Microsoft

T-SQL Fundamentals

Fourth Edition

Professional

Itzik Ben-Gan

Microsoft

T-SQL Fundamentals

Itzik Ben-Gan

Note SQL supports other predicates that operate on a multivalued subquery; those are *SOME*, *ANY*, and *ALL*. They are rarely used and therefore are not covered in this book.

The form of the *IN* predicate is

<scalar_expression> IN (<multivalued subquery>)

The predicate evaluates to *TRUE* if *scalar_expression* is equal to any of the values returned by the subquery. Recall the request in the previous section—returning orders that were handled by employees with a last name starting with a certain letter. Because more than one employee can have a last name starting with the same letter, this request should be handled with the *IN* predicate and not with an equality operator. For example, the following query returns orders placed by employees with a last name starting with *D*:

```
SELECT orderid
FROM Sales.Orders
WHERE empid IN
  (SELECT E.empid
   FROM HR.Employees AS E
   WHERE E.lastname LIKE N'D%');
```

Because this solution uses the *IN* predicate, this query is valid with any number of values returned—none, one, or more. This query returns the following output, shown here in abbreviated form:

```
orderid
-----------
10258
10270
10275
10285
10292
...
10978
11016
11017
11022
11058

(166 rows affected)
```

You might wonder why you don't implement this task by using a join instead of subqueries, like this:

```
SELECT O.orderid
FROM HR.Employees AS E
  INNER JOIN Sales.Orders AS O
    ON E.empid = O.empid
WHERE E.lastname LIKE N'D%';
```

Similarly, you're likely to stumble into many other querying tasks you can solve with either subqueries or joins. I don't know of a reliable rule of thumb that says a subquery is better than a join or the other way around. In some cases the database engine optimizes both the same way, sometimes joins perform better, and sometimes subqueries perform better. My approach is to first write a solution query that is intuitive and then, if performance is not satisfactory, try query revisions among other tuning methods. Such query revisions might include using joins instead of subqueries or the other way around. Also, consider keeping the different query rewrites; future changes in the database engine or the data might result in a different query being faster.

As another example of using multivalued subqueries, suppose you need to write a query that returns orders placed by customers from the United States. You can write a query against the *Orders* table that returns orders where the customer ID is in the set of customer IDs of customers from the United States. You can implement the last part in a self-contained, multivalued subquery. Here's the complete solution query:

```
SELECT custid, orderid, orderdate, empid
FROM Sales.Orders
WHERE custid IN
  (SELECT C.custid
   FROM Sales.Customers AS C
   WHERE C.country = N'USA');
```

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

```
custid      orderid     orderdate   empid
----------- ----------- ----------- -----------
65          10262       2020-07-22  8
89          10269       2020-07-31  5
75          10271       2020-08-01  6
65          10272       2020-08-02  6
65          10294       2020-08-30  4
...
32          11040       2022-04-22  4
32          11061       2022-04-30  4
71          11064       2022-05-01  1
89          11066       2022-05-01  7
65          11077       2022-05-06  1

(122 rows affected)
```

As with any other predicate, you can negate the *IN* predicate with the *NOT* operator. For example, the following query returns customers who did not place any orders:

```
SELECT custid, companyname
FROM Sales.Customers
WHERE custid NOT IN
  (SELECT O.custid
   FROM Sales.Orders AS O);
```

Note that it's considered a best practice to qualify the subquery, namely to add a filter, to exclude *NULLs*. I didn't do this here because I didn't explain the reason for this recommendation. I'll explain it later, in the section "*NULL* trouble."

The subquery returns the IDs of all customers that appear in the *Orders* table. In other words, it returns only the IDs of customers who placed orders. The outer query returns customers with IDs that do not appear in the result of the subquery—in other words, customers who did not place orders. This query returns the following output:

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

You might wonder whether specifying a *DISTINCT* clause in the subquery can help performance, because the same customer ID can occur more than once in the *Orders* table. The database engine is smart enough to consider removing duplicates without you asking it to do so explicitly, so this isn't something you need to worry about.

The last example in this section demonstrates the use of multiple self-contained subqueries in the same query—both single-valued and multivalued. Before I describe the task, run the following code to create a table called *dbo.Orders* in the *TSQLV6* database, and populate it with even-numbered order IDs from the *Sales.Orders* table:

```
USE TSQLV6;
DROP TABLE IF EXISTS dbo.Orders;
CREATE TABLE dbo.Orders(orderid INT NOT NULL CONSTRAINT PK_Orders PRIMARY KEY);

INSERT INTO dbo.Orders(orderid)
  SELECT orderid
  FROM Sales.Orders
  WHERE orderid % 2 = 0;
```

I describe the *INSERT* statement in more detail in Chapter 8, "Data modification," so don't worry if you're not familiar with it yet.

You need to write a query that returns all individual order IDs that are missing between the minimum and maximum ones in the table. It can be quite complicated to solve this task with a query without any helper tables or functions. You might find the *Nums* table introduced in Chapter 3, "Joins," useful here. Remember that the *Nums* table contains a sequence of integers, starting with 1, with no gaps. Recall that starting with SQL Server 2022, you can alternatively use the *GENERATE_SERIES* function described in Chapter 2 to generate a sequence of numbers. But in this example I'll use the *Nums* table. To return all missing order IDs, query the *Nums* table and filter only numbers that are between the minimum and maximum ones in the *dbo.Orders* table, and that do not appear as order IDs in the *Orders* table. You can use scalar self-contained subqueries to return the minimum and maximum order IDs and a multivalued self-contained subquery to return the set of all existing order IDs. Here's the complete solution query:

```
SELECT n
FROM dbo.Nums
WHERE n BETWEEN (SELECT MIN(O.orderid) FROM dbo.Orders AS O)
            AND (SELECT MAX(O.orderid) FROM dbo.Orders AS O)
  AND n NOT IN (SELECT O.orderid FROM dbo.Orders AS O);
```

Because the code that populated the *dbo.Orders* table filtered only even-numbered order IDs, this query returns all odd-numbered values between the minimum and maximum order IDs in the *Orders* table. The output of this query is shown here in abbreviated form:

```
n
-----------
10249
10251
10253
10255
10257
...
11067
11069
11071
11073
11075

(414 rows affected)
```

When you're done, run the following code for cleanup:

```
DROP TABLE IF EXISTS dbo.Orders;
```

Correlated subqueries

Correlated subqueries are subqueries that refer to attributes from the tables that appear in the outer query. This means the subquery is dependent on the outer query and cannot be invoked as a standalone query. Logically, the subquery is evaluated separately for each outer row in the logical query processing step in which it appears. For example, the query in Listing 4-1 returns orders with the maximum order ID for each customer.

LISTING 4-1 Correlated subquery

```
USE TSQLV6;

SELECT custid, orderid, orderdate, empid
FROM Sales.Orders AS O1
WHERE orderid =
  (SELECT MAX(O2.orderid)
   FROM Sales.Orders AS O2
   WHERE O2.custid = O1.custid);
```

The outer query is issued against an instance of the *Orders* table called *O1*; it filters orders where the order ID is equal to the value returned by the subquery. The subquery filters orders from a second instance of the *Orders* table called *O2*, where the inner customer ID is equal to the outer customer ID, and returns the maximum order ID from those filtered orders. In other words, for each row in *O1*, the subquery returns the maximum order ID for the current customer. If the outer order ID and the order

ID returned by the subquery match, the query returns the outer row. This query returns the following output, shown here in abbreviated form:

```
custid      orderid     orderdate  empid
----------- ----------- ---------- -----------
91          11044       2022-04-23 4
90          11005       2022-04-07 2
89          11066       2022-05-01 7
88          10935       2022-03-09 4
87          11025       2022-04-15 6
...
5           10924       2022-03-04 3
4           11016       2022-04-10 9
3           10856       2022-01-28 3
2           10926       2022-03-04 4
1           11011       2022-04-09 3

(89 rows affected)
```

Because of the dependency on the outer query, correlated subqueries are usually harder to figure out than self-contained subqueries. To simplify things, I suggest you focus your attention on a single row in the outer table and think about the logical processing that takes place in the inner query for that row. For example, focus your attention on the following row from the table in the outer query, which has order ID 10248:

```
custid      orderid     orderdate                   empid
----------- ----------- --------------------------- -----------
85          10248       2020-07-04 00:00:00.000     5
```

When the subquery is evaluated for this row, the correlation to *O1.custid* means *85*. If you substitute the correlation manually with *85*, you get the following:

```
SELECT MAX(O2.orderid)
FROM Sales.Orders AS O2
WHERE O2.custid = 85;
```

This query returns the order ID 10739. The outer row's order ID—10248—is compared with the inner one—10739—and because there's no match in this case, the outer row is filtered out. The subquery returns the same value for all rows in *O1* with the same customer ID, and only in one case is there a match—when the outer row's order ID is the maximum for the current customer. Thinking in such terms will make it easier for you to grasp the concept of correlated subqueries.

The fact that correlated subqueries are dependent on the outer query makes it harder to troubleshoot problems with them compared to self-contained subqueries. You can't just highlight the subquery portion and run it. For example, if you try to highlight and run the subquery portion in Listing 4-1, you get the following error:

```
Msg 4104, Level 16, State 1, Line 119
The multi-part identifier "O1.custid" could not be bound.
```

This error indicates that the identifier *O1.custid* cannot be bound to an object in the query, because *O1* is not defined in the query. It is defined only in the context of the outer query. To troubleshoot correlated subqueries, you need to substitute the correlation with a constant, and after ensuring the code is correct, substitute the constant with the correlation.

As another example, suppose you need to query the *Sales.OrderValues* view and return for each order the percentage of the current order value out of the customer total. In Chapter 7, "T-SQL for data analysis," I provide a solution to this task that uses window functions; here I'll explain how to solve the task by using subqueries. It's always a good idea to try to come up with several solutions to each task, because the different solutions will usually vary in complexity and performance.

You can write an outer query against one instance of the *OrderValues* view called *O1*. In the *SELECT* list, divide the current value by the result of a correlated subquery against a second instance of *OrderValues* called *O2* that returns the current customer's total. Here's the complete solution query:

```
SELECT orderid, custid, val,
  CAST(100. * val / (SELECT SUM(O2.val)
                     FROM Sales.OrderValues AS O2
                     WHERE O2.custid = O1.custid)
       AS NUMERIC(5,2)) AS pct
FROM Sales.OrderValues AS O1
ORDER BY custid, orderid;
```

The *CAST* function is used to convert the datatype of the expression to *NUMERIC* with a precision of 5 (the total number of digits) and a scale of 2 (the number of digits after the decimal point).

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

```
orderid     custid      val        pct
----------- ----------- ---------- ------
10643       1           814.50     19.06
10692       1           878.00     20.55
10702       1           330.00     7.72
10835       1           845.80     19.79
10952       1           471.20     11.03
11011       1           933.50     21.85
10308       2           88.80      6.33
10625       2           479.75     34.20
10759       2           320.00     22.81
10926       2           514.40     36.67
...

(830 rows affected)
```