Improving SQL efficiency using CASE

Some time ago I wrote 'The Power of Decode' - a paper on using the DECODE function to improve report performance. I was aware at the time that DECODE was being replaced by CASE but wanted to make sure that the paper applied to as many Oracle versions as possible. CASE was introduced in Oracle 8.1.6, however, and is a much better option because it is

1) More flexible than DECODE
2) Easier to read
3) ANSI-compatible (if that matters to you)

However, CASE is essentially a better implementation of DECODE so the reasons for using either are similar. In this article I'll focus on improving application performance by improving the efficiency of your code. One of the first and most valuable lessons I learnt about Oracle performance is to do as much work in as few steps as possible. The Oracle server engine is designed to handle large data sets efficiently but sometimes developers try to break them up into smaller discrete pieces of work (the row-by-row approach). I suspect that they feel they have more control this way and it maps on to a typical developer's procedural approach, but it normally isn't the most efficient way of accessing an Oracle database.

I often see reports developed using reporting tools or by embedding SQL in other languages, that include several SQL statements accessing the same tables in slightly different ways to retrieve individual pieces of data in the report layout. Each of the individual SQL statements is a separate request to the database and causes work at the server end.

> *To give you a trivial example, why do this?*

```
SELECT deptno, SUM(sal) FROM emp WHERE deptno = 10
GROUP BY deptno;

SELECT deptno, SUM(sal) FROM emp WHERE deptno = 20
GROUP BY deptno;
```

> *When you could retrieve the same results using this.*

```
SELECT deptno, SUM(sal) FROM emp WHERE deptno IN (10,20)
GROUP BY deptno;
```

Any technique that offers the possibility of using fewer SQL statements to achieve the same end result may have a beneficial effect on performance. Analytic functions can be a big help in this area but CASE and DECODE have their place too.

Definition

The first thing to note is that CASE expressions are defined in the *Expressions* chapter of the Oracle *SQL Language Reference Manual*. This offers our first hint of the power of CASE, because it indicates that we can use it wherever we might use any other expression, in the SELECT, WHERE or ORDER BY clauses for example.

I like Oracle's high level description of CASE which sums up what we're going to use it for.

> *"CASE expressions let you use IF ... THEN ... ELSE logic in SQL statements without having to invoke procedures."*

Note that there's no need to use a procedural language – it's all available in a single SQL statement. Here are the formal definitions of the two variants

Simple CASE Expression

```
CASE expr WHEN comparison_expr_1 THEN return_expr_1
      [WHEN comparison_expr_2 THEN return_expr_2 ….]
      [ELSE default] END
```

Where: -

Expr is a valid expression that is evaluated once.

Comparison_Expr_(1-n) are compared to the *Condition*

Return_expr_(1-n) are the results returned if the matching *Expr* = *Condition*

default is the value returned if none of the *Comparison_Expr*s = *Expr*. If no value is specified for *default* and none of the *Comparison_Expr*s = *Expr*, then CASE will return NULL.

Searched CASE Expression

```
CASE WHEN condition_1 THEN return_expr_1
      [WHEN condition_2 THEN return_expr_2 ….]
      [WHEN condition_n THEN return_expr_n ….]
       [ELSE default] END
```

Where: -

Condition_(1-n) are valid expressions that could be evaluated to TRUE (e.g. amount_sold > 1000; cust_last_name = 'BURNS'; a.amount_sold / a.unit_price > b.amount_sold / b.unit_price)

Return_expr_(1-n) are the results returned if the matching condition was true.

default is the result returned if none of the WHEN conditions evaluates to TRUE. If no value is specified for *default* and none of the WHEN conditions are TRUE, then CASE will return NULL.

So Oracle will evaluate each condition and as soon as one of them is TRUE, it will return the related expression that follows the THEN keyword and then exit the CASE structure. The difference between the Searched Case and Simple Case is that the latter compares a single expression against possible results, whereas the Searched Case expression allows us to test multiple conditions which may not be related.

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. (It's the Searched Case Expression variant I'm using here)

```
if (condition1)
      return(result1);
elseif (condition2)
      return(result2);
…

elseif (conditionn)
      return(resultn);
else
      return(default);
```

To finish off the definition of CASE expressions there are some important data type rules highlighted in this section of the documentation

> *"For a simple CASE expression, the expr and all comparison_exprs must either have the same datatype (CHAR, VARCHAR2, NCHAR, or NVARCHAR2, NUMBER, BINARY_FLOAT, or BINARY_DOUBLE) or must all have a numeric datatype. If all expressions have a numeric datatype, then Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that datatype, and returns that datatype.*
>
> *For both simple and searched CASE expressions, all of the return_exprs must either have the same datatype (CHAR, VARCHAR2, NCHAR, or NVARCHAR2, NUMBER, BINARY_FLOAT, or BINARY_DOUBLE) or must all have a numeric datatype. If all return expressions have a numeric datatype, then Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that datatype, and returns that datatype."*

Basic Usage

Okay, that's the boring bit out of the way and it's time to turn to the first example. All of the examples included are designed to work against the sample SH (sales history) schema that has been available since Oracle 9i. I selected this because

- It contains a reasonable volume of data, including the 900,000+ row SALES table.
- I think it's a fair reflection of a business application.
- It has a standard published definition and sensible table and column names. Full documentation for the schema is available in the Sample Schemas manual in the generic documentation set. This means that you can create the same schema (if it's not already loaded into your database), test the examples and play around with different approaches.

I ran the examples against Oracle 10.1.0.4.0, but you should find identical results on any version of 9i or 10g. (I'd be extremely interested in any variations you might come across.) I've used the cost-based optimiser and the execution plans are generated using the SQL*Plus Autotrace facility.

Example 1 illustrates the way in which DECODE was often used to improve report formatting.

Example 1

```
SELECT cust_id, cust_first_name, cust_last_name,
       CASE cust_gender
              WHEN 'M' THEN 'Male'
              WHEN 'F' THEN 'Female'
       ELSE 'UNKNOWN'
       END gender
FROM customers
WHERE ROWNUM < 6;

   CUST_ID CUST_FIRST_NAME      CUST_LAST_NAME                           GENDER
---------- -------------------- ---------------------------------------- ------
     49671 Abigail              Ruddy                                    Male
      3228 Abigail              Ruddy                                    Male
      6783 Abigail              Ruddy                                    Male
     10338 Abigail              Ruddy                                    Male
     13894 Abigail              Ruddy                                    Male
```

This statement checks the cust_gender column of the customers table and if the value = ‘M’, then it returns 'Male' or if it's ‘F’ it returns ‘Female’. I’ve included a default clause that displays ‘UNKNOWN’ if it’s not one of the two expected values.

The ROWNUM test limits the output for the example because there are 55,500 customers! That’s one aspect of the new sample schemas that can make them harder to work with than the old EMP and DEPT – sometimes you only want a small output example.

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) and CASE, it masks some of the more powerful general functionality which I’ll turn to next.

Logic-dependent Aggregation

Imagine a situation where the Sales Manager requests a report to examine the effect on 2001 (calendar year) revenue of applying a 10% mark-up on Photo-related products. The report needs to give the total revenue for each product category and subcategory. This entails calculating the total of the sales.amount_sold column for all products, which is straightforward using GROUP BY and SUM as shown in Example 2a.

Example 2a

```
REM First a few SQL*Plus formatting commands

SET PAGES 999
SET LINES 160
COLUMN prod_category FORMAT A30
COLUMN prod_subcategory FORMAT A26
COLUMN dollars FORMAT 999,999,990.90

BREAK ON prod_category SKIP 1

COMPUTE SUM OF dollars ON prod_category

REM Now the query

SELECT p.prod_category, p.prod_subcategory, sum(s.amount_sold) AS dollars
FROM sales s, times t, products p
WHERE s.time_id = t.time_id
AND  s.prod_id = p.prod_id
AND  t.calendar_year = 2001
GROUP BY p.prod_category, p.prod_subcategory
ORDER BY p.prod_category, p.prod_subcategory;

PROD_CATEGORY                  PROD_SUBCATEGORY                    DOLLARS
------------------------------ ------------------------- ---------------
Electronics                    Game Consoles                1,205,027.35
                               Home Audio                   2,779,398.57
                               Y Box Accessories              161,004.00
                               Y Box Games                    559,421.03
****************************                                ---------------
sum                                                         4,704,850.95

Hardware                       Desktop PCs                  2,230,713.39
                               Portable PCs                 3,453,656.62
****************************                                ---------------
sum                                                         5,684,370.01

Peripherals and Accessories    Accessories                    663,034.82
```

```
                              CD-ROM                          669,134.90
                              Memory                        1,228,555.41
                              Modems/Fax                      874,702.07
                              Monitors                      3,191,525.93
                              Printer Supplies              1,232,754.58
****************************                               ---------------
sum                                                         7,859,707.71

Photo                         Camcorders                    2,819,074.98
                              Camera Batteries                757,626.90
                              Camera Media                    551,090.37
                              Cameras                       2,205,836.66
****************************                               ---------------
sum                                                         6,333,628.91

Software/Other                Accessories                     521,342.80
                              Bulk Pack Diskettes              88,216.04
                              Documentation                   827,932.29
                              Operating Systems             1,020,370.87
                              Recordable CDs                  367,478.04
                              Recordable DVD Discs            728,564.36
****************************                               ---------------
sum                                                         3,553,904.40


22 rows selected.
```

Returning a different value for Photo products adds a little complication. There are several possible solutions. We could use two different copies of the sales table in the FROM clause, or we could use a UNION of two complementary data sets, Photo and non-Photo products, as shown in Example 2b.

(Note - at this stage, I'll enable the SQL*Plus AUTOTRACE facility to expose the execution plans of the various approaches to the problem. If you haven't used this before, you can find more information **HERE**)

Example 2b

```
SELECT p.prod_category, p.prod_subcategory,
       SUM(s.amount_sold) * 1.1 AS dollars
FROM sales s, times t, products p
WHERE s.time_id = t.time_id
AND s.prod_id = p.prod_id
AND t.calendar_year = 2001
AND p.prod_category = 'Photo'
GROUP BY p.prod_category, p.prod_subcategory
UNION ALL
SELECT p.prod_category, p.prod_subcategory, sum(s.amount_sold) AS dollars
FROM sales s, times t, products p
WHERE s.time_id = t.time_id
AND s.prod_id = p.prod_id
AND t.calendar_year = 2001
AND p.prod_category != 'Photo'
GROUP BY p.prod_category, p.prod_subcategory
ORDER BY 1, 2;

PROD_CATEGORY                 PROD_SUBCATEGORY                 DOLLARS
----------------------------- ------------------------- ---------------
Electronics                   Game Consoles                 1,205,027.35
                              Home Audio                    2,779,398.57
                              Y Box Accessories               161,004.00
                              Y Box Games                     559,421.03
```

```
******************************                              ---------------
sum                                                            4,704,850.95

Hardware                        Desktop PCs                    2,230,713.39
                                Portable PCs                   3,453,656.62
******************************                              ---------------
sum                                                            5,684,370.01

Peripherals and Accessories     Accessories                      663,034.82
                                CD-ROM                           669,134.90
                                Memory                         1,228,555.41
                                Modems/Fax                       874,702.07
                                Monitors                       3,191,525.93
                                Printer Supplies               1,232,754.58
******************************                              ---------------
sum                                                            7,859,707.71

Photo                           Camcorders                     3,100,982.48
                                Camera Batteries                 833,389.59
                                Camera Media                     606,199.41
                                Cameras                        2,426,420.33
******************************                              ---------------
sum                                                            6,966,991.80

Software/Other                  Accessories                      521,342.80
                                Bulk Pack Diskettes               88,216.04
                                Documentation                    827,932.29
                                Operating Systems              1,020,370.87
                                Recordable CDs                   367,478.04
                                Recordable DVD Discs             728,564.36
******************************                              ---------------
sum                                                            3,553,904.40


22 rows selected.
```

Although this statement will produce the desired results it will perform two full table scans against the sales table to return the complementary data sets which are then UNIONed. (Note that I've used UNION ALL because we know that the two data sets are already complementary.) So the output shown in bold text comes from the first query block, before the UNION ALL, and the rest comes from the second query block.

The execution plan generated by Oracle for this query is as follows

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=879 Card=18 Bytes=1152)

   1    0   SORT (ORDER BY) (Cost=878 Card=18 Bytes=1152)
   2    1     UNION-ALL
   3    2       SORT (GROUP BY) (Cost=433 Card=3 Bytes=192)
   4    3         HASH JOIN (Cost=429 Card=45967 Bytes=2941888)
   5    4           TABLE ACCESS (FULL) OF 'TIMES' (TABLE) (Cost=15 Card=365
                                                                  Bytes=4380)
   6    4           HASH JOIN (Cost=412 Card=183769 Bytes=9555988)
   7    6             TABLE ACCESS (BY INDEX ROWID) OF 'PRODUCTS' (TABLE)
                                                  (Cost=3 Card=14 Bytes=490)
   8    7               INDEX (RANGE SCAN) OF 'PRODUCTS_PROD_CAT_IX' (INDEX)
                                                  (Cost=1 Card=14)

   9    6             PARTITION RANGE (ITERATOR) (Cost=400 Card=918843
                                                          Bytes=15620331)
```

```
  10    9              TABLE ACCESS (FULL) OF 'SALES' (TABLE) (Cost=400
                                                 Card=918843 Bytes=15620331)
  11    2       SORT (GROUP BY) (Cost=444 Card=15 Bytes=960)
  12   11         HASH JOIN (Cost=430 Card=183869 Bytes=11767616)
  13   12           TABLE ACCESS (FULL) OF 'PRODUCTS' (TABLE) (Cost=3 Card=58
                                                              Bytes=2030)
  14   12           HASH JOIN (Cost=424 Card=229837 Bytes=6665273)
  15   14             TABLE ACCESS (FULL) OF 'TIMES' (TABLE) (Cost=15 Card=365
                                                              Bytes=4380)
  16   14             PARTITION RANGE (ITERATOR) (Cost=400 Card=918843
                                                           Bytes=15620331)
  17   16               TABLE ACCESS (FULL) OF 'SALES' (TABLE) (Cost=400
                                                 Card=918843 Bytes=15620331)

Statistics
----------------------------------------------------------
         14  recursive calls
          0  db block gets
       1173  consistent gets
          0  physical reads
          0  redo size
       1306  bytes sent via SQL*Net to client
        519  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          5  sorts (memory)
          0  sorts (disk)
         22  rows processed
```

An Interlude

As this is the first execution plan we've come across, it's worth a brief interlude to examine it in a little more detail. I'll use the step numbers in the first column for reference.

The first important point is that the cost based optimizer chose different plans for the two different result sets which are UNIONed. That's because, although they look very similar, they are interested in different volumes of data, so different access paths are appropriate.

First Query Block (for Photo sales)

a) Steps 8 and 7 retrieve the rows for Photo products from PRODUCTS, using an index range scan. PRODUCTS_PROD_CAT_IX is a non-unique index on the PROD_CATEGORY column. Because Photo products are a small subset of PRODUCTS, Oracle has decided that an indexed retrieval is most efficient.

b) Steps 10, 9 and 6 retrieve the related rows from the partitioned SALES table using a Hash Join against a full table scan of SALES.

c) Steps 5 and 4 retrieve all the related rows from the TIMES table using a Hash Join

d) Step 3 groups the resulting set of data from PRODUCTS, SALES and TIMES for Photo products.

Second Query Block (for non-Photo sales)

e) Step 15 retrieves all of the rows for calendar year 2001 from the TIMES table using a full table scan.

f) Steps 14, 16 and 17 retrieve all of the related rows from the partitioned SALES table using a full table scan and a Hash Join.

g) Steps 13 and 12 retrieve all of the rows from the PRODUCTS table (eliminating Photo products) and then use a Hash Join to join the results to the last rowset. Note that, because we need to retrieve nearly all of the rows from the products table, it's more efficient for Oracle to use a full table scan this time.

h) Step 11 groups the resulting set of data from PRODUCTS, SALES and TIMES for Photo products.

UNION and ORDER BY

i) Step 2 performs a UNION ALL operation on the results from d) and h) above

j) Step 1 performs the final sort of the aggregated results, so that they're ORDERed BY PROD_CATEGORY then PROD_SUBCATEGORY

I like the autotrace facility because it allows me to run the query, see the results, the execution plan and some basic resource usage statistics. However when it comes to reading the execution plan, a nicer facility is probably the DBMS_XPLAN package, so I suggest you read the documentation and try that too.

Interlude over - let's get back to tuning the query.

The only reason that we require two scans of sales is to return all the non-Photo products and their amount_sold in one data set, using SUM(amount_sold); and to return another data set containing the Photo products, using SUM(amount_sold) * 1.1 to calculate the total amount_sold. The two sets are then UNIONed.

We can optimise this query by retrieving all of the amount_sold values in one scan of the sales table and then using CASE to selectively apply a calculation to the results for the Photo products in the SELECT list, as shown in example 2c.

Example 2c

```
SELECT   p.prod_category, p.prod_subcategory,
       SUM(CASE p.prod_category
             WHEN 'Photo' THEN amount_sold *1.1
             ELSE amount_sold
       END) AS dollars
FROM     sales s, times t, products p
WHERE s.time_id = t.time_id
AND s.prod_id = p.prod_id
AND t.calendar_year = 2001
GROUP BY p.prod_category, p.prod_subcategory
ORDER BY 1, 2;
```

```
PROD_CATEGORY                PROD_SUBCATEGORY                    DOLLARS
---------------------------- ------------------------- ---------------
Electronics                  Game Consoles                 1,205,027.35
                             Home Audio                    2,779,398.57
                             Y Box Accessories               161,004.00
                             Y Box Games                     559,421.03
**************************** ***************************---------------
sum                                                        4,704,850.95

Hardware                     Desktop PCs                   2,230,713.39
                             Portable PCs                  3,453,656.62
**************************** ***************************---------------
sum                                                        5,684,370.01
```

```
Peripherals and Accessories    Accessories                     663,034.82
                               CD-ROM                          669,134.90
                               Memory                        1,228,555.41
                               Modems/Fax                      874,702.07
                               Monitors                      3,191,525.93
                               Printer Supplies              1,232,754.58
****************************                                ---------------
sum                                                          7,859,707.71

Photo                          Camcorders                    3,100,982.48
                               Camera Batteries                833,389.59
                               Camera Media                    606,199.41
                               Cameras                       2,426,420.33
****************************                                ---------------
sum                                                          6,966,991.80

Software/Other                 Accessories                     521,342.80
                               Bulk Pack Diskettes              88,216.04
                               Documentation                   827,932.29
                               Operating Systems             1,020,370.87
                               Recordable CDs                  367,478.04
                               Recordable DVD Discs            728,564.36
****************************                                ---------------
sum                                                          3,553,904.40


22 rows selected.
```

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 sales table, which represents a useful improvement.

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=448 Card=75 Bytes=4800)

   1    0   SORT (GROUP BY) (Cost=448 Card=75 Bytes=4800)
   2    1     HASH JOIN (Cost=430 Card=229837 Bytes=14709568)
   3    2       TABLE ACCESS (FULL) OF 'PRODUCTS' (TABLE) (Cost=3 Card=72
                                                          Bytes=2520)
   4    2       HASH JOIN (Cost=424 Card=229837 Bytes=6665273)
   5    4         TABLE ACCESS (FULL) OF 'TIMES' (TABLE) (Cost=15 Card=365
                                                          Bytes=4380)
   6    4         PARTITION RANGE (ITERATOR) (Cost=400 Card=918843
                                                          Bytes=15620331)
   7    6           TABLE ACCESS (FULL) OF 'SALES' (TABLE) (Cost=400
                                          Card=918843 Bytes=15620331)
Statistics
----------------------------------------------------------
          8  recursive calls
          0  db block gets
        587  consistent gets
          0  physical reads
          0  redo size
       1306  bytes sent via SQL*Net to client
        519  bytes received via SQL*Net from client
          3  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
         22  rows processed
```

Example 2c requires half the number of consistent gets and less than half the number of sorts that example 2b does. However, you'll probably notice that if you run this in a single user test environment that, because we're operating on fairly small volumes of data, the time to complete the requests and return the data is very similar – around 1 second in my tests. Which is why generating the execution plans and resource usage statistics is important. If you had many users running this report against larger data sets, the difference would become more noticeable.

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 three tables and to GROUP BY product_category and product_subcategory, so no change in those two parts of the statement. We know we're interested in all products so let's eliminate the product_category check from the two different WHERE clauses in example 2b which leaves us with two identical WHERE clauses which facilitate the joins between the sales, times and product tables and limit the data to the calendar year 2001. 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   sales s, times t, products p
WHERE  s.time_id = t.time_id
AND    s.prod_id = p.prod_id
AND    t.calendar_year = 2001
GROUP BY p.prod_category, p.prod_subcategory
ORDER BY p.prod_category, p.prod_subcategory;
```

This leaves us with our new SELECT clause to look at. The first two grouping columns remain the same - product_category and product_subcategory from the products table. The third 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 amount_sold for all the products in the product_category and product_subcategory but uses different values for amount_sold, depending on whether the category is 'Photo' or not. So here is a high-level procedural view of how example 2c works.

```
FOR EACH product subcategory (GROUP BY prod_category, prod_subcategory)
        Generate the total amount_sold for that product subcategory (SUM)
                IF the related product_category is 'Photo', THEN
                        Use amount_sold * 1.1
                ELSE (by default)
                        Use amount_sold
                END IF
```

Pivot Tables and Multi-part Logic

A common use of CASE is to generate pivot tables or cross-matrix reports (although the new MODEL clause in 10g is more powerful). We might want to modify our previous report to just display sales information for Photo Products but to have *one column per month* for the last quarter of 2001.

To achieve this, we'll select the usual product and sales data, group on the product category and sub-category and apply the 10% mark-up to Photo products. However, we'll also check which month the sale occurred in before adding the amount sold to a given column, one for each month. It's important to remember here that if there is no ELSE clause and none of the conditions is TRUE, the default value of NULL will be returned, so the running SUM for that month will be

unaffected. That's the approach I've used here and the behaviour has been consistent over multiple versions of Oracle, but if you're cautious, you can simply add ELSE 0 or ELSE NULL to each CASE expression.

Example 3a

(Note that this is the first example to use a Searched CASE expression where we have a number of WHEN clauses containing discrete logical tests, rather than comparing each to the same initial test expression. This can be clearly identified because the first WHEN keyword appears immediately after the CASE keyword.)

```
SET LINES 80
COLUMN prod_category FORMAT A16
COLUMN prod_subcategory FORMAT A20
COLUMN oct_dollars format 999,990.90
COLUMN nov_dollars format 999,990.90
COLUMN dec_dollars format 999,990.90
SELECT   p.prod_category, p.prod_subcategory,
   SUM(CASE WHEN p.prod_category = 'Photo' AND t.calendar_month_number = 10
                           AND t.calendar_year = 2001
            THEN amount_sold *1.1
       WHEN p.prod_category != 'Photo' AND t.calendar_month_number = 10
                           AND t.calendar_year = 2001
            THEN amount_sold
   END) AS OCT_dollars,
   SUM(CASE WHEN p.prod_category = 'Photo' AND t.calendar_month_number = 11
                           AND t.calendar_year = 2001
            THEN amount_sold *1.1
       WHEN p.prod_category != 'Photo' AND t.calendar_month_number = 11
                           AND t.calendar_year = 2001
            THEN amount_sold
   END) AS NOV_dollars,
   SUM(CASE WHEN p.prod_category = 'Photo' AND t.calendar_month_number = 12
                           AND t.calendar_year = 2001
            THEN amount_sold *1.1
       WHEN p.prod_category != 'Photo' AND t.calendar_month_number = 12
                           AND t.calendar_year = 2001
            THEN amount_sold
   END) AS DEC_dollars
FROM     sales s, times t, products p
WHERE s.time_id = t.time_id
AND s.prod_id = p.prod_id
GROUP BY p.prod_category, p.prod_subcategory
ORDER BY 1, 2
/

PROD_CATEGORY    PROD_SUBCATEGORY     OCT_DOLLARS NOV_DOLLARS DEC_DOLLARS
---------------- -------------------- ----------- ----------- -----------
Electronics      Game Consoles         105,876.22   88,989.00  193,546.85
                 Home Audio            220,133.02  246,483.07  252,605.84
                 Y Box Accessories      14,769.55   15,101.27   12,499.64
                 Y Box Games            52,945.94   55,197.39   45,690.73

Hardware         Desktop PCs           153,857.16  239,435.84  180,910.18
                 Portable PCs          192,513.25  201,927.44  205,868.81

Peripherals and  Accessories            65,193.15   53,754.82   69,484.05
Accessories

                 CD-ROM                 63,934.53   65,239.50   37,439.03
                 Memory                126,492.82  116,220.35  120,243.47
                 Modems/Fax             85,488.22   75,540.22   82,163.87
```

```
                Monitors              338,795.95  309,261.00  318,457.99
                Printer Supplies      126,890.11  103,908.88   98,610.61

Photo           Camcorders            276,726.14  302,757.06  309,163.10
                Camera Batteries       87,368.25   77,983.22   64,453.74
                Camera Media           55,979.88   60,589.43   46,312.73
                Cameras               222,135.47  217,513.90  250,361.61

Software/Other  Accessories            60,090.11   46,264.68   50,513.14
                Bulk Pack Diskettes     8,816.24    6,497.00    7,066.86
                Documentation          63,202.96   67,330.54   78,742.43
                Operating Systems      89,284.76  106,532.85   96,790.97
                Recordable CDs         22,849.65   20,342.21   26,734.34
                Recordable DVD Discs   74,207.23   57,712.70   60,317.69

22 rows selected.
```

One of the problems with DECODE is that writing multi-part logical expressions using the AND operator can be a little cumbersome as each AND operation would require an additional nested DECODE so, although example 3a might look long-winded and slightly difficult to follow (imagine if the report covered 18 months, rather than 3), the DECODE version would be worse! You're likely to find CASE expressions much easier to work with.

However, I've just fallen into a common trap when using CASE. The logic it allows us to implement is so flexible that it can encourage us to produce logically consistent but inefficient code if we're not careful. The thing to keep in mind is that when you use CASE in the SELECT list, it is a *post-retrieval* function. What this example will do is trawl through all of the sales figures, then apply a DECODE function to it in such a way as to exclude most sales from the result, because they didn't occur in the last quarter of 2001.

Another way of looking at this is that we are using CASE expressions in the SELECT clause to eliminate results that should have been eliminated much earlier, in the WHERE clause. Why retrieve data that we know we are going to discard subsequently! After all, that's the whole point of these techniques, to *reduce the workload* required to produce the reports

A better way of achieving the same result is shown in Example 3b.

Example 3b

```
SELECT   p.prod_category, p.prod_subcategory,
   SUM(CASE WHEN p.prod_category = 'Photo' AND t.calendar_month_number = 10
            THEN amount_sold *1.1
       WHEN p.prod_category != 'Photo' AND t.calendar_month_number = 10
            THEN amount_sold
   END) AS OCT_dollars,
   SUM(CASE WHEN p.prod_category = 'Photo' AND t.calendar_month_number = 11
            THEN amount_sold *1.1
       WHEN p.prod_category != 'Photo' AND t.calendar_month_number = 11
            THEN amount_sold
   END) AS NOV_dollars,
   SUM(CASE WHEN p.prod_category = 'Photo' AND t.calendar_month_number = 12
            THEN amount_sold *1.1
       WHEN p.prod_category != 'Photo' AND t.calendar_month_number = 12
            THEN amount_sold
   END) AS DEC_dollars
FROM     sales s, times t, products p
WHERE s.time_id = t.time_id
AND s.prod_id = p.prod_id
AND t.calendar_year = 2001
AND t.calendar_month_number BETWEEN 10 AND 12
```

```
GROUP BY p.prod_category, p.prod_subcategory
ORDER BY 1, 2
/
```

The additional lines in the WHERE clause, shown in bold text will ensure that we reduce the volume of data that we're processing to the minimum first. We're not interested in any sales data that doesn't occur in the last quarter of 2001, so let's not even bother selecting it and, given that we've just eliminated the data that we're not interested in, there's no need to check the year in the CASE expressions any more.

The interesting thing is that the optimiser will choose the same execution plan for both of these queries and so any performance gain is minimal. However, it's a useful principle when writing SQL statements to eliminate as much data as possible as early as possible – with the most selective WHERE clause. This gives the optimiser the best chance of choosing an efficient access path and reduces the resource requirements.

In many cases the difference between having logic in the WHERE clause instead of the SELECT clause will be dramatic because Oracle will be able to use an index to retrieve a smaller amount of data more quickly. The golden rule is

*Use the WHERE clause to eliminate all unnecessary data **first** and then use CASE in the SELECT list for additional processing.*

Beyond Equality

All of the examples so far have used simple equality tests. This is the limit of what the DECODE function can do. (There are workarounds to this using the SIGN, GREATEST or LEAST functions, for example – see the original DECODE paper for details).

However, CASE Expressions allows us to mix and match conditional tests on different combinations of columns, literal values and operators. For example, the sales manager might like to see the previous report modified so that the 10% markup is only applied to sales where the amount_sold is between 1000 and 2000

Example 3c

```
SELECT   p.prod_category, p.prod_subcategory,
   SUM(CASE WHEN p.prod_category = 'Photo' AND t.calendar_month_number = 10
            AND s.amount_sold BETWEEN 1000 AND 2000
            THEN amount_sold *1.1
       WHEN p.prod_category != 'Photo' AND t.calendar_month_number = 10
            THEN amount_sold
   END) AS OCT_dollars,
   SUM(CASE WHEN p.prod_category = 'Photo' AND t.calendar_month_number = 11
            AND s.amount_sold BETWEEN 1000 AND 2000
            THEN amount_sold *1.1
       WHEN p.prod_category != 'Photo' AND t.calendar_month_number = 11
            THEN amount_sold
   END) AS NOV_dollars,
   SUM(CASE WHEN p.prod_category = 'Photo' AND t.calendar_month_number = 12
            AND s.amount_sold BETWEEN 1000 AND 2000
            THEN amount_sold *1.1
       WHEN p.prod_category != 'Photo' AND t.calendar_month_number = 12
            THEN amount_sold
   END) AS DEC_dollars
FROM     sales s, times t, products p
WHERE s.time_id = t.time_id
AND s.prod_id = p.prod_id
```

```
AND t.calendar_year = 2001
AND t.calendar_month_number BETWEEN 10 AND 12
GROUP BY p.prod_category, p.prod_subcategory
ORDER BY 1, 2
/
```

PROD_CATEGORY	PROD_SUBCATEGORY	OCT_DOLLARS	NOV_DOLLARS	DEC_DOLLARS
Electronics	Game Consoles	105,876.22	88,989.00	193,546.85
	Home Audio	220,133.02	246,483.07	252,605.84
	Y Box Accessories	14,769.55	15,101.27	12,499.64
	Y Box Games	52,945.94	55,197.39	45,690.73
Hardware	Desktop PCs	153,857.16	239,435.84	180,910.18
	Portable PCs	192,513.25	201,927.44	205,868.81
Peripherals and Accessories	Accessories	65,193.15	53,754.82	69,484.05
	CD-ROM	63,934.53	65,239.50	37,439.03
	Memory	126,492.82	116,220.35	120,243.47
	Modems/Fax	85,488.22	75,540.22	82,163.87
	Monitors	338,795.95	309,261.00	318,457.99
	Printer Supplies	126,890.11	103,908.88	98,610.61
Photo	Camcorders	276,726.14	302,757.06	309,163.10
	Camera Batteries			
	Camera Media			
	Cameras	60,808.07	59,295.81	67,030.02
Software/Other	Accessories	60,090.11	46,264.68	50,513.14
	Bulk Pack Diskettes	8,816.24	6,497.00	7,066.86
	Documentation	63,202.96	67,330.54	78,742.43
	Operating Systems	89,284.76	106,532.85	96,790.97
	Recordable CDs	22,849.65	20,342.21	26,734.34
	Recordable DVD Discs	74,207.23	57,712.70	60,317.69

```
22 rows selected.
```

Hold on a minute. There's something wrong with the results for Camera Batteries and Camera Media. There aren't any. The problem here is that up until now I've been relying on the default value of NULL being returned if none of the conditions is true, so NULL will be added to the total, having no effect. (N.B this is subtly different behaviour to how NULL affects an addition operation, for example. A NULL value in a SUM operation will not force the result to be NULL, it will effectively be ignored.) However because the amount_sold for 'Camera Batteries' is not between 1000 and 2000 for any of the three months (so the first condition fails) but they are 'Photo' products (so the second condition fails) NULL will be added to the total repeatedly, with the end result of NULL. If what we really want to do is show a value of zero, then we need to add an ELSE clause to each of the CASE expressions, as follows.

```
SELECT   p.prod_category, p.prod_subcategory,
   SUM(CASE WHEN p.prod_category = 'Photo' AND t.calendar_month_number = 10
            AND s.amount_sold BETWEEN 1000 AND 2000
            THEN amount_sold *1.1
       WHEN p.prod_category != 'Photo' AND t.calendar_month_number = 10
            THEN amount_sold
       ELSE 0
   END) AS OCT_dollars,
   SUM(CASE WHEN p.prod_category = 'Photo' AND t.calendar_month_number = 11
            AND s.amount_sold BETWEEN 1000 AND 2000
```

```
            THEN amount_sold *1.1
       WHEN p.prod_category != 'Photo' AND t.calendar_month_number = 11
            THEN amount_sold
       ELSE 0
   END) AS NOV_dollars,
   SUM(CASE WHEN p.prod_category = 'Photo' AND t.calendar_month_number = 12
            AND s.amount_sold BETWEEN 1000 AND 2000
            THEN amount_sold *1.1
       WHEN p.prod_category != 'Photo' AND t.calendar_month_number = 12
            THEN amount_sold
       ELSE 0
   END) AS DEC_dollars
FROM     sales s, times t, products p
WHERE s.time_id = t.time_id
AND s.prod_id = p.prod_id
AND t.calendar_year = 2001
AND t.calendar_month_number BETWEEN 10 AND 12
GROUP BY p.prod_category, p.prod_subcategory
ORDER BY 1, 2
/
```

PROD_CATEGORY	PROD_SUBCATEGORY	OCT_DOLLARS	NOV_DOLLARS	DEC_DOLLARS
Electronics	Game Consoles	105,876.22	88,989.00	193,546.85
	Home Audio	220,133.02	246,483.07	252,605.84
	Y Box Accessories	14,769.55	15,101.27	12,499.64
	Y Box Games	52,945.94	55,197.39	45,690.73
Hardware	Desktop PCs	153,857.16	239,435.84	180,910.18
	Portable PCs	192,513.25	201,927.44	205,868.81
Peripherals and Accessories	Accessories	65,193.15	53,754.82	69,484.05
	CD-ROM	63,934.53	65,239.50	37,439.03
	Memory	126,492.82	116,220.35	120,243.47
	Modems/Fax	85,488.22	75,540.22	82,163.87
	Monitors	338,795.95	309,261.00	318,457.99
	Printer Supplies	126,890.11	103,908.88	98,610.61
Photo	Camcorders	276,726.14	302,757.06	309,163.10
	Camera Batteries	**0.00**	**0.00**	**0.00**
	Camera Media	**0.00**	**0.00**	**0.00**
	Cameras	60,808.07	59,295.81	67,030.02
Software/Other	Accessories	60,090.11	46,264.68	50,513.14
	Bulk Pack Diskettes	8,816.24	6,497.00	7,066.86
	Documentation	63,202.96	67,330.54	78,742.43
	Operating Systems	89,284.76	106,532.85	96,790.97
	Recordable CDs	22,849.65	20,342.21	26,734.34
	Recordable DVD Discs	74,207.23	57,712.70	60,317.69

```
22 rows selected.
```

That's better!

Conclusion

Although this paper only shows a few simple examples, it should be clear that CASE expressions are powerful tool when developing complex reports that perform efficiently. There are no practical limits on the complexity of the conditions you can test.

The Down side

Before we summarise the strengths of CASE expressions, let's focus on some of the potential weaknesses if we don't use them appropriately. Most of these are related to coding style and are therefore under our control. The first is that of code readability. Even a simple query such as Example 3c can be a little difficult to take in at first. The longer queries that you're likely to come across in business applications can 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 CASE expressions very clear.

The second, which I mentioned earlier, is that CASE is a post-retrieval function and it is easy to write code which is spectacularly inefficient but functionally correct. Remember the golden rule :-

*Use the WHERE clause to eliminate all unnecessary data **first** and then use CASE for additional processing.*

The Up side

CASE expressions give 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 the requirement and take a completely different approach to the task. Instead of limiting our tuning efforts to improving the speed of individual queries by investigating access paths and join methods, we can reduce the overall number of queries to retrieve all the data we require and then use CASE 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.

CASE 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 conditional processing to the data, row by row, based on the column values. We can often use CASE to perform that additional processing instead, using more efficient set-based SQL.

CASE works with all modern versions of Oracle and isn't dependent 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 algorithm more efficiently (although I'm sure this will happen in time). Like most performance tuning activities, the big improvements come from making smart decisions about your approach before you begin work.

Further Information

Oracle Documentation

Sample Schemas
http://download-west.oracle.com/docs/cd/B14117_01/server.101/b10771/toc.htm

CASE Expressions
http://download-west.oracle.com/docs/cd/B14117_01/server.101/b10759/expressions004.htm#sthref809

Related Articles

Sample Schemas article on OTN
http://www.oracle.com/technology/oramag/oracle/02-jul/o42schema.html

Original DECODE paper
http://doug.burns.tripod.com/decode.html

Daniel Morgan's DECODE and CASE reference
http://www.psoug.org/reference/decode_case.html

AskTom thread on deciphering execution plans
http://asktom.oracle.com/pls/ask/f?p=4950:8:10404375799506113809::NO::F4950_P8_DISPLAYID,F4950_P8_CRITERIA:231814117467,

Acknowledgements

I'd like to thank the following people for sparing some of their time to read through the paper, checking it for accuracy and making some very useful suggestions for improvement. As usual, the final decisions and all of the mistakes are mine and there would have been more if it wasn't for their efforts. Cheers, guys.

Andrew Campbell, Sun Microsystems

Colin Garside, BUPA

John Gilroy, Ask Jeeves

Jari Kuhanen, Sun Microsystems