SQL Server Update

August 1999

Using CASE Expressions

By Craig S. Mullins

CASE expressions are often over-looked but can be extremely useful to change very complex query requirements into simpler, and sometimes more efficient SQL statements. The CASE expression enables many forms of conditional processing to be placed into a SQL statement. By using CASE, more logic can be placed into SQL statements instead of being expressed in a host language or 4GL program.

Microsoft’s implementation of CASE expressions in SQL Server is compliant with the ANSI SQL-92 standard.

A Quick Example

Sometimes a quick example is worth several hundred descriptive words, so let’s take a look at a sample SQL statement using a CASE expression on the title table in the sample pubs database:

SQL Statement #1

```
SELECT title, price,
       Budget = CASE price
           WHEN price > 20.00 THEN 'Expensive'
           WHEN price BETWEEN 10.00 AND 19.99 THEN 'Moderate'
           WHEN price < 10.00 THEN 'Inexpensive'
           ELSE 'Unknown'
      END,
FROM titles
```

This statement would return results similar to these:

```
Title                               Price           Budget
------------------------------ ----------- ---------------
Cooking with Computers: Surrep 11.95       Moderate
Straight Talk About Computers  19.99       Moderate
The Busy Executive's Database  19.99       Moderate
You Can Combat Computer Stress 2.99        Inexpensive
Silicon Valley Gastronomic Tre 19.99       Moderate
The Gourmet Microwave          2.99        Inexpensive
But Is It User Friendly?       22.95       Expensive
Secrets of Silicon Valley      20.00       Moderate
Net Etiquette                  (null)      Unknown
```

This output is not comprehensive but shows enough detail to help describe the effects of the CASE expression. The CASE expression is exercised on each row returned. It categorizes the contents of the price column into four different groups: expensive, moderate, inexpensive, and unknown. This is easier and more efficient than writing a SQL statement that uses UNION to combine the results of the four categories. The following UNION statement would return equivalent results:

SQL Statement #2

```
SELECT title, price, 'Expensive'
FROM titles
WHERE price > 20.00
UNION ALL
SELECT title, price, 'Moderate'
FROM titles
WHERE price BETWEEN 10.00 AND 19.99 THEN 'Moderate'
UNION ALL
SELECT title, price, 'Inexpensive'
FROM titles
WHERE price < 10.00
UNION ALL
SELECT title, price, 'Unknown'
FROM titles
WHERE price IS NULL
go
```

UNION ALL is used instead of just UNION because no duplicates need to be removed. Each SELECT in the UNION returns a distinct result set.

You can see where this UNION formulation would be less efficient than the previous CASE formulation. In the UNION example SQL Server would have to make four passes through the data—one for each SELECT used. In the CASE example, one pass through the data is sufficient to return the correct results. Obviously, the CASE formulation will outperform the UNION formulation.

Another very useful capability of CASE expressions is to transform a result set from multiple rows into a summary row. Consider the situation where sales data is stored in a table by month. One row is stored per month with a table structure that looks like the following:

```
CREATE TABLE prodsales
 (product char(3),
  mnth    smallint,
  sales   money)
```

In this table, sales amounts (sales) are stored by month (mnth) and product code (product). The mnth column stores an integer value ranging from 1 (for January) to 12 (for December). You can use the following single SQL statement to product one row per product with 12 totals, one for each month:

SQL Statement #3

```
SELECT product,
  SUM(CASE mnth WHEN 1 THEN sales ELSE NULL END) AS jan,
  SUM(CASE mnth WHEN 2 THEN sales ELSE NULL END) AS feb,
  SUM(CASE mnth WHEN 3 THEN sales ELSE NULL END) AS mar,
  SUM(CASE mnth WHEN 4 THEN sales ELSE NULL END) AS apr,
  SUM(CASE mnth WHEN 5 THEN sales ELSE NULL END) AS may,
  SUM(CASE mnth WHEN 6 THEN sales ELSE NULL END) AS jun,
  SUM(CASE mnth WHEN 7 THEN sales ELSE NULL END) AS jul,
  SUM(CASE mnth WHEN 8 THEN sales ELSE NULL END) AS aug,
  SUM(CASE mnth WHEN 9 THEN sales ELSE NULL END) AS sep,
  SUM(CASE mnth WHEN 10 THEN sales ELSE NULL END) AS oct,
  SUM(CASE mnth WHEN 11 THEN sales ELSE NULL END) AS nov,
  SUM(CASE mnth WHEN 12 THEN sales ELSE NULL END) AS dec
FROM prodsales
GROUP BY product
```

This statement generates a row for each product with twelve monthly sales totals. The CASE expression causes the sales amount to by added to the appropriate bucket by checking the mnth column. If the month value is for the appropriate "month bucket", then the sales amount is added using SUM; if not, then NULL is specified, thereby avoiding adding anything to the SUM. Using CASE expressions in this manner simplifies aggregation and reporting. It provides a quick way of transforming normalized data structures into the more common denormalized formats that most business users are accustomed to viewing on reports.

This same basic idea can be used to create many types of summary rows. For example, to produce a summary row by quarter instead of by month, simply modify the CASE expressions as shown below:

SQL Statement #4

```
SELECT product,
  SUM(CASE WHEN mth BETWEEN 1 AND 3 THEN sales ELSE NULL END) AS q1,
  SUM(CASE WHEN mth BETWEEN 4 AND 6 THEN sales ELSE NULL END) AS q2,
  SUM(CASE WHEN mth BETWEEN 7 AND 9 THEN sales ELSE NULL END) AS q3,
  SUM(CASE WHEN mth BETWEEN 10 AND 12 THEN sales ELSE NULL END) AS q4
FROM prodsales
GROUP BY product
```

A More Complicated Example

Using searched CASE expressions and nested subqueries in SELECT statements very complex processing can be accomplished with a single SQL statement. Consider, once again, the sample pubs database. The following query checks the royalty percentage by title and places the percentage into a category based on its value:

SQL Statement #5

```
SELECT au_lname, au_fname, title, Category =
     CASE
      WHEN (SELECT AVG(royaltyper) FROM titleauthor ta
                 WHERE t.title_id = ta.title_id) > 65
           THEN 'Very High'
      WHEN (SELECT AVG(royaltyper) FROM titleauthor ta
                 WHERE t.title_id = ta.title_id)
                       BETWEEN 55 and 64
           THEN 'High'
      WHEN (SELECT AVG(royaltyper) FROM titleauthor ta
                 WHERE t.title_id = ta.title_id)
                       BETWEEN 41 and 54
           THEN 'Moderate'
      ELSE 'Low'
     END
FROM authors a,
     titles t,
     titleauthor ta
WHERE a.au_id = ta.au_id
AND   ta.title_id = t.title_id
ORDER BY Category, au_lname, au_fname
```

Within a SELECT statement, the searched CASE expression allows values to be replaced in the results set based on comparison values. In this example, the royalty percentage (royaltyper) in the titleauthor table is checked and a category is specified for each author based on the royalty percentage returned.

Usage and Syntax

CASE expressions can be used in SQL anywhere an expression can be used. This provides great flexibility because expressions can be used in a wide number of places. Example of where CASE expressions can be used include in the SELECT list, WHERE clauses, HAVING clauses, IN lists, DELETE and UPDATE statements, and inside of built-in functions.

There are two basic formulations that a CASE expression can take: simple CASE expressions and searched CASE expressions. A simple CASE expression checks one expression against multiple values. Within a SELECT statement, a simple CASE expression allows only an equality check; no other comparisons are made. A simple CASE expression operates by comparing the first expression to the expression in each WHEN clause for equivalency. If these expressions are equivalent, the expression in the THEN clause will be returned.

The basic syntax for a simple CASE expressions is shown below:

```
CASE expression
   WHEN expression1 THEN expression1
   [[WHEN expression2 THEN expression2] [...]]
   [ELSE expressionN]
END
```

A searched CASE expression is more feature-laden. It allows comparison operators, and the use of AND and/or OR between each Boolean expression. The simple CASE expression checks only for equivalent values and can not contain Boolean expressions. The basic syntax for a searched CASE expressions is shown below:

```
CASE
   WHEN Boolean_expression1 THEN expression1
   [[WHEN Boolean_expression2 THEN expression2] [...]]
   [ELSE expressionN]
END
```

You have seen samples of each type of CASE expression in the SQL examples depicted previously in this article. SQL statements #1 and #3 are simple CASE expressions; SQL statements #4 and #5 are searched CASE expressions.

Using CASE Expressions When Modifying Data

CASE expressions can also be used with data modification statements. Using CASE in conjunction with a SQL UPDATE statement enables developers to conditionally modify large amounts of data using a single SQL statement. Consider the following example:

SQL Statement #6

```
UPDATE titles
      SET price =
            CASE
                  WHEN (price < 5.0 AND ytd_sales > 999.99)
                        THEN price * 1.25
                  WHEN (price < 5.0 AND ytd_sales < 1000.00)
                        THEN price * 1.15
                  WHEN (price > 4.99 AND ytd_sales > 999.99)
                        THEN price * 1.2
                  ELSE price
            END
```

This statement examines book title criteria to determine whether prices should be modified. The CASE expression uses a combination of current price and year-to-date sales to specify a price increase percentage. Any criteria that can be expressed in terms of SQL predicates in CASE expressions can be used to update rows conditionally.

General Usage Guidelines

All data types used in the THEN clause of CASE expressions must be compatible data types. If the data type used is not compatible then SQL Server will return an error because implicit data type conversion is not supported.

Be sure that all possibilities are covered within the CASE expressions. For example, consider a CASE expression that must be coded on numeric data that can range from -1 to 100, and then an outlying value of 1000. Do not omit or overlap any of the possible data values. Furthermore, be aware of the data type of the values being examined in the CASE expression. Do not leave gaps in the range of possible values by not using appropriate values. For example, if the data is decimal (instead of integer, say) do not ignore the portion of the number to the right of the decimal point. A good example of this is shown in the previous SQL statement #6. If the value for 999.99 is changed to 999.0 then any values between 999.01 and 999.99 will be lumped in with the ELSE condition – which is clearly not the desired intent of this statement.

The bottom line is that CASE expressions are very powerful, yet often neglected. Favor using CASE expressions under the following circumstances:

- when data needs to be converted from one type to another for display and no function exists to accomplish the task
- when a summary row needs to be created from detail data

- when conditional processing needs to be executed to determine results
- when using UNION to “glue together” different subsets of a single table

One final usage guideline is to use the COALESCE function with your CASE expressions when you wish to avoid NULLs. Consider an employee table that contains three columns for wages: salary, commission, and retainer_fee. Any single employee can only receive one of these types of wages. So, two of the columns will contain NULLs, but one will always contain a value. The following statement uses the COALESCE function to return only the non-NULL value:

```
SELECT COALESCE(salary, commission, retainer_fee, 0)
FROM employee
```

The COALESCE function will go through the list of expressions (in this case columns and constants) and return the first non-NULL value encountered. In this case, the numeric constant 0 is added at the end of the list to provide a default value if all of the columns should happen to be NULL.

Summary

CASE expressions bring a vast degree of power and control to SQL Server programmers. A working knowledge of CASE expressions can make accessing and updating SQL Server data easier and perhaps, even more efficient. Additionally, CASE expressions enable more work to be accomplished using a single SQL statement, which should also improve efficiency and decrease development time. As such, CASE expressions should be a component of every SQL Server developer’s arsenal of programming techniques.