Chapter 12
Advanced Querying Tools

Most of the querying tools we've discussed so far in this book are what you should expect to see and use most frequently. You'll probably get by just fine with the things you've learned up to this point, but if you want to become a querying *master* and impress your coworkers (and most importantly your boss), you'll want to learn and understand the more *uncommon* querying tools that are available to us.

Who knows, some of the tools you'll learn in this chapter may be exactly what you need to know to accomplish a given task.

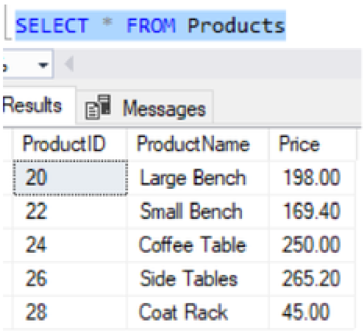
The APPLY Operator

The APPLY operator works very similarly to a correlated subquery, which we discussed in chapter 10. It involves using a left input and a right input, where the right input can reference elements from the left input.

The best way to understand the APPLY operator is to look at some examples. There are two versions of the APPLY operator: CROSS and OUTER.

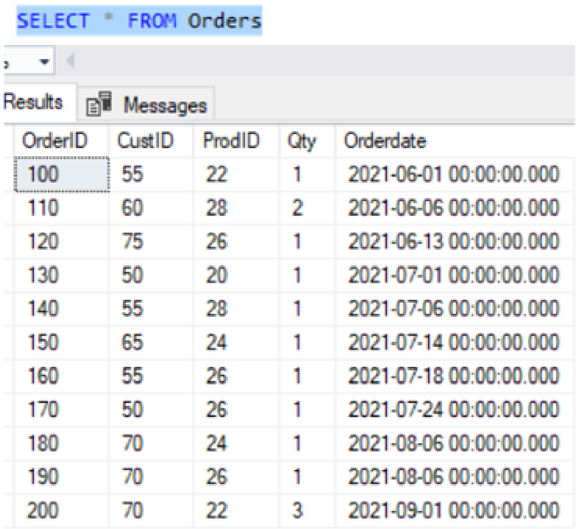
CROSS APPLY

Take a look at the Products and Orders tables:



```sql
SELECT * FROM Products
```

ProductID	ProductName	Price
20	Large Bench	198.00
22	Small Bench	169.40
24	Coffee Table	250.00
26	Side Tables	265.20
28	Coat Rack	45.00

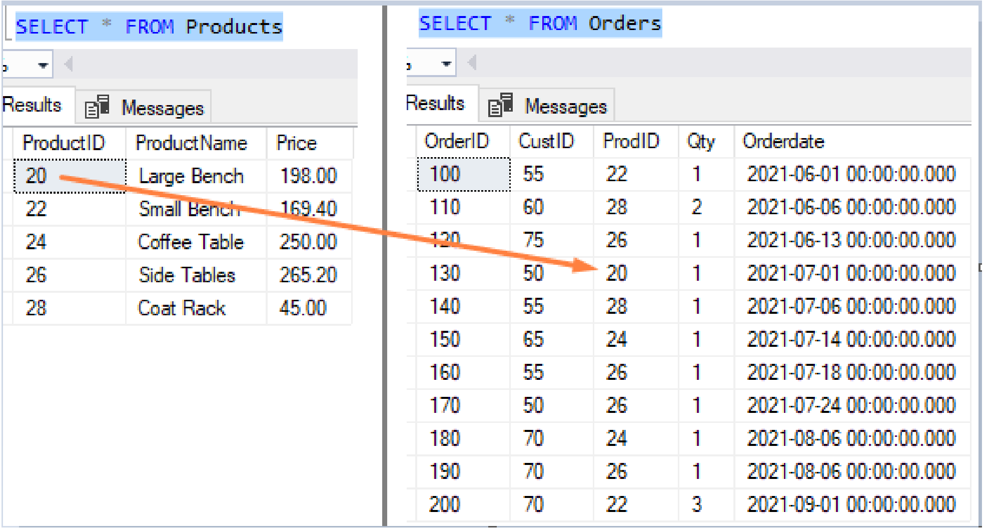


```sql
SELECT * FROM Orders
```

OrderID	CustID	ProdID	Qty	Orderdate
100	55	22	1	2021-06-01 00:00:00.000
110	60	28	2	2021-06-06 00:00:00.000
120	75	26	1	2021-06-13 00:00:00.000
130	50	20	1	2021-07-01 00:00:00.000
140	55	28	1	2021-07-06 00:00:00.000
150	65	24	1	2021-07-14 00:00:00.000
160	55	26	1	2021-07-18 00:00:00.000
170	50	26	1	2021-07-24 00:00:00.000
180	70	24	1	2021-08-06 00:00:00.000
190	70	26	1	2021-08-06 00:00:00.000
200	70	22	3	2021-09-01 00:00:00.000

Let's say we wanted to create a result set where for each product, we displayed all the orders that have been made for that product. We'll return all columns from the Products table and just the OrderID and Orderdate columns of the corresponding rows in the Orders table.

Let's walk down each row from the left input (the Products table) like SQL Server will do. The first product is the Large Bench, Product ID # **20**. Which rows in the right table reference that product? Looks like it's just this one:



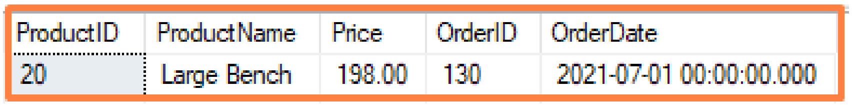
```
SELECT * FROM Products
```

ProductID	ProductName	Price
20	Large Bench	198.00
22	Small Bench	169.40
24	Coffee Table	250.00
26	Side Tables	265.20
28	Coat Rack	45.00

```
SELECT * FROM Orders
```

OrderID	CustID	ProdID	Qty	Orderdate
100	55	22	1	2021-06-01 00:00:00.000
110	60	28	2	2021-06-06 00:00:00.000
120	75	26	1	2021-06-13 00:00:00.000
130	50	20	1	2021-07-01 00:00:00.000
140	55	28	1	2021-07-06 00:00:00.000
150	65	24	1	2021-07-14 00:00:00.000
160	55	26	1	2021-07-18 00:00:00.000
170	50	26	1	2021-07-24 00:00:00.000
180	70	24	1	2021-08-06 00:00:00.000
190	70	26	1	2021-08-06 00:00:00.000
200	70	22	3	2021-09-01 00:00:00.000

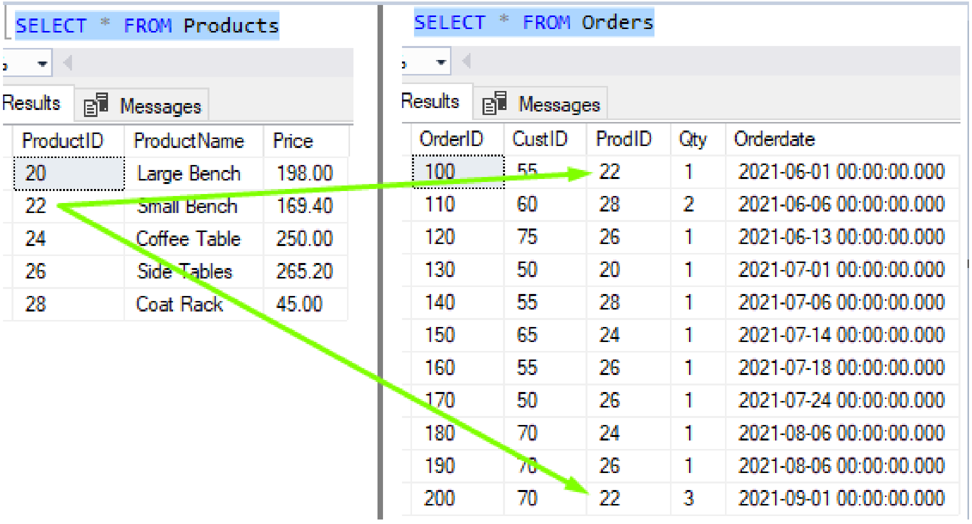
So in our final result set (so far), we'll see a row like this:



ProductID	ProductName	Price	OrderID	OrderDate
20	Large Bench	198.00	130	2021-07-01 00:00:00.000

(I'm purposefully not showing you the query yet. We'll get to it soon)

That takes care of that product. Then we move onto the next product, which is the Small Bench. The rows that correlate to it are the following:



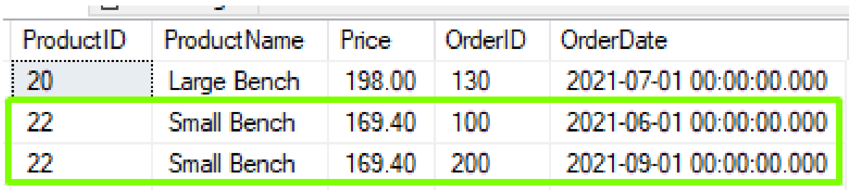
```
SELECT * FROM Products
```

ProductID	ProductName	Price
20	Large Bench	198.00
22	Small Bench	169.40
24	Coffee Table	250.00
26	Side Tables	265.20
28	Coat Rack	45.00

```
SELECT * FROM Orders
```

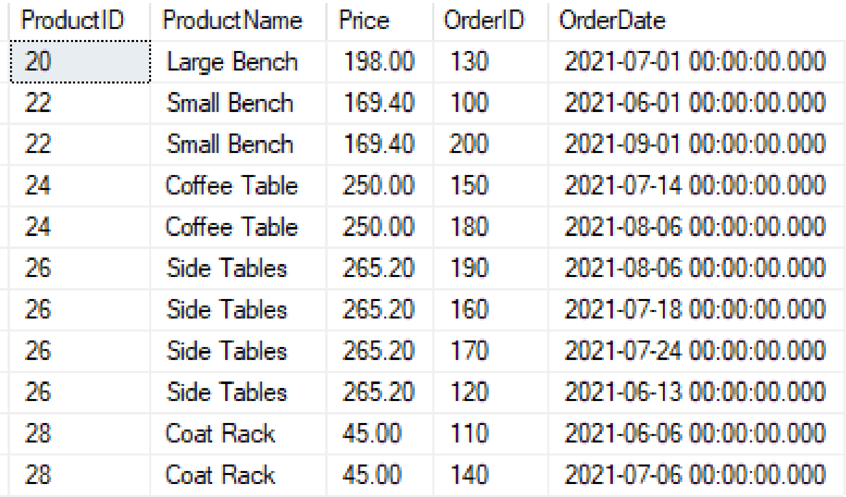
OrderID	CustID	ProdID	Qty	Orderdate
100	55	22	1	2021-06-01 00:00:00.000
110	60	28	2	2021-06-06 00:00:00.000
120	75	26	1	2021-06-13 00:00:00.000
130	50	20	1	2021-07-01 00:00:00.000
140	55	28	1	2021-07-06 00:00:00.000
150	65	24	1	2021-07-14 00:00:00.000
160	55	26	1	2021-07-18 00:00:00.000
170	50	26	1	2021-07-24 00:00:00.000
180	70	24	1	2021-08-06 00:00:00.000
190	70	26	1	2021-08-06 00:00:00.000
200	70	22	3	2021-09-01 00:00:00.000

Cool, so we'll see those rows in the final result set, too:



ProductID	ProductName	Price	OrderID	OrderDate
20	Large Bench	198.00	130	2021-07-01 00:00:00.000
22	Small Bench	169.40	100	2021-06-01 00:00:00.000
22	Small Bench	169.40	200	2021-09-01 00:00:00.000

Folks, this process keeps happening for all rows in the Products table, which is the left input. For each product, we figure out what rows correlate to it in the right input. The final result set ends up looking like this:



ProductID	ProductName	Price	OrderID	OrderDate
20	Large Bench	198.00	130	2021-07-01 00:00:00.000
22	Small Bench	169.40	100	2021-06-01 00:00:00.000
22	Small Bench	169.40	200	2021-09-01 00:00:00.000
24	Coffee Table	250.00	150	2021-07-14 00:00:00.000
24	Coffee Table	250.00	180	2021-08-06 00:00:00.000
26	Side Tables	265.20	190	2021-08-06 00:00:00.000
26	Side Tables	265.20	160	2021-07-18 00:00:00.000
26	Side Tables	265.20	170	2021-07-24 00:00:00.000
26	Side Tables	265.20	120	2021-06-13 00:00:00.000
28	Coat Rack	45.00	110	2021-06-06 00:00:00.000
28	Coat Rack	45.00	140	2021-07-06 00:00:00.000

To write the query that created this result set, we start with the left input. In this case, it's just a query against the Products table:

```
SELECT P.*
FROM Products as P
```

Then we outline our CROSS APPLY operator:

```
SELECT P.*
FROM Products as P
CROSS APPLY
(
  <inner query>
) AS <cross apply alias name>
```

In our case, we want the **<inner query>** to be a query against the Orders table, where we correlate the ProductID value from the left table. It looks like this:

```
SELECT P.*
FROM Products as P
CROSS APPLY
(
  SELECT OrderID, OrderDate
  FROM Orders
  WHERE ProdID = P.ProductID
) AS <cross apply alias name>
```

The WHERE clause in this inner query is where we see the correlation happening.

Then we'll outline an alias for our CROSS APPLY operator:

```
SELECT P.*
FROM Products as P
CROSS APPLY
(
  SELECT OrderID, OrderDate
  FROM Orders
  WHERE ProdID = P.ProductID
) AS RS
```

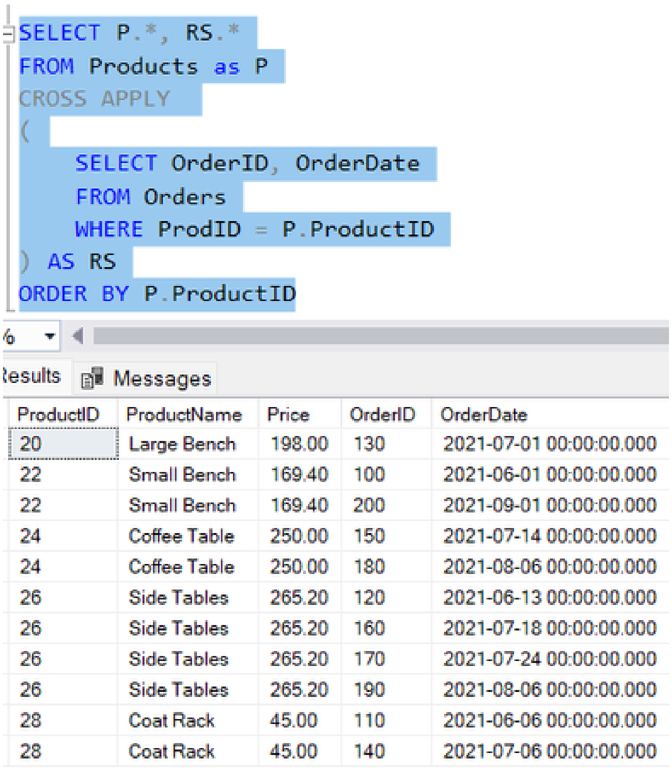
I chose to call it "*RS*" for "*Right Side*".

Finally in the outer SELECT statement, we can simply outline all columns from this derived CROSS APPLY table expression:

```
SELECT P.*, RS.*
FROM Products as P
CROSS APPLY
(
  SELECT OrderID, OrderDate
  FROM Orders
  WHERE ProdID = P.ProductID
) AS RS
ORDER BY P.ProductID
```

(I also put a simple ORDER BY clause to present the information in order by Product ID).

Again, here's what the final result set looks like:

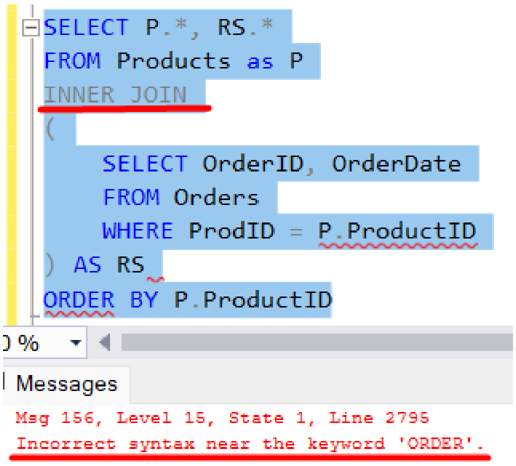


```
SELECT P.*, RS.*
FROM Products as P
CROSS APPLY
(
    SELECT OrderID, OrderDate
    FROM Orders
    WHERE ProdID = P.ProductID
) AS RS
ORDER BY P.ProductID
```

ProductID	ProductName	Price	OrderID	OrderDate
20	Large Bench	198.00	130	2021-07-01 00:00:00.000
22	Small Bench	169.40	100	2021-06-01 00:00:00.000
22	Small Bench	169.40	200	2021-09-01 00:00:00.000
24	Coffee Table	250.00	150	2021-07-14 00:00:00.000
24	Coffee Table	250.00	180	2021-08-06 00:00:00.000
26	Side Tables	265.20	120	2021-06-13 00:00:00.000
26	Side Tables	265.20	160	2021-07-18 00:00:00.000
26	Side Tables	265.20	170	2021-07-24 00:00:00.000
26	Side Tables	265.20	190	2021-08-06 00:00:00.000
28	Coat Rack	45.00	110	2021-06-06 00:00:00.000
28	Coat Rack	45.00	140	2021-07-06 00:00:00.000

The reason it's called a *CROSS* APPLY is because for each left input, we basically create a *cross product* of all the rows in the right table that correlate to it.

The way I think of a CROSS APPLY is like a JOIN on a *query* instead of a table. The JOIN operator does not let you specify a *query* as the right input. For example, this doesn't work:



```
SELECT P.*, RS.*
FROM Products as P
INNER JOIN
(
    SELECT OrderID, OrderDate
    FROM Orders
    WHERE ProdID = P.ProductID
) AS RS
ORDER BY P.ProductID
```

```
Msg 156, Level 15, State 1, Line 2795
Incorrect syntax near the keyword 'ORDER'.
```

Don't get too focused on that error message. SQL Server is simply very confused about what we're trying to do here.

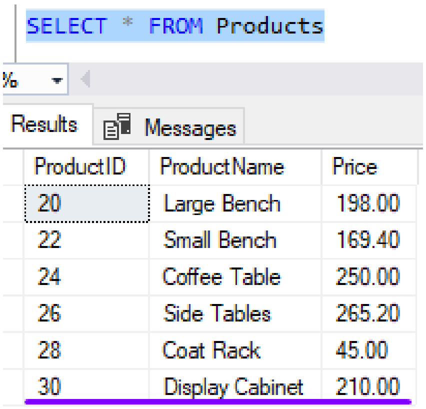
If we need to use a correlated query as the right input, instead of a table, we should think about using the CROSS APPLY operator.

OUTER APPLY

What if there were a product that does not have any orders made for it? Let's add a row to the Products table:

```
INSERT INTO Products (ProductName, Price)
VALUES ('Display Cabinet', 210.00)
```

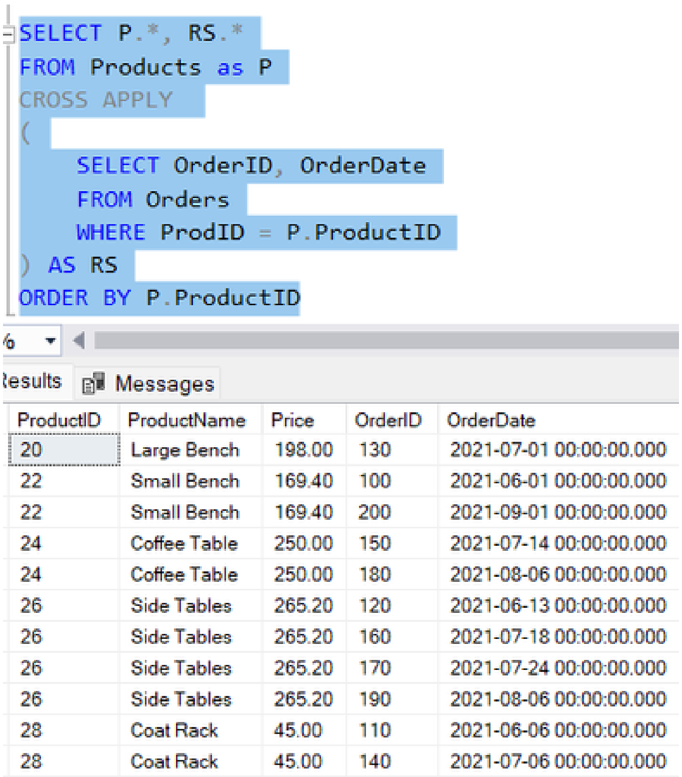
Let's query our Products table to see what the ID is for this new product:



```
SELECT * FROM Products
```

ProductID	ProductName	Price
20	Large Bench	198.00
22	Small Bench	169.40
24	Coffee Table	250.00
26	Side Tables	265.20
28	Coat Rack	45.00
30	Display Cabinet	210.00

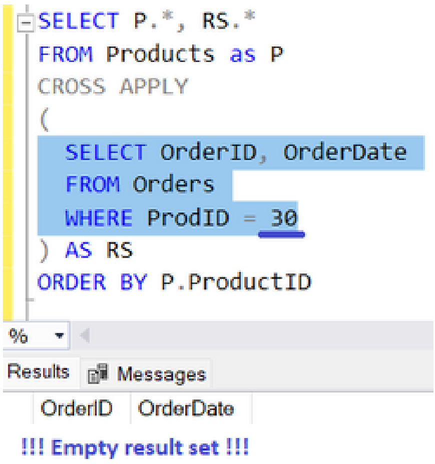
Now let's run our CROSS APPLY query again:



```
SELECT P.*, RS.*
FROM Products as P
CROSS APPLY
(
    SELECT OrderID, OrderDate
    FROM Orders
    WHERE ProdID = P.ProductID
) AS RS
ORDER BY P.ProductID
```

ProductID	ProductName	Price	OrderID	OrderDate
20	Large Bench	198.00	130	2021-07-01 00:00:00.000
22	Small Bench	169.40	100	2021-06-01 00:00:00.000
22	Small Bench	169.40	200	2021-09-01 00:00:00.000
24	Coffee Table	250.00	150	2021-07-14 00:00:00.000
24	Coffee Table	250.00	180	2021-08-06 00:00:00.000
26	Side Tables	265.20	120	2021-06-13 00:00:00.000
26	Side Tables	265.20	160	2021-07-18 00:00:00.000
26	Side Tables	265.20	170	2021-07-24 00:00:00.000
26	Side Tables	265.20	190	2021-08-06 00:00:00.000
28	Coat Rack	45.00	110	2021-06-06 00:00:00.000
28	Coat Rack	45.00	140	2021-07-06 00:00:00.000

Notice we get the same result set as before, and we don't see a row for our new '*Display Cabinet*' product. The CROSS APPLY operator will exclude rows from the left that return an empty result set in the inner query. Let's plug in the ProductID of our new product to prove that it returns an empty result set in the inner query:



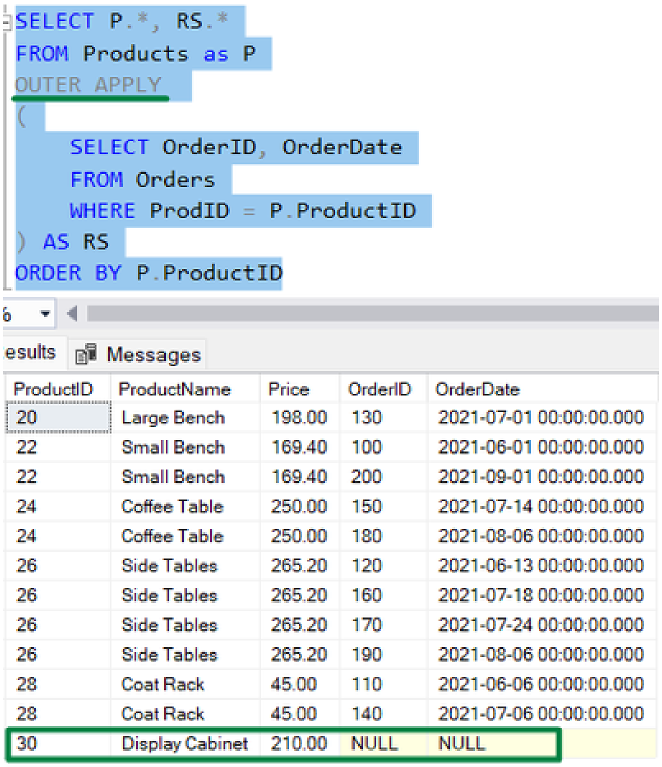
```
SELECT P.*, RS.*
FROM Products as P
CROSS APPLY
(
    SELECT OrderID, OrderDate
    FROM Orders
    WHERE ProdID = 30
) AS RS
ORDER BY P.ProductID
```

OrderID	OrderDate

!!! Empty result set !!!

Since this new product doesn't have any orders made for it, it is excluded from the result set of the CROSS APPLY query.

But what if we wanted to see this new product in our final result set anyway, despite it not having any orders? We could use the OUTER APPLY operator instead:



```
SELECT P.*, RS.*
FROM Products as P
OUTER APPLY
(
    SELECT OrderID, OrderDate
    FROM Orders
    WHERE ProdID = P.ProductID
) AS RS
ORDER BY P.ProductID
```

ProductID	ProductName	Price	OrderID	OrderDate
20	Large Bench	198.00	130	2021-07-01 00:00:00.000
22	Small Bench	169.40	100	2021-06-01 00:00:00.000
22	Small Bench	169.40	200	2021-09-01 00:00:00.000
24	Coffee Table	250.00	150	2021-07-14 00:00:00.000
24	Coffee Table	250.00	180	2021-08-06 00:00:00.000
26	Side Tables	265.20	120	2021-06-13 00:00:00.000
26	Side Tables	265.20	160	2021-07-18 00:00:00.000
26	Side Tables	265.20	170	2021-07-24 00:00:00.000
26	Side Tables	265.20	190	2021-08-06 00:00:00.000
28	Coat Rack	45.00	110	2021-06-06 00:00:00.000
28	Coat Rack	45.00	140	2021-07-06 00:00:00.000
30	Display Cabinet	210.00	NULL	NULL

OUTER APPLY will include rows from the left side that return an empty result set in the inner query.

Let's delete the new product for cleanup:

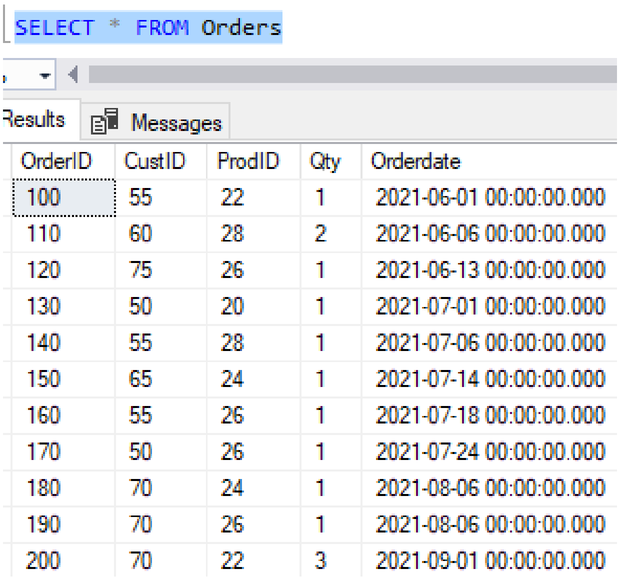
```
DELETE FROM Products WHERE ProductID = 30
DBCC CHECKIDENT('Products', RESEED, 28)
```

The PIVOT Operator

The PIVOT operator is a tool we can use to group and aggregate data, and present it in a state of *columns* instead of a state of rows.

PIVOT is used to present aggregate data in a different way. When it comes to querying data, there are times when we not only need to make sure the data we've gathered is accurate, but we also might want to present the data in a format that is easy to understand. The PIVOT operator is used to do just that.

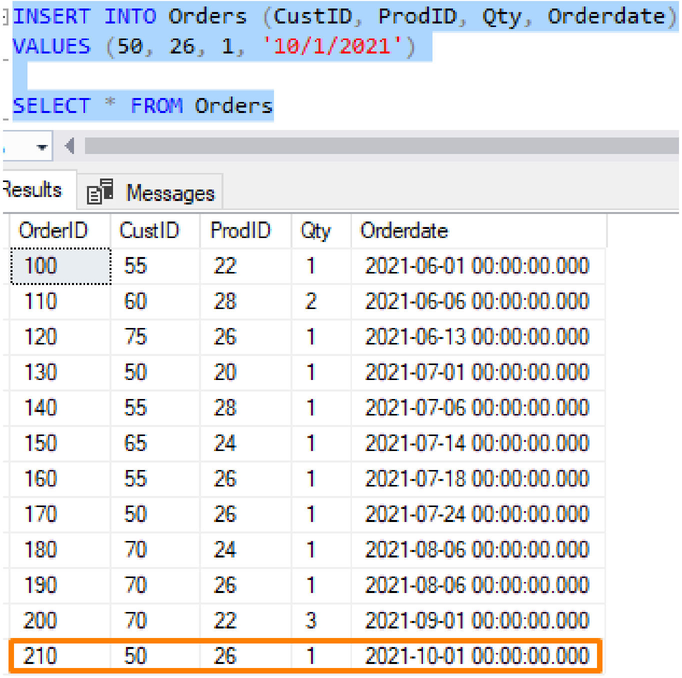
Take a look at the Orders table:



```
SELECT * FROM Orders
```

OrderID	CustID	ProdID	Qty	Orderdate
100	55	22	1	2021-06-01 00:00:00.000
110	60	28	2	2021-06-06 00:00:00.000
120	75	26	1	2021-06-13 00:00:00.000
130	50	20	1	2021-07-01 00:00:00.000
140	55	28	1	2021-07-06 00:00:00.000
150	65	24	1	2021-07-14 00:00:00.000
160	55	26	1	2021-07-18 00:00:00.000
170	50	26	1	2021-07-24 00:00:00.000
180	70	24	1	2021-08-06 00:00:00.000
190	70	26	1	2021-08-06 00:00:00.000
200	70	22	3	2021-09-01 00:00:00.000

Let's actually add one more row for Customer # 50 to this table to discuss later:

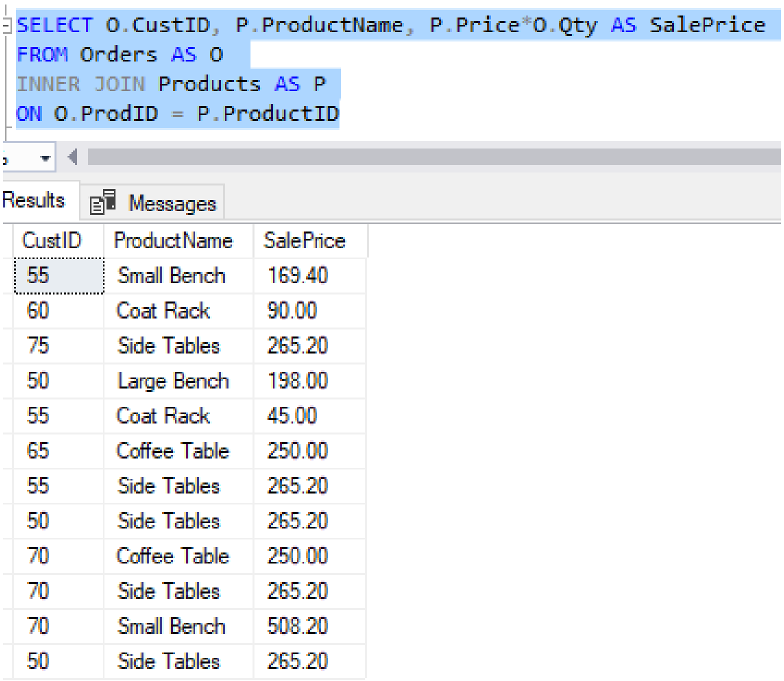


```
INSERT INTO Orders (CustID, ProdID, Qty, Orderdate)
VALUES (50, 26, 1, '10/1/2021')

SELECT * FROM Orders
```

OrderID	CustID	ProdID	Qty	Orderdate
100	55	22	1	2021-06-01 00:00:00.000
110	60	28	2	2021-06-06 00:00:00.000
120	75	26	1	2021-06-13 00:00:00.000
130	50	20	1	2021-07-01 00:00:00.000
140	55	28	1	2021-07-06 00:00:00.000
150	65	24	1	2021-07-14 00:00:00.000
160	55	26	1	2021-07-18 00:00:00.000
170	50	26	1	2021-07-24 00:00:00.000
180	70	24	1	2021-08-06 00:00:00.000
190	70	26	1	2021-08-06 00:00:00.000
200	70	22	3	2021-09-01 00:00:00.000
210	50	26	1	2021-10-01 00:00:00.000

We'll start with a simple JOIN query to tell us the customer, product, and sale price of each order:



```
SELECT O.CustID, P.ProductName, P.Price*O.Qty AS SalePrice
FROM Orders AS O
INNER JOIN Products AS P
ON O.ProdID = P.ProductID
```

```
CustID  ProductName   SalePrice
55      Small Bench   169.40
60      Coat Rack     90.00
75      Side Tables   265.20
50      Large Bench   198.00
55      Coat Rack     45.00
65      Coffee Table  250.00
55      Side Tables   265.20
50      Side Tables   265.20
70      Coffee Table  250.00
70      Side Tables   265.20
70      Small Bench   508.20
50      Side Tables   265.20
```

Let's think about some information we might want to gather from this data. What if we wanted to know how much income we have made per customer, **per product**.

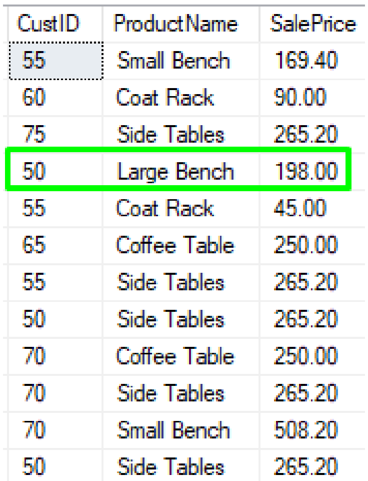
In other words, for each customer, we want to know how much money they have spent on each of our products.

We don't have very many customers or products in our database, so let's put together a small grid to help us gather this data:

CustID	Large Bench	Small Bench	Coffee Table	Side Tables	Coat Rack
50					
55					
60					
65					
70					
75					

At the intersection of the customer ID and product is where we put how much money we have made from that customer, for that product.

We'll go down the list of orders and gather our details. We'll start with Customer # 50. The first product they purchased was the Large Bench:

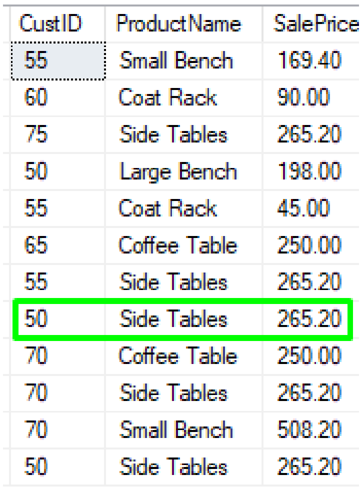


CustID	ProductName	SalePrice
55	Small Bench	169.40
60	Coat Rack	90.00
75	Side Tables	265.20
50	Large Bench	198.00
55	Coat Rack	45.00
65	Coffee Table	250.00
55	Side Tables	265.20
50	Side Tables	265.20
70	Coffee Table	250.00
70	Side Tables	265.20
70	Small Bench	508.20
50	Side Tables	265.20

Cool, so let's fill our grid with the income we have made from that customer for the large bench:

CustID	Large Bench	Small Bench	Coffee Table	Side Tables	Coat Rack
50	198.00				
55					
60					
65					
70					
75					

So far, so good. Then we keep going down the list of orders to see what else customer # 50 has purchased. The next item appears to be the Side Tables:

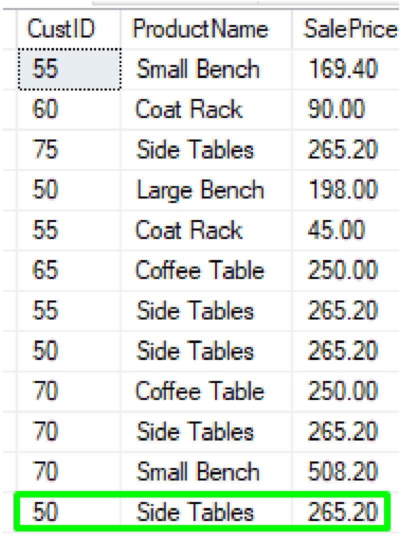


CustID	ProductName	SalePrice
55	Small Bench	169.40
60	Coat Rack	90.00
75	Side Tables	265.20
50	Large Bench	198.00
55	Coat Rack	45.00
65	Coffee Table	250.00
55	Side Tables	265.20
50	Side Tables	265.20
70	Coffee Table	250.00
70	Side Tables	265.20
70	Small Bench	508.20
50	Side Tables	265.20

Alright, let's populate the Side Tables column with this sale price:

CustID	Large Bench	Small Bench	Coffee Table	Side Tables	Coat Rack
50	198.00			**265.20**	
55					
60					
65					
70					
75					

We continue down the list of orders. It looks like customer # 50 has one final order, which is another purchase of the side tables:



CustID	ProductName	SalePrice
55	Small Bench	169.40
60	Coat Rack	90.00
75	Side Tables	265.20
50	Large Bench	198.00
55	Coat Rack	45.00
65	Coffee Table	250.00
55	Side Tables	265.20
50	Side Tables	265.20
70	Coffee Table	250.00
70	Side Tables	265.20
70	Small Bench	508.20
50	Side Tables	265.20

Okay, so we actually need to update the value in our grid for this customer-product combination. It looks like we have made a *sum* of $530.40 from Side Table sales to customer # 50. Let's update the value in our grid:

CustID	**Large Bench**	**Small Bench**	**Coffee Table**	**Side Tables**	**Coat Rack**
50	198.00			**530.40**	
55					
60					
65					
70					
75					

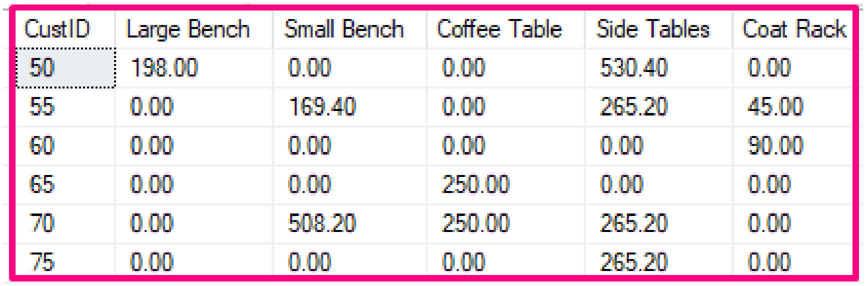
So that's all the orders for customer # 50. That customer hasn't purchased any other products, so there isn't anything to put at the intersection of that customer and those unpurchased products. We can just put **0.00** for those products:

CustID	Large Bench	Small Bench	Coffee Table	Side Tables	Coat Rack
50	198.00	**0.00**	**0.00**	530.40	**0.00**
55					
60					
65					
70					
75					

Cool, so we've gathered everything for customer # 50.

Folks, we would need to do this work for each customer. We would need to go down the list of orders and see what products have been purchased for each customer and populate our grid accordingly. This might be easy enough for a total of 11 orders, but what if there were *hundreds* or *thousands* of orders? It would take a very long time to gather these details.

Using PIVOT, we can write a simple query to present the information in exactly the way we see it in our grid. Like this:



CustID	Large Bench	Small Bench	Coffee Table	Side Tables	Coat Rack
50	198.00	0.00	0.00	530.40	0.00
55	0.00	169.40	0.00	265.20	45.00
60	0.00	0.00	0.00	0.00	90.00
65	0.00	0.00	250.00	0.00	0.00
70	0.00	508.20	250.00	265.20	0.00
75	0.00	0.00	0.00	265.20	0.00

(I'm purposefully not showing you the query that created this *awesome* result set. We'll figure it out together).

Writing a PIVOT query

…
…
…