

```
SQL> DECLARE
  2      v_obj_tt stringList_tt;
```

```
  3      v_count_nr NUMBER;
  4      CURSOR c_rec IS
  5      SELECT (SELECT f_change_tx(obj3_tx) FROM dual) obj_tx
  6      FROM (SELECT /*+ NO_MERGE */ * FROM test_tab ORDER BY obj3_tx);
  7  BEGIN
  8      OPEN c_rec;
  9      FOR i IN 1..5 LOOP
 10          FETCH c_rec BULK COLLECT INTO v_obj_tt LIMIT 1000;
 11          SELECT COUNT(DISTINCT column_value)
 12          INTO v_count_nr FROM TABLE(CAST (v_obj_tt AS stringList_tt));
 13          counter_pkg.p_check;
 14          dbms_output.put_line('-real count:'||v_count_nr);
 15      END LOOP;
 16      CLOSE c_rec;
 17  END;
 18  /
Fired:160
-real count:160
Fired:268
-real count:268
Fired:56
-real count:57
Fired:62
-real count:63
Fired:22
-real count:23
```

Since the result set is ordered, for every fetch, the total number of function calls would be equal to or one less than the number of distinct values in the set (one less could happen if the same value spawns multiple fetches).
In general, scalar sub-query caching is a very interesting technique that developers should know about. Its greatest benefit is that it can drastically decrease the number of context switches between SQL and PL/SQL and does not require any changes to underlying functions, only adjustments to SQL queries.

PL/SQL Function Result Cache

The query-level caching techniques shown above are very powerful, but in real life, the same PL/SQL functions could be called multiple times in the same session from different queries. Taking it one step farther, your function could return the same results for the same IN for any session. Overall, those two issues could be summarized to the need to have results of PL/SQL functions be reused as widely as possible, irrelevant of fetches, calls, and even sessions. Starting with Oracle 11g, this can be resolved using a new feature called the PL/SQL Function Result Cache, which covers all of the cases described.

Result Cache Basics

From the developer's point of view, enabling this feature is very simple, involving just one extra clause. However, from the administrative side, there are many hidden activities as shown later. The result-cache-enabled function looks as follows:

```
CREATE OR REPLACE FUNCTION f_getDept_dsp (i_deptno NUMBER) RETURN VARCHAR2
RESULT_CACHE
IS
    v_out_tx VARCHAR2(256);
BEGIN
    IF i_deptno IS NULL THEN RETURN NULL; END IF;
```

```
    SELECT initcap(dname) INTO v_out_tx
    FROM dept WHERE deptno=i_deptno;
    counter_pkg.v_nr:=counter_pkg.v_nr+1;
    RETURN v_out_tx;
END;
```

Adding the RESULT_CACHE clause seems very simple, but it completely changed the behavior of this simple function:

```
SQL> SELECT empno, f_getDept_dsp(deptno) dept_dsp FROM emp;
     EMPNO  DEPT_DSP
---------- ------------------------
      7369  Research
       ...
14 rows selected.

SQL> exec counter_pkg.p_check;
Fired:3

SQL> SELECT empno, f_getDept_dsp(deptno) dept_dsp FROM emp;
     EMPNO DEPT_DSP
---------- ----------
      7369 Research
       ...
14 rows selected.

SQL> exec counter_pkg.p_check;
Fired:0
```

In this example, the same function was fired in two different queries against the EMP table that references three departments. The first time, the total number of function calls was three (matching query-level caching), but for the second query, there were no function calls. Obviously, the information must have come from somewhere. This time it came from the PL/SQL Function Result Cache.
Contrary to previously described techniques, the PL/SQL Result Cache has a set of fully documented and published information access methods. They consist of dynamic data dictionary views and a special API in the DBMS_RESULT_CACHE. This API gets you the summary while views let you dig in to the details. From a high-level database management overview, the internal information about the result cache is as follows:

```
SQL> exec dbms_result_cache.memory_report;
R e s u l t   C a c h e   M e m o r y   R e p o r t
[Parameters]
Block Size          = 1K bytes
Maximum Cache Size  = 15M bytes (15K blocks)
Maximum Result Size = 768K bytes (768 blocks)
[Memory]
Total Memory = 166200 bytes [0.012% of the Shared Pool]
... Fixed Memory = 5352 bytes [0.000% of the Shared Pool]
... Dynamic Memory = 160848 bytes [0.011% of the Shared Pool]
....... Overhead = 128080 bytes
....... Cache Memory = 32K bytes (32 blocks)
........... Unused Memory = 27 blocks
........... Used Memory = 5 blocks
............... Dependencies = 2 blocks (2 count)
............... Results = 3 blocks
................... PLSQL   = 3 blocks (3 count)
```

```
SQL> SELECT * FROM v$result_cache_statistics;
   ID NAME                           VALUE
----- ------------------------------ ----------
    1 Block Size (Bytes)             1024
    2 Block Count Maximum            15360
    3 Block Count Current            32
    4 Result Size Maximum (Blocks)   768
    5 Create Count Success           3
    6 Create Count Failure           0
    7 Find Count                     25
    8 Invalidation Count             0
    9 Delete Count Invalid           0
   10 Delete Count Valid             0
   11 Hash Chain Length              1
   12 Find Copy Count                25
```

Both the summary and the view show how much memory is being consumed from the limit allocated by your DBA (managed by the "result_cache_max_result" parameter). But the view also shows that for all result cache-enabled functions, there was a total of 25+3, or 28 requests, out of which only 3 were distinct and 25 were reused.
The view also shows that all cache results are valid. This is a very important point to understand. Starting with Oracle 11g Release 2 onward for each occurrence of caching, Oracle gathers the names of all tables that were touched while the function was executed (in Oracle 11g Release 1 you needed to explicitly use the RELIES ON clause). This does not include packages or the session context, but only tables. This means that you should not enable RESULT_CACHE on functions that depend on such session-level resources, because Oracle would not be able to detect its changes. You can see what objects are of interest for the result cache by running the following query:

```
SQL> SELECT id, type, status, name FROM V$RESULT_CACHE_OBJECTS;
   ID TYPE       STATUS    NAME
----- ---------- --------- ------------------------------
    0 Dependency Published SCOTT.F_GETDEPT_DSP
    1 Result     Published "SCOTT"."F_GETDEPT_DSP"::8."F_GETDEPT_DSP"
                           #762ba075453b8b0d#1
    2 Dependency Published SCOTT.DEPT
    3 Result     Published "SCOTT"."F_GETDEPT_DSP"::8."F_GETDEPT_DSP"
                           #762ba075453b8b0d#1
    4 Result     Published "SCOTT"."F_GETDEPT_DSP"::8."F_GETDEPT_DSP"
                           #762ba075453b8b0d#1
```

Using the results of this query, you can clearly see that Oracle is aware of three cached values (type=Result). To maintain the existing cache intact, Oracle needs to monitor two objects (type=Dependency): function SCOTT.F_GETDEPT_DSP and table SCOTT.DEPT. This view does not show you which cached results depend upon which monitored objects, but the following query allows you to answer this question too:

```
SQL> SELECT rco.id,  rco.name,  ao.owner||'.'||ao.object_name object_name
  2  FROM   v$result_cache_objects    rco,
  3         v$result_cache_dependency rcd,
  4         all_objects               ao
  5  WHERE  rco.id = rcd.result_id
  6  AND    rcd.object_no = ao.object_id
  7  order by 1;
   ID NAME                                        OBJECT_NAME
----- ----------------------------------------  -----------------
    1 "SCOTT"."F_GETDEPT_DSP"::8."F_ GETDEPT_DSP"  SCOTT.DEPT
```

```
    1 "SCOTT"."F_GETDEPT_DSP"::8.″F_ GETDEPT_DSP"  SCOTT.F_GETDEPT_DSP
    3 "SCOTT"."F_GETDEPT_DSP"::8."F_ GETDEPT_DSP"  SCOTT.DEPT
    3 "SCOTT"."F_GETDEPT_DSP"::8.″F_ GETDEPT_DSP"  SCOTT.F_GETDEPT_DSP
    4 "SCOTT"."F_GETDEPT_DSP"::8."F_ GETDEPT_DSP"  SCOTT.DEPT
    4 "SCOTT"."F_GETDEPT_DSP"::8.″F_ GETDEPT_DSP"  SCOTT.F_GETDEPT_DSP
```

The query shows clearly that all three cache entries depend upon both elements. It is important to clarify what this "dependency" really means. In terms of tables, the explanation is very simple. Any INSERT/UPDATE/DELETE or DDL against the table would invalidate the cache, even if there were no change to the data. The function F_GETDEPT_DSP is also in the monitoring list because Oracle needs to handle the case when it is recompiled and the underlying logic has been modified. In this case, Oracle does not check to determine whether or not code changes are significant or even whether they exist at all. If the timestamp is different, the cache is gone.
Keeping data consistency in the result cache is a very challenging task. That is why there are some restrictions to take into account. As of Oracle 12c, in order for the function to be cached, the following conditions should be met:

1. The function is not defined in the anonymous block and is not pipelined.
2. The function does not have OUT or IN/OUT parameters. IN-parameters cannot be LOBs, REF CURSOR, objects, collections or records. Returning values cannot be LOBs, REF CURSORS, or objects. Records and collections are supported if they do not contain previously listed types.
3. References cannot include dictionary tables, temporary tables, sequences, or non-deterministic SQL functions (for example, CURRENT_DATE, SYSDATE, and so on).

Impact of the PL/SQL Result Cache

Keep in mind that the PL/SQL function Result Cache is implemented in such a way that both SQL and PL/SQL code could benefit from it. If you use your functions inside of a SQL statement, there will first be a context switch between SQL and PL/SQL. Only afterwards will Oracle retrieve cached values. On the other hand, the DETERMINISTIC clause and sub-query result cache eliminate the context switch altogether as illustrated with another variation of the F_CHANGE_TX function:

```
CREATE OR REPLACE FUNCTION f_change_cache_tx (i_tx varchar2) RETURN VARCHAR2
RESULT_CACHE IS
BEGIN
    counter_pkg.v_nr:=counter_pkg.v_nr+1;
    RETURN LOWER(i_tx);
END;
```

Now let's check this function against the DETERMINISTIC clause:

```
SQL> exec runstats_pkg.rs_start
SQL> SELECT MAX(f_change_cache_tx(obj1_tx)) FROM test_tab;
MAX(F_CHANGE_CACHE_TX(OBJ1_TX))
--------------------------------------------------------------------
x
SQL> exec counter_pkg.p_check;
Fired:26
SQL> exec runstats_pkg.rs_middle
SQL> SELECT MAX(f_change_det_tx(obj1_tx)) FROM test_tab;
MAX(F_CHANGE_DET_TX(OBJ1_TX))
--------------------------------------------------------------------
x
SQL> exec counter_pkg.p_check;
```

```
Fired:7627
SQL> exec runstats_pkg.rs_stop
Run1 ran in 65 cpu hsecs
Run2 ran in 16 cpu hsecs
run 1 ran in 406.25% of the time
Name                                  Run1        Run2        Diff
STAT...CPU used by this session         64          16         -48
LATCH.Result Cache: RC Latch        50,079           0     -50,079
Run1 latches total versus runs -- difference and pct
Run1          Run2        Diff       Pct
52,388       2,057     -50,331  2,546.82%
```

The results are confusing if you only look at the function count and the clock. Even so, the function was fired 26 times vs. 7627 times. It took four times longer to get exactly the same result. The truth is that to operate with a result cache, Oracle uses a lot of latches, which are known to be CPU-intense. This means that the RESULT_CACHE does not scale to a very high number of simultaneous sessions. Although, if your cached functions are not as light as in the example above, fired less frequently, and contain many more I/O operations, the benefits of the result cache will outweigh the expenditure of extra latches. You can find many details about RESULT_CACHE scalability in the series of blog posts written by Alex Fatkulin (http://afatkulin.blogspot.com).
Overall, PL/SQL Result Cache is a very powerful mechanism, but it has many associated costs, both in terms of memory and CPU. For this reason, despite its benefits, it should only be deployed to a production environment under very tight controls and only after proper testing, especially from a scalability point of view.

Summary

Understanding how SQL and PL/SQL work together is critical for good database system development. The conceptual differences between these languages are large, but they complement each other in a way that is unique in the industry. SQL does the "heavy lifting" of data retrieval while PL/SQL handles the procedural logic. Together, they form the backbone of good database-centric development. SQL continues to expand its capabilities. It can do more and more that used to only be possible using PL/SQL. However, PL/SQL still allows you to do things that could not be done well or at all in SQL. The trick is knowing when and why to use each language to leverage its strengths and maintain good system performance and functionality.

About the Author

Michael Rosenblum is a Software Architect/Senior DBA at Dulcian, Inc. where he is responsible for system tuning and application architecture. Michael supports Dulcian developers by writing complex PL/SQL routines and researching new features. He is the co-author of PL/SQL for Dummies (Wiley Press, 2006), contributing author of Expert PL/SQL Practices (APress, 2011), and author of a number of database-related articles (IOUG Select Journal, ODTUG Tech Journal) and conference papers. Michael is an Oracle ACE, a frequent presenter at various Oracle user group conferences (Oracle OpenWorld, ODTUG, IOUG Collaborate, RMOUG, NYOUG, etc), and winner of the ODTUG Kaleidoscope 2009 Best Speaker Award. In his native Ukraine, he graduated summa cum laude from the Kiev National Economic University where he received a Master of Science degree in Information Systems.