The Power of DECODE()

Doug Burns

Introduction

This paper is about one of the functions in Oracle's implementation of SQL. Doesn't sound too exciting, does it? Except that DECODE() is no ordinary function, but a core programming construct which adds enormous power to SQL. It allows us to include a small, but significant, amount of post-retrieval procedural logic into our SQL code, without the need to develop PL/SQL or use another 3GL. Over the many years that I've used it across a variety of Oracle versions, using a large number of different client tools, it hasn't changed. That's what I call core. I'm unhappy to report, though, that at almost every company I've worked for (including software houses), I've had to introduce people to the techniques outlined in this document.

But I'm not a programmer any more and functions, expressions and syntax diagrams bore me, so why take such an interest in a programming technique? For the same reasons that every DBA ends up becoming a part-time SQL consultant - Performance. There are some situations where, short of re-development using a 3GL, DECODE is simply the only way to make a complex report run quickly enough.

This paper will explain how to use DECODE to write extremely powerful individual SQL statements which perform one pass of the relevant data and then break it up and format it to suit your reports, instead of using multiple separate SQL statements. The performance improvements are near linear in many cases. By this I mean you can combine 10 queries, each of which performs a full table scan, into one query which will perform one full table scan and the single query will take about the same length of time as one of the original 10 queries. I've done this many times, it works fantastically well and yet I'm surprised how rarely I see the technique used. When I was preparing this paper, a reviewer asked the question 'So, why isn't it used more often?'. I don't have an answer to that question yet, but I hope this paper will help to improve the situation.

Remember those boring syntax diagrams I mentioned? Well, sometimes there is no substitute for technical detail, so let's get that bit out of the way first.

Definition

The first thing to note is that the DECODE() function is defined in the Expr section of the Oracle *SQL Language Reference Manual*. This offers our first hint of the power of DECODE, because it indicates that we can use this function wherever we might use any other expression, in the SELECT, WHERE or ORDER BY clauses for example. So, here is the formal definition.

```
DECODE(expr, search1, result1 [,search2, result2 ….] [,default] )
```

Where: -

expr is any valid expression, which could include column names, constants, bind variables, function calls or arithmetic expressions. (E.g. 12039, 'A String', emp.empno, emp.sal * 1.15). This will represent the *A* value in an *A* <-> *B* comparison, so this is the expression that we're going to test the value of.

The next two parameters appear together as one or more repeating pairs (e.g. search1/result1, search2/result2)

search[1-n] is any valid expression, which will represent the *B* value in the comparison. Note that, if *search1* and *expr* are different data types then Oracle will attempt to convert *expr* and all *search* expressions to the same type as *search1*, but it is best not to depend on this automatic type casting. Another important point to note is that null equals null in a DECODE expression.

result[1-n] is the result returned by the function (or alternatively, the value of this expression) if the preceding *search* value matched *expr*. Oracle will only evaluate the minimum number of *search* expressions necessary to return the correct result. For example, if the server evaluates *search1* and discovers that *expr* = *search1* then the

function will return *result1* and will never evaluate *search2*, *search3* etc. For this reason, you should order the *search* expressions in decreasing likelihood of matching *expr,* from left to right. This will minimise the small amount of processing overhead required to evaluate the expressions.

default is the value returned if none of the *search* expressions matches *expr*. If no value is specified for *default* and none of the *search* expressions match *expr*, then Oracle will return null.

All of which is a slightly long-winded way of describing a very simple principle. Those of you with previous programming experience in other languages may find it simpler to understand a DECODE expression as a variation on an if … then … elseif … type of structure.

```
if (expr == search1)
      return(result1);
elseif (expr == search2)
      return(result2);
…

elseif (expr == searchn)
      return(resultn);
else
      return(default);
```

The switch structure used in C, Java and other languages is probably a more accurate equivalent.

```
switch ( expr ) {
      case search1 :
            return(result1);

      …

      case searchn :
            return(resultn);
      default :
            return(default);
      }
```

Basic Usage

Now let's turn to our first example. All of the examples included are designed to work against the standard SCOTT.EMP and SCOTT.DEPT tables and have been tested against Personal Oracle 7.2, Personal Oracle 8.0.3 and Oracle Lite 3.5. All the execution plans were generated by the SQL*Plus Autotrace facility, running against Personal Oracle 8.0.3 and using the rule-based optimiser. This is useful for demonstration purposes because it's more likely to give consistent execution plans across the small data volumes in the example tables and real-world volumes. It also means that the results should be the same on any database without the need to generate any statistics on the example tables. (i.e. Do try this at home. If the SCOTT example tables are in place, these examples should work unmodified.) Whether you use the cost or rule-based optimiser, the principles still hold true.

Example 1 is a simple example, illustrating the way in which DECODE is most commonly used, to improve report formatting.

Example 1

```
SELECT ename Name,
       DECODE(deptno, 10, 'Accounting',
                      20, 'Research',
                      30, 'Sales',
                      40, 'Operations',
```

```
                                'Unknown') Department
FROM emp;

NAME       DEPARTMENT
---------- ----------
KING       Accounting
BLAKE      Sales
CLARK      Accounting
JONES      Research
MARTIN     Sales
ALLEN      Sales
TURNER     Sales
JAMES      Sales
WARD       Sales
FORD       Research
SMITH      Research
SCOTT      Research
ADAMS      Research
MILLER     Accounting

14 rows selected.
```

This statement checks the deptno column of the emp table and if the value = 10, it displays 'Accounting'; if it's 20, it displays 'Research'; and so on. Notice as well that a default value of 'Unknown' has been specified, which will be displayed if the deptno column does not equal 10, 20, 30 or 40.

Of course, this is not only a simple example of the DECODE function in action, but also of flawed programming practice because the statement assumes that the hard-coded deptno value will not be modified. For a situation like this, we would probably join to a reference table (dept) containing the correct department name. For quick ad-hoc queries, however, this can be much more efficient and there are certain values which it may be safe to hard-code in this way (e.g. Male/Female)

Although translating code values into readable descriptions in reporting applications is the most common and obvious use of DECODE (particularly given the name of the function), it masks some of the more powerful general functionality which we will turn to next.

Logic-driven Column Calculations

Imagine a situation where the HR Manager requests a report to examine the effect of giving everyone in the Sales department a 20% salary increase. The report needs to give the total salary bill for each department in the company. This entails calculating the total of the emp.sal column for all employees in each department, which is straightforward using GROUP BY and SUM as shown in Example 2a.

Example 2a

```
SELECT d.dname department,
       SUM(e.sal) salary_total
FROM  emp e, dept d
WHERE d.deptno = e.deptno
GROUP BY d.dname;
```

Returning a different value for employees in the Sales department adds complication. There are a few possible solutions. We could use two different copies of the emp table in the FROM clause, or we could use a UNION of two complementary data sets, Sales and non-Sales employees, as shown in Example 2b.

Example 2b

```
SELECT d.dname department,
       SUM(e.sal) * 1.2  salary_total
FROM  emp e, dept d
WHERE d.deptno = e.deptno
AND   d.dname = 'SALES'
GROUP BY d.dname
UNION ALL
SELECT d.dname department,
       SUM(e.sal) salary_total
FROM  emp e, dept d
WHERE d.deptno = e.deptno
AND   d.dname != 'SALES'
GROUP BY d.dname;

DEPARTMENT     SALARY_TOTAL
-------------- ------------
ACCOUNTING             8750          (Returned by second half of UNION)
RESEARCH              10875                "              "
SALES                 11280          (Returned by first half of UNION)
```

Although this statement will produce the desired results it will perform two full table scans against the emp table to return the complementary data sets which are then UNIONed. (Note that we have used UNION ALL because we know that the two data sets are already complementary, so no SORT UNIQUE step is necessary to exclude the intersection between the sets.) The execution plan generated by Personal Oracle 8.0.3 for this query is as follows

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   UNION-ALL
   2    1     SORT (GROUP BY)
   3    2       NESTED LOOPS
   4    3         TABLE ACCESS (FULL) OF 'EMP'
   5    3         TABLE ACCESS (BY INDEX ROWID) OF 'DEPT'
   6    5           INDEX (UNIQUE SCAN) OF 'PK_DEPT' (UNIQUE)
   7    1     SORT (GROUP BY)
   8    7       NESTED LOOPS
   9    8         TABLE ACCESS (FULL) OF 'EMP'
  10    8         TABLE ACCESS (BY INDEX ROWID) OF 'DEPT'
  11   10           INDEX (UNIQUE SCAN) OF 'PK_DEPT' (UNIQUE)
```

The only reason that we require two scans of emp is to return all the non-Sales employees and their salaries in one data set, using SUM(emp.sal) to calculate the departmental salary bills; and to return another data set containing the employees in Sales, using SUM(emp.sal * 1.2) to calculate the Sales departments salary bill. The two sets are then UNIONed. We can optimise this query using DECODE by retrieving all of the employee salary information in one scan of the emp table and then including logic in the SELECT clause to selectively apply a calculation to the results for the Sales employees, as shown in example 2c.

Example 2c

```
SELECT d.dname department,
       SUM(DECODE(d.dname,
                  'SALES', e.sal * 1.2,
                           e.sal))  salary_total
FROM  emp e, dept d
WHERE d.deptno = e.deptno
GROUP BY d.dname
```

```
/

DEPARTMENT     SALARY_TOTAL
-------------- ------------
ACCOUNTING             8750
RESEARCH              10875
SALES                 11280
```

Although the results are identical and the two statements are functionally equivalent, it is clear from the execution plan that this will require only one full scan of the emp table, which would represent a significant performance improvement if emp was a large table.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   SORT (GROUP BY)
   2    1     NESTED LOOPS
   3    2       TABLE ACCESS (FULL) OF 'EMP'
   4    2       TABLE ACCESS (BY INDEX ROWID) OF 'DEPT'
   5    4         INDEX (UNIQUE SCAN) OF 'PK_DEPT' (UNIQUE)
```

So let's look at what we've changed. We'll leave the SELECT clause until last (because that is where the most significant changes are) and exclude the more straightforward parts of the statement first.

We still need to select FROM the same two tables and to GROUP BY the department name, so no change in those two parts of the statement. We know we're interested in all employees in the company so let's eliminate the department name check from the two different WHERE clauses in the first statement which leaves us with two identical WHERE clauses which facilitate the join between the emp and dept tables. Now that the two WHERE clauses are identical they return the same rows so we can reduce everything to one data set, with no need for the UNION any more. In fact, the query is starting to look like example 2a.

```
FROM emp e, dept d
WHERE d.deptno = e.deptno
GROUP BY d.dname
```

This leaves us with our new SELECT clause to look at. The first column remains the same - the department name from the dept table. The second column specification contains some of the logic which we've moved from the WHERE clauses of the UNION. It uses the SUM() function to generate a total salary for all the employees in the department but uses different values for salary, depending on whether the department is SALES or not. So here is a high-level procedural view of how example 2c works.

Action	**SQL Clause**
FOR EACH department	`GROUP BY d.dname`
Generate the total salary for that department, using …	`SUM`
FOR EACH employee	`DECODE …`
IF the related department (d.dname) is 'SALES', then	
Use salary (e.sal) * 1.2	
ELSE (by default)	
Use the employee's current salary (e.sal)	

Example 2c illustrates a couple of DECODE capabilities that we haven't used in earlier examples, which both result from DECODE's very open, generic model. First, the returned value (emp.sal or emp.sal*1.2) is not a translation of the value that we are testing against (dept.dname) – it isn't even in the same table. Second, DECODE will either return a field (emp.sal) or a calculation (emp.sal*1.2). It is important to remember that all of the parameters to DECODE can be complex expressions of any type.

Mind the Trap!

For our next example, we need to imagine that our emp and dept tables contain much larger volumes of data than the example tables supplied with Oracle, say 500 departments and many thousands of employees. Our previous report would have generated figures for all 500 departments and now the HR Manager says 'Very nice, but I really only want to look at the total salary figure for the Sales and Research departments'. Not wanting to disappoint our favourite user, we couldn't possibly suggest a little spreadsheet activity on the manager's part, so we rework our DECODE, as shown in example 3a.

Example 3a

```
SELECT SUM(DECODE(d.dname,
                        'SALES',    e.sal * 1.2,
                        'RESEARCH', e.sal)
          )       total_salary_bill
FROM  emp e, dept d
WHERE d.deptno = e.deptno
/

TOTAL_SALARY_BILL
-----------------
            22155
```

This will produce the total of SALES employee's salaries * 1.2, plus the total of RESEARCH employee's salaries. Where an employee is not in either of these two departments the default value of NULL will be returned, so that they won't affect the total. HR gets what it's looking for, we go down the pub and everyone's happy. Well, not really. The problem here is that one of the main strengths of DECODE can encourage us to produce logically consistent but inefficient code if we're not careful. The thing to keep in mind is that DECODE is a post-retrieval function. What this example will do is happily (if slowly) trawl through every employee's data, then apply a DECODE function to it in such a way as to exclude most employees from the result. So maybe we shouldn't read the employees data in the first place! A better way of achieving the same result is shown in Example 3b.

Example 3b

```
SELECT SUM (DECODE(d.dname,
                        'SALES',    e.sal * 1.2,
                        'RESEARCH', e.sal)
          )       total_salary_bill
FROM  emp e, dept d
WHERE d.deptno = e.deptno
AND d.dname IN ('SALES', 'RESEARCH')            /* New Line */
/
```

There is one small difference, a new line in the WHERE clause which restricts the query to employees in the SALES and RESEARCH departments. In practice, the situation is complicated by whether Oracle is able to use indexes to optimise the performance of the WHERE clause to reduce the amount of work that the server needs to perform, but I suggest you follow this basic approach in all cases. There is no point in using DECODE functions to improve the execution plans of SQL statements, only to introduce new performance problems. Remember, remember, remember – DECODE is a post-retrieval function.

Beyond Equality

The first few examples that we have looked at have tested whether columns are equal to specified values. However, it won't be long before someone asks us to produce a report based on columns being within a certain range. The HR

Manager may want to know what the salary bill for the entire company would be if all employees whose salary was less than £10,000 received an increase of £1,000.

Using our newly acquired skills, our first attempt may look something like this.

Example 4a

```
SELECT SUM(DECODE(e.sal,
            < 10000, e.sal + 1000,
            e.sal)) projected_salary_bill
FROM  emp e
/

< 10000, e.sal + 1000,
                *
ERROR at line 2:
ORA-00936: missing expression
```

This statement isn't valid because Oracle doesn't treat '< 10000' as a valid expression to compare e.sal to. It's easy to doubt the value of DECODE when you first encounter this problem because it's difficult to imagine any significant application without range-checking abilities. Fortunately, a closer look at the variety of other functions at our disposal suggests that we could combine DECODE with other functions to support range checking, including the SIGN() and TRUNC() functions and the functions that we will look at here, GREATEST() and LEAST(). In the past, I tended to use and recommend the SIGN() function because it supports all combinations of <, > and = with one function call. The advantage of GREATEST and LEAST is that they are more widely available across Oracle versions (e.g. SIGN is not available in Oracle Lite 3.5, whereas GREATEST and LEAST are)

The definitions are very simple.

```
GREATEST(expr, [expr, …])
LEAST(expr, [expr, …])
```

Both functions accept a list of one or more expressions and return whichever expression is the greatest or least respectively. We'll use the GREATEST function to develop a working replacement for Example 4a.

Example 4b

```
SELECT SUM(DECODE(GREATEST(e.sal, 10000),
                        10000, e.sal + 1000,
                        e.sal)) projected_salary_bill
FROM emp e;

PROJECTED_SALARY_BILL
---------------------
                43025
```

For each employee, this example compares the greater of the employee's salary and 10,000 and then, if the result is 10,000 (which would mean that the employee's salary is lower) it applies the salary increase.

There are a number of built-in functions that become even more powerful when combined with DECODE and these are well worth some additional investigation. If there isn't a function that supports what you are looking to achieve, you could develop your own function using PL/SQL and reference it within the parameters of the DECODE function.

Multi-Part Logic

DECODE function calls can be nested to support multi-part logic. For example, if we want to repeat our previous example, but restrict our salary increase of £1000 to those employed in the SALES department, we could use the following statement.

Example 5a

```
SELECT d.dname department,
      SUM(DECODE(GREATEST(e.sal, 10000),
            10000, DECODE(d.dname,
                        'SALES', e.sal + 1000,
                        e.sal),
            e.sal)) projected_salary_bill
FROM  emp e, dept d
WHERE d.deptno = e.deptno
GROUP BY d.dname
/

DEPARTMENT     PROJECTED_SALARY_BILL
-------------- ---------------------
ACCOUNTING                      8750
RESEARCH                       10875
SALES                          15400
```

Oracle first checks to see if the employee's salary is less than £10000, using the GREATEST function. If it is, then Oracle will check whether the employee's department is SALES. If both tests are true, then e.sal+1000 will be returned. In all other cases, e.sal will be returned because that is the default value of both the inner and outer DECODEs.

Nested DECODEs facilitate AND logic (i.e. if the inner test AND the outer test are true), but what about implementing OR logic using DECODE? In practice, this is very simple. All that's required is for us to implement multiple *search* values to test the initial *expr* against which return the same *result*. If we wanted to modify our previous example to show the result of giving the same salary increase if the employee's department is SALES OR RESEARCH, it might look something like this.

Example 5b

```
SELECT d.dname department,
   SUM(DECODE(GREATEST(e.sal, 10000),
           10000, DECODE(d.dname,
                        'SALES', e.sal + 1000,
                        'RESEARCH', e.sal + 1000,
                        e.sal),
           e.sal)) projected_salary_bill
FROM       emp e, dept d
WHERE d.deptno = e.deptno
GROUP BY d.dname
/

DEPARTMENT     PROJECTED_SALARY_BILL
-------------- ---------------------
ACCOUNTING                      8750
RESEARCH                       15875
SALES                          15400
```

User-definable Report Ordering

Because DECODE is treated in the same way as any other expression, it can appear in the SELECT, WHERE and ORDER BY clauses. We can use this to develop a generic reporting statement with parameter-driven ordering.

Example 6

```
SELECT d.dname department,
       e.ename employee,
       e.sal salary
FROM   emp e, dept d
WHERE d.deptno = e.deptno
ORDER BY DECODE(&&Order_Option,
                  'E', e.name,
                  'D', d.dname,
                  d.dname)
/
```

This statement will generate a report containing all employees, their salary and the department that they work in. If the Order_Option parameter is 'E', the report will be ordered by the employee's name and if it is 'D', by the employee's department. If the parameter is not one of these values, the report will be ordered by the employee's department name by default. This may be useful in reducing the amount of code contained in an application, but there is one very important factor to be aware of. As a direct consequence of the DECODE function in the ORDER BY clause, Oracle will not be able to use an index to control the ordering of the report so it will always perform a sort. That might not be too much of a problem for certain reports, but restricts the usefulness of this technique.

A Real-world Example

I first encountered the power of DECODE a number of years ago whilst working on a government project alongside Oracle consultants. We were implementing an Application Form Processing system that would be used to maintain a record of every form sent out and received in one very large EVENTS table with 20-25 million rows (it seemed big at the time). The design was based on a legacy of an earlier system that we had to co-exist with. One of the key functions of the system was the production of management information reports, used to measure performance against key indicators. (e.g. 'How many forms did we send out last week?', 'How long does it take us to send a reply after we have received form S937?', 'Just what *are* all these forms for?'.)

We were using a very early release of Reports 2 to help us to combine various queries to generate high quality report output with a standard format template. This was initially very successful against our few thousand test records, as these things usually are, until we increased the volume of our test data and realised that our reports would take hours and days to run. In the shadow of a looming deadline, we consulted Oracle for advice and were encouraged to reduce the number of individual queries in each Reports2 Data Model so we spent a couple of desperate days re-thinking our approach. The DECODE function came to our rescue, reducing the execution time of our reports from hours to a few minutes and days to a few hours.

The EVENTS table in the database contained a row for each event during the application handling process. For the purposes of our example we'll use the EVENTS table defined below, which uses a subset of the columns from the original table.

EVENTS

EVENT_TYPE	VARCHAR2(2)
EVENT_DATE	DATE

Receipt Events	AX, AH, AB
Send Events	TY, LO, LL
Other Events	QC, DL, MA, IC, JF

Receipt events were generated whenever a form was received back from the applicant, Send events when the organisation sent a form to the applicant and a small number of other events covered a variety of activities. The report needed to show the number of receipt and send events in the last year and also in the last 2 days. For example,

```
Total Receipts    Total Sends       Recent Receipts   Recent Sends
2,354,456         1,857,374         23,950            21,349
```

Our first attempt used one discrete query for each of the values shown in the report, using Reports2 to combine the results. For example, the Total Receipts value was generated using the SQL shown in example 7a (but remember that there were 4 of these queries used in the report, each performing a full scan of our big events table).

Example 7a

```
SELECT COUNT(1)
FROM events
WHERE event_type IN ('AX', 'AH', 'AB')
AND event_date > ADD_MONTHS(sysdate, -12)
/
```

Our first thought about how to improve the report was that, given that we were interested in all the receipt and send events during the past year, why not have one query retrieve both sets of events in one pass of the EVENTS table and then use DECODE to generate two different SUMs for the different events? Once we'd achieved that it occurred to us that, since we were retrieving the event data for the last year, we should be able to use DECODE against the same scan of the table to extract the figures for events in the last 2 days separately. The end result is shown in example 7b.

Example 7b

```
SELECT SUM(DECODE(event_type,
                  'AX', 1, 'AH', 1, 'AB', 1)) total_receipts,
SUM(DECODE(event_type,
                  'TY', 1, 'LO', 1, 'LL', 1)) total_sends,
SUM(DECODE(GREATEST(event_date, TRUNC(SYSDATE - 2)),
                  event_date, DECODE(event_type,
                              'AX', 1, 'AH', 1, 'AB', 1)
                  ))      recent_receipts,
SUM(DECODE(GREATEST(event_date, TRUNC(SYSDATE - 2)),
                  event_date, DECODE(event_type,
                              'TY', 1, 'LO', 1, 'LL', 1)
                  ))      recent_sends
FROM events
WHERE event_type IN ('AX', 'AH', 'AB', 'TY', 'LO', 'LL')
AND event_date > ADD_MONTHS(SYSDATE, -12)
```

There are several important things to note about this statement. First, notice the way that we use a check against event_type in the WHERE clause to reduce the number of events retrieved *before* applying the DECODE logic in the SELECT clause. Likewise with event_date, because we only want this year's events. The resulting WHERE clause will return all send and receive events from the past year.

Then the SELECT clause does the real work. If we look at the first column definition, we will see that we are returning the sum of 1 if the event_type is AX, AH or AB (i.e. a receipt event) or NULL (the default) if it is any other type of event. This has the effect of returning the total number of receipt events. The second column definition is almost identical, but for send events. The third column uses a nested DECODE to first check the event_date to see if it is within the past 2 days and, if it is, an inner DECODE to test whether it's a receipt event. If both are true, 1 is

added to the SUM, if either is not true, nothing (NULL) is added to the SUM, giving us the total number of receipt events in the past 2 days.

The changes resulted in the report running almost 4 times more quickly, because we were able to produce the results of 4 separate queries using only one query which scanned through the same data and then sub-divided it to produce the 4 separate totals. Remember that this report only contains 4 values, whereas some of the reports contained 25, 50 or 100! This use of SUM, DECODE and return values of 1 or NULL is an excellent way of using one SQL statement to perform large numbers of apparently unrelated COUNTs against one data set, instead of performing numerous SELECT COUNT(1) queries with different WHERE clauses. This technique is particularly useful for MIS reporting against data warehouses.

The Down-side

Before we summarise the strengths of the DECODE function, let's focus on some of the potential weaknesses if we don't use it appropriately. Most of these are related to coding style and are therefore under our control to a certain extent. The first is that of code readability. It should be clear from Example 7 that clean and readable code formatting is essential when using a number of DECODEs, particularly if they are nested. The example I have shown was one of the smaller queries from the application concerned and when some individual queries grew to over 50 lines, they could become difficult to understand or maintain. The best approach is to develop clear coding standards from the start, which should include some form of alignment of indentations to make the individual components of the DECODE functions very clear.

The second, which was mentioned earlier, is that DECODE is a post-retrieval function and it is easy to write code which is spectacularly inefficient but functionally correct. As a first principle for any SQL statement you may write, you should attempt to reduce the rows retrieved using the best access path and the most selective WHERE clause. Only then should you use DECODE for additional processing.

The final potential problem with DECODE is that it is an Oracle-specific extension and is not included in the ANSI standards. This is a problem if code-portability to other databases is an issue, but shouldn't distract you from the extra power in the Oracle implementation. Many of us use PL/SQL, after all!

Conclusion (The Up-side)

DECODE gives us the power to not just retune the access paths of our SQL queries, but to take a step back from our code, look at our requirements and take a completely different approach to the task. Instead of limiting our tuning efforts to improving the speed of individual queries, we can reduce the overall number of queries to retrieve all the data we require and then use DECODE to perform certain post-retrieval tasks. This reminds me of one of the first pieces of Oracle tuning advice I heard, which still holds true today. Reduce the number of 'trips' to the database to the minimum required to achieve the objective. If a report is performing multiple accesses against the same tables it is worth examining whether these might be combined.

DECODE bridges the gap between pure SQL and embedding SQL in 3GLs. In some cases, the only reason that we use a 3GL is to perform cursor loops that allow us to apply additional processing to the data, row by row, based on the column values. We can often use DECODE to perform that additional processing instead.

DECODE works with all versions of Oracle and isn't dependant on optimiser improvements in newer versions. This is because the performance advantage comes from taking a different approach to the problem that requires Oracle to perform less work, regardless of which optimiser is in use. The way I see it, the optimiser can only really be expected to optimise your access paths, not attempt to rewrite your code more efficiently (although I'm sure this will happen in time). Like most performing tuning activities, the big improvements come from making smart decisions about your approach in advance.

About the Author

Doug Burns is an independent consultant who has 8 years experience working with Oracle in a range of industries and countries, from Oil Companies in Dharhan to Internet Service Providers in Amsterdam. He is currently assisting ICL with a server implementation at the national railway company in Romania, but is available for other

assignments. He has taught Oracle DBA courses for Oracle UK in the past and still spends part of his time teaching DBA courses for Learning Tree International. He can be contacted at dougburns@hotmail.com. When not working or answering his 7 year old daughter Mica's impossible questions about life and why Windows applications crash, Doug plays guitar badly and tracks the ascendant star of Glasgow Celtic Football Club.

References/Acknowledgement

This first version of this paper was written in 1994, using **Oracle7 Server SQL Language Reference Manual (Part No. 778-70-1292) (*December 1992*)** as the only direct source of information, but I have since checked syntax validity with the online documentation for versions up to 8i

As I was reviewing the paper, I decided to search for other references on the same subject. Although I didn't use these as sources for this paper, they are good sources of additional information. First, there is a chapter in **Oracle - The Complete Reference (Oracle Press) (ISBN 0-07-882285-8) by Koch and Loney** which, as well as having an almost identical title to this paper(!) covers very similar ground, with additional examples. Second, there are a number of sources on the web, particularly **http://www.doag.org/mirror/frank/faqsql.htm**

Thanks to Paul from Oracle UK's Edinburgh office for introducing me to the techniques described in this paper during 3 hectic months. Now, if only I could remember his surname!