Subqueries In CASE Expressions

In this post, I'm going to take a look at how SQL Server handles subqueries in CASE expressions. I'll also introduce some more exotic join functionality in the process.

Scalar expressions

For simple CASE expressions with no subqueries, we can just evaluate the CASE expression as we would any other scalar expression:

```
create table T1 (a int, b int, c int)
```

```
|--Compute Scalar(DEFINE:([Expr1004]=CASE WHEN [T1].[a]>(0) THEN [T1].[b] ELSE
[T1].[c] END))
|--Table Scan(OBJECT:([T1]))
```

This query plan scans T1 and evaluates the CASE expression once for each row. The compute scalar operator computes the value of the CASE expression including evaluating the condition and deciding whether to evaluate the THEN clause or the ELSE clause.

When we introduce subqueries into the CASE expression, things get a bit more complex and a lot more interesting.

WHEN clause

Let's start by adding a simple subquery to the WHEN clause:

```
create table T2 (a int, b int)
```

```
T1.b
```

```
|--Compute Scalar(DEFINE:([Expr1009]=CASE WHEN [Expr1010] THEN [T1].[b] ELSE
[T1].[c] END))
|--Nested Loops(Left Semi Join, OUTER REFERENCES:([T1].[a]), DEFINE:([Expr1010] =
[PROBE VALUE]))
|--Table Scan(OBJECT:([T1]))
|--Table Scan(OBJECT:([T2]), WHERE:([T2].[a]=[T1].[a]))
```

As with other EXISTS subqueries, this query plan uses a left semi-join to test whether for each row in T1 we have a matching row in T2. However, a normal semi-join (or anti-semi-join) only returns rows for matches (or non-matches). In this case, we need to return something (either T1.b or T1.c) for every row in T1. We cannot simply discard a row of T1 just because there is no matching row in T2.

The solution is a special type of semi-join with a probe column. This semi-join returns all outer rows whether they match or not and sets the probe column (in this case [Expr1010]) to true or false to indicate whether or not it found a matching row of T1. Finally, we evaluate the CASE expression using the probe column to determine what value to return.

THEN clause

Now let’s try adding a simple subquery to the THEN clause:

create table T3 (a int unique clustered, b int)

insert T1 values(0, 0, 0)

insert T1 values(1, 1, 1)

I’ve added a unique constraint to T3 to guarantee that the scalar subquery returns only one row. Without the constraint, the query plan would be more complex as the optimizer would need to ensure that the subquery indeed returns only one row and would need to raise an error if it returned more than one row.

I’ve also added two rows to T1. The WHEN clause evaluates to false for the first row and to true for the second row. Thus, we will evaluate the ELSE clause for the first row and the THEN clause for the second row. Note that we cannot evaluate the THEN subquery unconditionally; we can only evaluate it if the WHEN clause is true.

Here is the statistics profile output for this plan:

Rows

Executes

|--Compute Scalar(DEFINE:([Expr1008]=CASE WHEN [T1].[a]>(0) THEN [T3].[b] ELSE [T1].[c] END))

|--Nested Loops(Left Outer Join, PASSTHRU:(IsFalseOrNull [T1].[a]>(0)), OUTER REFERENCES:([T1].[b]))

2

|--Table Scan(OBJECT:([T1]))

0

|--Clustered Index Seek(OBJECT:([T3].[UQ__T3__412EB0B6]), SEEK:([T3].[a]=[T1].[b]) ORDERED FORWARD)

This query plan uses a special type of nested loops join that includes a passthru predicate. The join evaluates the passthru predicate on each outer row. If the passthru predicate evaluates to true, the join returns the row immediately similar to a semi- or outer join. If the passthru predicate evaluates to false, the join proceeds normally and tries to join the outer row with an inner row.

In this example, the passthru predicate is the inverse (note the IsFalseOrNull function) of the WHEN clause in our CASE expression. If the WHEN clause evaluates to true, the passthru predicate evaluates to false, we execute the join, and the seek on the inner side of the join evaluates the THEN subquery. If the WHEN clause evaluates to false, the passthru predicates evaluates to true, we skip the join, and we do not execute the seek or the THEN subquery.

Note how the scan of T1 returns 2 rows yet we only execute the seek of T3 once. This is because in the example the WHEN clause is only true for one of the two rows. A passthru predicate is the only scenario where the number of rows on the outer side of a nested loops join does not precisely match the number of executes on the inner side.

Also note that we use an outer join since there is no guarantee that the THEN subquery will actually return any rows. (We do have a guarantee that it will return no more than one row thanks to the unique constraint.) If the subquery returns no rows, the outer join simply returns NULL for T3.b. If we used an inner join, we would incorrectly discard the row of T1.

Caution: I ran these examples on SQL Server 2005. If you run this example on SQL Server 2000, you will still get a passthru predicate, but it will appear in showplan as a regular where clause predicate. Unfortunately, on SQL Server 2000, there is no easy way using showplan to differentiate a regular predicate from a passthru predicate.

ELSE clause and multiple WHEN clauses

A subquery in the ELSE clause works the same way as a subquery in the THEN clause. We use a passthru predicate to evaluate the subquery conditionally.

Similarly, a CASE expression with multiple WHEN clauses with subqueries in each THEN clause also works the same way. The passthru predicates just get progressively more complex.

For example:

create table T4 (a int unique clustered, b int)

create table T5 (a int unique clustered, b int)

select

when T1.a > 0 then

(select T3.b from T3 where T3.a = T1.a)

when T1.b > 0 then

(select T4.b from T4 where T4.a = T1.b)

(select T5.b from T5 where T5.a = T1.c)

```
|--Compute Scalar(DEFINE:([Expr1016]=CASE WHEN [T1].[a]>(0) THEN [T3].[b] ELSE
CASE WHEN [T1].[b]>(0) THEN [T4].[b] ELSE [T5].[b] END END))
|--Nested Loops(Left Outer Join, PASSTHRU:([T1].[a]>(0) OR [T1].[b]>(0)), OUTER
REFERENCES:([T1].[c]))
|--Nested Loops(Left Outer Join, PASSTHRU:([T1].[a]>(0) OR IsFalseOrNull [T1].[b]>(0)),
OUTER REFERENCES:([T1].[b]))
| |--Nested Loops(Left Outer Join, PASSTHRU:(IsFalseOrNull [T1].[a]>(0)), OUTER
REFERENCES:([T1].[a]))
| | |--Table Scan(OBJECT:([T1]))
| | |--Clustered Index Seek(OBJECT:([T3].[UQ__T3__164452B1]), SEEK:([T3].[a]=[T1].[a])
ORDERED FORWARD)
| |--Clustered Index Seek(OBJECT:([T4].[UQ__T4__182C9B23]), SEEK:([T4].[a]=[T1].[b])
ORDERED FORWARD)
|--Clustered Index Seek(OBJECT:([T5]. Gaming [UQ__T5__1A14E395]),
SEEK:([T5].[a]=[T1].[c]) ORDERED FORWARD)
```

This query plan has three nested loops joins with passthru predicates. For each row of T1, only one of the three passthru predicates evaluates to true and only one of the three subqueries is executed. Note that while the second WHEN clause is “T1.b > 0,” there is an implication that the first WHEN clause, “T1.a > 0,” is false. This is true for the ELSE clause as well. Thus, the passthru predicates for the second and third subqueries include the check “T1.a > 0 OR …”.

Probe column as the passthru predicate

Finally, let’s try a query with subqueries in both the WHEN and the THEN clauses. Also, just to make things a bit more interesting, I’m also going to move the CASE expression from the SELECT list to the WHERE clause.

select *

from T1

where 0 =

```
case

when exists (select * from T2 where T2.a = T1.a) then

(select T3.b from T3 where T3.a = T1.b)

else

T1.c

end
```

```
|--Filter(WHERE:((0)=CASE WHEN [Expr1013] THEN [T3].[b] ELSE [T1].[c] END))
|--Nested Loops(Left Outer Join, PASSTHRU:(IsFalseOrNull [Expr1013]), OUTER
REFERENCES:([T1].[b]))
|--Nested Loops(Left Semi Join, OUTER REFERENCES:([T1].[a]), DEFINE:([Expr1013] =
[PROBE VALUE]))
| |--Table Scan(OBJECT:([T1]))
| |--Table Scan(OBJECT:([T2]), WHERE:([T2].[a]=[T1].[a]))
|--Clustered Index Seek(OBJECT:([T3].[UQ__T3__164452B1]), SEEK:([T3].[a]=[T1].[b])
ORDERED FORWARD)
```

This query plan has a left semi join with a probe column to evaluate the subquery in the WHEN clause and a nested loops join with a passthru predicate on the probe column to determine whether to evaluate the subquery in the THEN clause. Because we moved the CASE expression to the WHERE clause, instead of a compute scalar operator to evaluate an output value for the SELECT list, we have a filter operator to determine whether each row should be returned. Everything else is exactly the same.