Microsoft

T-SQL Fundamentals

Third Edition

Professional

Itzik Ben-Gan

SolidQ

Microsoft

T-SQL Fundamentals, Third Edition

Itzik Ben-Gan

```
56          Customer QNIVZ  10833
56          Customer QNIVZ  10999
56          Customer QNIVZ  11020
57          Customer WVAXS  NULL
58          Customer AHXHT  10322
58          Customer AHXHT  10354
58          Customer AHXHT  10474
58          Customer AHXHT  10502
58          Customer AHXHT  10995
...
91          Customer CCFIZ  10792
91          Customer CCFIZ  10870
91          Customer CCFIZ  10906
91          Customer CCFIZ  10998
91          Customer CCFIZ  11044

(832 row(s) affected)
```

Two customers in the *Customers* table did not place any orders. Their IDs are 22 and 57. Observe that in the output of the query, both customers are returned with *NULLs* in the attributes from the *Orders* table. Logically, the rows for these two customers were discarded by the second phase of the join (the filter based on the *ON* predicate), but the third phase added those as outer rows. Had the join been an inner join, these two rows would not have been returned. These two rows are added to preserve all the rows of the left table.

It might help to think of the result of an outer join as having two kinds of rows with respect to the preserved side—inner rows and outer rows. Inner rows are rows that have matches on the other side based on the *ON* predicate, and outer rows are rows that don't. An inner join returns only inner rows, whereas an outer join returns both inner and outer rows.

A common question about outer joins that is the source of a lot of confusion is whether to specify a predicate in the *ON* or *WHERE* clause of a query. You can see that with respect to rows from the preserved side of an outer join, the filter based on the *ON* predicate is not final. In other words, the *ON* predicate does not determine whether a row will show up in the output, only whether it will be matched with rows from the other side. So when you need to express a predicate that is not final—meaning a predicate that determines which rows to match from the nonpreserved side—specify the predicate in the *ON* clause. When you need a filter to be applied after outer rows are produced, and you want the filter to be final, specify the predicate in the *WHERE* clause. The *WHERE* clause is processed after the *FROM* clause—specifically, after all table operators have been processed and (in the case of outer joins) after all outer rows have been produced. Also, the *WHERE* clause is final with respect to rows that it filters out, unlike the *ON* clause. To recap, in the *ON* clause you specify nonfinal, or matching, predicates. In the *WHERE* clause you specify final, or filtering, predicates.

Suppose you need to return only customers who did not place any orders or, more technically speaking, you need to return only outer rows. You can use the previous query as your basis, adding a *WHERE* clause that filters only outer rows. Remember that outer rows are identified by the *NULLs* in the attributes from the nonpreserved side of the join. So you can filter only the rows in which one of the attributes on the nonpreserved side of the join is *NULL*, like this:

```
SELECT C.custid, C.companyname
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
WHERE O.orderid IS NULL;
```

This query returns only two rows, with the customers 22 and 57:

```
custid      companyname
----------- ---------------
22          Customer DTDMN
57          Customer WVAXS

(2 row(s) affected)
```

Notice a couple of important things about this query. Recall the discussions about *NULLs* earlier in the book: When looking for a *NULL*, you should use the operator *IS NULL* and not an equality operator. You do this because when an equality operator compares something with a *NULL*, it always returns *UNKNOWN*—even when it's comparing two *NULLs*. Also, the choice of which attribute from the nonpreserved side of the join to filter is important. You should choose an attribute that can have only a *NULL* when the row is an outer row and not otherwise (for example, not a *NULL* originating from the base table). For this purpose, three cases are safe to consider: a primary key column, a join column, and a column defined as *NOT NULL*. A primary key column cannot be *NULL*; therefore, a *NULL* in such a column can only mean that the row is an outer row. If a row has a *NULL* in the join column, that row is filtered out by the second phase of the join, so a *NULL* in such a column can only mean that it's an outer row. And obviously, a *NULL* in a column that is defined as *NOT NULL* can only mean that the row is an outer row.

To practice what you learned and get a better grasp of outer joins, make sure you perform the exercises for this chapter.

Beyond the fundamentals of outer joins

This section covers more advanced aspects of outer joins and is provided as optional reading for when you feel comfortable with the fundamentals of outer joins.

Including missing values

You can use outer joins to identify and include missing values when querying data. For example, suppose you need to query all orders from the *Orders* table in the *TSQLV4* database. You need to ensure that you get at least one row in the output for each date in the range January 1, 2014 through December 31, 2016. You don't want to do anything special with dates within the range that have orders, but you do want the output to include the dates with no orders, with *NULLs* as placeholders in the attributes of the order.

To solve the problem, you can first write a query that returns a sequence of all dates in the requested period. You can then perform a left outer join between that set and the *Orders* table. This way, the result also includes the missing dates.

To produce a sequence of dates in a given range, I usually use an auxiliary table of numbers. I create a table called *dbo.Nums* with a column called *n*, and populate it with a sequence of integers (1, 2, 3, and so on). I find that an auxiliary table of numbers is an extremely powerful general-purpose tool I end up using to solve many problems. You need to create it only once in the database and populate it with as many numbers as you might need. The *TSQLV4* sample database already has such an auxiliary table.

As the first step in the solution, you need to produce a sequence of all dates in the requested range. You can achieve this by querying the *Nums* table and filtering as many numbers as the number of days in the requested date range. You can use the *DATEDIFF* function to calculate that number. By adding *n – 1* days to the starting point of the date range (January 1, 2014), you get the actual date in the sequence. Here's the solution query:

```
SELECT DATEADD(day, n-1, CAST('20140101' AS DATE)) AS orderdate
FROM dbo.Nums
WHERE n <= DATEDIFF(day, '20140101', '20161231') + 1
ORDER BY orderdate;
```

This query returns a sequence of all dates in the range January 1, 2014 through December 31, 2016, as shown here in abbreviated form:

```
orderdate
-----------
2014-01-01
2014-01-02
2014-01-03
2014-01-04
2014-01-05
...
2016-12-27
2016-12-28
2016-12-29
2016-12-30
2016-12-31

(1096 row(s) affected)
```

The next step is to extend the previous query, adding a left outer join between *Nums* and the *Orders* tables. The join condition compares the order date produced from the *Nums* table and the *orderdate* from the *Orders* table by using the expression *DATEADD(day, Nums.n – 1, CAST('20140101' AS DATE))* like this:

```
SELECT DATEADD(day, Nums.n - 1, CAST('20140101' AS DATE)) AS orderdate,
  O.orderid, O.custid, O.empid
FROM dbo.Nums
  LEFT OUTER JOIN Sales.Orders AS O
    ON DATEADD(day, Nums.n - 1, CAST('20140101' AS DATE)) = O.orderdate
WHERE Nums.n <= DATEDIFF(day, '20140101', '20161231') + 1
ORDER BY orderdate;
```

This query produces the following output, shown here in abbreviated form:

```
orderdate   orderid     custid      empid
----------- ----------- ----------- -----------
2014-01-01  NULL        NULL        NULL
2014-01-02  NULL        NULL        NULL
2014-01-03  NULL        NULL        NULL
2014-01-04  NULL        NULL        NULL
2014-01-05  NULL        NULL        NULL
...
2014-06-29  NULL        NULL        NULL
2014-06-30  NULL        NULL        NULL
2014-07-01  NULL        NULL        NULL
2014-07-02  NULL        NULL        NULL
2014-07-03  NULL        NULL        NULL
2014-07-04  10248       85          5
2014-07-05  10249       79          6
2014-07-06  NULL        NULL        NULL
2014-07-07  NULL        NULL        NULL
2014-07-08  10250       34          4
2014-07-08  10251       84          3
2014-07-09  10252       76          4
2014-07-10  10253       34          3
2014-07-11  10254       14          5
2014-07-12  10255       68          9
2014-07-13  NULL        NULL        NULL
2014-07-14  NULL        NULL        NULL
2014-07-15  10256       88          3
2014-07-16  10257       35          4
...
2008-12-2   NULL        NULL        NULL
2008-12-2   NULL        NULL        NULL
2008-12-2   NULL        NULL        NULL
2008-12-3   NULL        NULL        NULL
2008-12-3   NULL        NULL        NULL

(1446 row(s) affected)
```

Dates that do not appear as order dates in the *Orders* table appear in the output of the query with *NULLs* in the order attributes.

Filtering attributes from the nonpreserved side of an outer join

When you need to review code involving outer joins to look for logical bugs, one of the things you should examine is the *WHERE* clause. If the predicate in the *WHERE* clause refers to an attribute from the nonpreserved side of the join using an expression in the form <*attribute*> <*operator*> <*value*>, it's usually an indication of a bug. This is because attributes from the nonpreserved side of the join are *NULLs* in outer rows, and an expression in the form *NULL* <*operator*> <*value*> yields *UNKNOWN* (unless it's the *IS NULL* operator explicitly looking for *NULLs*). Recall that a *WHERE* clause filters *UNKNOWN* out. Such a predicate in the *WHERE* clause causes all outer rows to be filtered out, effectively nullifying the outer join. Effectively, the join becomes an inner join. So the programmer either made a mistake in the join type or in the predicate.

If this is not clear yet, the following example might help. Consider the following query:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
WHERE O.orderdate >= '20160101';
```

The query performs a left outer join between the *Customers* and *Orders* tables. Prior to applying the *WHERE* filter, the join operator returns inner rows for customers who placed orders and outer rows for customers who didn't place orders, with *NULLs* in the order attributes. The predicate *O.orderdate >= '20160101'* in the *WHERE* clause evaluates to *UNKNOWN* for all outer rows, because those have a *NULL* in the *O.orderdate* attribute. All outer rows are eliminated by the *WHERE* filter, as you can see in the output of the query, shown here in abbreviated form:

```
custid      companyname     orderid     orderdate
----------- --------------- ----------- ----------
1           Customer NRZBB  10835       2016-01-15
1           Customer NRZBB  10952       2016-03-16
1           Customer NRZBB  11011       2016-04-09
2           Customer MLTDN  10926       2016-03-04
3           Customer KBUDE  10856       2016-01-28
...
90          Customer XBBVR  10910       2016-02-26
91          Customer CCFIZ  10906       2016-02-25
91          Customer CCFIZ  10870       2016-02-04
91          Customer CCFIZ  10998       2016-04-03
91          Customer CCFIZ  11044       2016-04-23

(270 row(s) affected)
```

This means that the use of an outer join here was futile. The programmer either made a mistake in using an outer join or in specifying the predicate in the *WHERE* clause.

Using outer joins in a multi-join query

Recall the discussion about all-at-once operations in Chapter 2, "Single-table queries." The concept describes the fact that all expressions that appear in the same logical query processing phase are evaluated as a set, at the same point in time. However, this concept is not applicable to the processing of table operators in the *FROM* phase. Table operators are logically evaluated from left to right. Rearranging the order in which outer joins are processed might result in different output, so you cannot rearrange them at will.

Some interesting bugs have to do with the logical order in which outer joins are processed. For example, a common bug could be considered a variation of the bug in the previous section. Suppose you write a multi-join query with an outer join between two tables, followed by an inner join with a third table. If the predicate in the inner join's *ON* clause compares an attribute from the nonpreserved side of the outer join and an attribute from the third table, all outer rows are discarded. Remember that outer rows have *NULLs* in the attributes from the nonpreserved side of the join, and comparing a *NULL* with anything yields *UNKNOWN*. *UNKNOWN* is filtered out by the *ON* filter. In other words, such

a predicate nullifies the outer join, effectively turning it into an inner join. For example, consider the following query:

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
  INNER JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid;
```

The first join is an outer join returning customers and their orders and also customers who did not place any orders. The outer rows representing customers with no orders have *NULLs* in the order attributes. The second join matches order lines from the *OrderDetails* table with rows from the result of the first join, based on the predicate *O.orderid = OD.orderid*; however, in the rows representing customers with no orders, the *O.orderid* attribute is *NULL*. Therefore, the predicate evaluates to *UNKNOWN,* and those rows are discarded. The output shown here in abbreviated form doesn't contain the customers 22 and 57, the two customers who did not place orders:

```
custid      orderid     productid   qty
----------- ----------- ----------- ------
85          10248       11          12
85          10248       42          10
85          10248       72          5
79          10249       14          9
79          10249       51          40
...
65          11077       64          2
65          11077       66          1
65          11077       73          2
65          11077       75          4
65          11077       77          2

(2155 row(s) affected)
```

Generally, outer rows are dropped whenever any kind of outer join (left, right, or full) is followed by a subsequent inner join or right outer join. That's assuming, of course, that the join condition compares the *NULLs* from the left side with something from the right side.

There are several ways to get around the problem if you want to return customers with no orders in the output. One option is to use a left outer join in the second join as well:

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
  LEFT OUTER JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid;
```

This way, the outer rows produced by the first join aren't filtered out, as you can see in the output shown here in abbreviated form: