CHAPTER 17

Table-Valued Functions

We've had a solid introduction to functions of the scalar kind. Now we'll look at the other functional option provided to us by SQL Server: table-valued functions, or TVFs for short. TVFs let you create parameterized tables, which you can use in your queries to provide extra functionality that a normal join would struggle to match. Time to talk tables again!

Why TVFs Are Cool

TVFs were introduced in SQL Server 2005, and have become a widely used component of most SQL Server developers' toolkits. I use them all the time, as they often provide the granular control needed to solve a particular problem that a view or stored procedure cannot give. Views and stored procedures (which we'll meet in the very next chapter) both have their place and are both very powerful, but there are certain times when nothing but a TVF will do.

There are several reasons why TVFs are so great:

- A view can only contain a single `SELECT` statement; a TVF can contain multiple statements, all of which combine to produce the end table.
- TVFs can replace stored procedures. Stored procedures can return result sets, but they cannot be easily used in queries. TVFs can be embedded directly in queries.
- Bugs in TVFs can be easily fixed, as long as the function signature doesn't change.
- TVFs can accept parameters, which allow you to dictate the output of the table.
- TVFs can hide a lot of complexity and improve code reuse.
- TVFs are great when a fairly small result set is being returned.

Sometimes, TVFs Are Not So Cool

Of course, there are times when a TVF may not be the best solution to a problem. As with any technology, TVFs can be abused and used for evil purposes! Some reasons why you wouldn't use a TVF are (duh-duh-DUH!):

- TVFs can greatly affect performance. A TVF is executed for every single row returned by a `SELECT` statement, so great care needs to be taken in how they are used.
- Views offer much lower overhead compared to TVFs; if you can do what you need to do in a single `SELECT` statement, use a view.
- If you don't need to use the result set in other queries, use a stored procedure; these generally perform better than TVFs.
- If a large result set is being returned, a TVF may not offer the best solution due to their procedural nature. Consider a set-based solution instead (we'll be looking at an example of this in the next two chapters).

None of this should put you off using TVFs. I've seen TVFs make a 10-minute import process run for 4 hours. Conversely, I've also seen TVFs reduce queries that took 30 minutes to run down to a few seconds. When you are trying to improve performance, it's all about knowing what the various technologies can do, and having a play-around with them to see what works best.

Building a TVF

Let's put a TVF together so we can see how they work. We want to build something that tells us how many subrecords of a particular type are available for each contact. We also want to give the user the option of requesting totals for all types of subrecord (addresses, notes, phone numbers, and roles), or just for one particular type of subrecord. The record set we will return is the following:

- `ContactId`

- `AddressCount`
- `NoteCount`
- `PhoneNumberCount`
- `RoleCount`

If a count is available for the selected type, we will return a record for the requested contact. If no count is available, no record will be returned.

This function needs to accept two parameters: the `ContactId` for which we are requesting totals, and a `TableName` parameter, to specify from which table we should be returning totals. If `'All'` is passed as the `TableName`, all totals will be returned.

The header of this function looks similar to the UDF we created in the last chapter, with a function name and parameters:

```
CREATE FUNCTION dbo.ContactCounts
(
@ContactId INT,
@TableName VARCHAR(40)
)
```

This gives us the parameters we were talking about. Now we need to return a table instead of a scalar value. We do this on the next line:

```
RETURNS @CountsTable TABLE (ContactId INT, AddressCount INT, NoteCount INT, PhoneNumberCount INT, R
```

It's important to note that we have specified a variable name for the table, `@CountsTable`. A variable name *must* be specified; we use this name to access the table in the body of the function. After the `TABLE` keyword is used to declare we are returning a table, we have the usual list of column definitions.

We've declared the columns we talked about earlier. Now we move into the body of the function. This is where the action takes place. Here's what we want to do:

- If the `@TableName` variable holds `'All'` or `'ContactAddresses'`, obtain the address count for the specified contact.
- If the `@TableName` variable holds `'All'` or `'ContactNotes'`, obtain the note count for the specified contact.
- If the `@TableName` variable holds `'All'` or `'ContactPhoneNumbers'`, obtain the phone number count for the specified contact.
- If the `@TableName` variable holds `'All'` or `'ContactRoles'`, obtain the role count for the specified contact.
- If at least one of the counts is greater than zero, insert a row into the table.
- Return the table.

The first four lines are pretty much the same; we just obtain a count for the required table, and then assign that count to an appropriate variable. Once the variables have been assigned, we can attempt the insert. The full function definition follows, along with a `DROP FUNCTION` check at the top.

```
USE AddressBook;

IF EXISTS (SELECT 1 FROM sys.objects WHERE [name] = 'ContactCounts'
AND [type] = 'TF')
BEGIN
DROP FUNCTION dbo.ContactCounts;
END;

GO

-- Table-Valued function to return contact record counts
-- Acceptable values for @TableName are All, ContactAddresses, ContactNotes, --
```

```
-- ContactPhoneNumbers, and ContactRoles.
CREATE FUNCTION dbo.ContactCounts
(@ContactId INT, @TableName VARCHAR(40)
)
RETURNS @CountsTable TABLE (ContactId INT, AddressCount INT, NoteCount INT,
PhoneNumberCount INT, RoleCount INT)
AS
BEGIN

-- Variables to hold the counts
DECLARE @AddressCount INT, @NoteCount INT, @PhoneNumberCount INT, @RoleCount INT;

-- Return address count
IF (@TableName IN ('All', 'ContactAddresses'))
BEGIN
SELECT @AddressCount = COUNT(1)
FROM dbo.ContactAddresses CA
WHERE CA.ContactId = @ContactId;
END;

-- Return note count
IF (@TableName IN ('All', 'ContactNotes'))
BEGIN
SELECT @NoteCount = COUNT(1) FROM dbo.ContactNotes CN WHERE CN.ContactId = @ContactId;
END;

-- Return phone number count
IF (@TableName IN ('All', 'ContactPhoneNumbers'))
BEGIN
SELECT @PhoneNumberCount = COUNT(1) FROM dbo.ContactPhoneNumbers CPN WHERE CPN.ContactId = @Contact
END;

-- Return role count
IF (@TableName IN ('All', 'ContactRoles'))
BEGIN
SELECT @RoleCount = COUNT(1) FROM dbo.ContactRoles CR WHERE CR.ContactId = @ContactId;
END;

-- If we have at least one valid value, add the row
IF (@AddressCount > 0 OR @NoteCount > 0 OR @PhoneNumberCount > 0 OR @RoleCount > 0)
BEGIN

INSERT INTO @CountsTable (ContactId, AddressCount, NoteCount, PhoneNumberCount, RoleCount) SELECT @

END;

RETURN;

END;

GO
```

There is quite a bit of code here, but it isn't overly complicated. We declare four variables to hold the counts we generate. The first count to be returned is for `ContactAddresses`; if the user has requested all totals or just the address total, we obtain the count from the `ContactAddresses` table. We do exactly the same for notes, phone numbers, and roles.

This final section of code is where the magic happens:

```
-- If we have at least one valid value, add the row
IF (@AddressCount > 0 OR @NoteCount > 0 OR @PhoneNumberCount > 0 OR @RoleCount > 0)
BEGIN

INSERT INTO @CountsTable (ContactId, AddressCount, NoteCount, PhoneNumberCount, RoleCount) SELECT @

END;

RETURN;
```

If at least one of the variables is greater than zero, we will insert a row into the `@CountTable` table variable. We return the `ContactId` and the appropriate counts. Note that some of the counts will be `NULL` if only one

count in particular was requested. At this point, we have a row in the table. The `RETURN` keyword is called and the function completes. Remember that you do not return the table variable (unlike scalar functions, in which you do return the variable containing the value you wish to return); you just call `RETURN`. SQL Server knows the table variable was declared in the function header, so it knows what it needs to return.

Save this script as `c:\temp\sqlbasics\apply\25 - Create ContactCounts Function.sql`, and run it. After seeing **Command(s) completed successfully**, the function is ready to rock!

Adding the Script to SQLCMD

Don't forget to add the script to our `00 - Apply.sql` script:

```
:setvar currentFile "25 - Create ContactCounts Function.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

Creating the Rollback Script

We also need to create a rollback script. Enter this script:

```
USE AddressBook;

IF EXISTS (SELECT 1 FROM sys.objects WHERE [name] = 'ContactCounts' AND [type] = 'TF')
BEGIN
DROP FUNCTION dbo.ContactCounts;
END;

GO
```

Note the `type` column check—we are looking for `TF`, which is how SQL Server identifies a TVF.

Save the script as `c:\temp\sqlbasics\rollback\25 - Create ContactCounts Function Rollback.sql`. Add this to `00 - Rollback.sql`.

```
:setvar currentFile "25 - Create ContactCounts Function Rollback.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

Using TVFs in Queries

Now to test our query. Open up a New Query Window. Before typing anything, press Ctrl+Shift+R to refresh Intellisense (the pop-up information that appears as you are typing). Refreshing Intellisense will ensure the new function we just created will be picked up. Type this:

```
USE AddressBook;

SELECT * FROM dbo.ContactCounts(
```

Intellisense should display something like the pop-up shown in Figure 17-1.

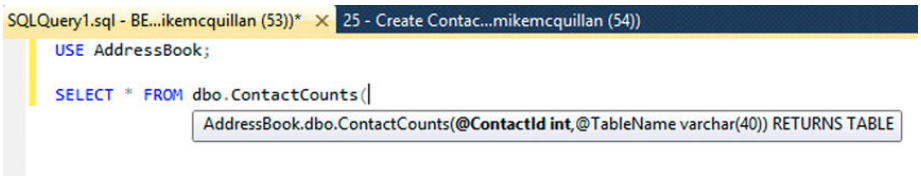


Figure 17-1. *Intellisense at work*

We can see the function declaration, and Intellisense is telling us what values we need to provide. It is also telling us the function returns a table. Let's complete the statement. We'll start by returning all records for Stephen Gerrard (`ContactId` 1).

```
USE AddressBook;

SELECT * FROM dbo.ContactCounts(1, 'All');
```

Figure 17-2 has the results.

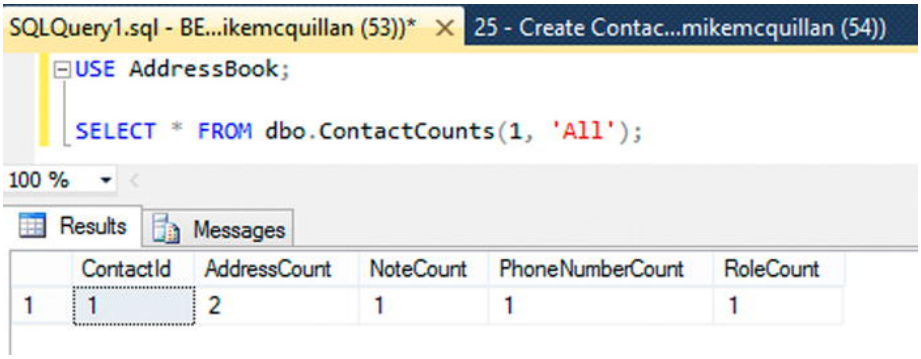


Figure 17-2. *Returning counts for all tables*

Looks good! Change `'All'` to `'ContactAddresses'`. The results should match Figure 17-3.

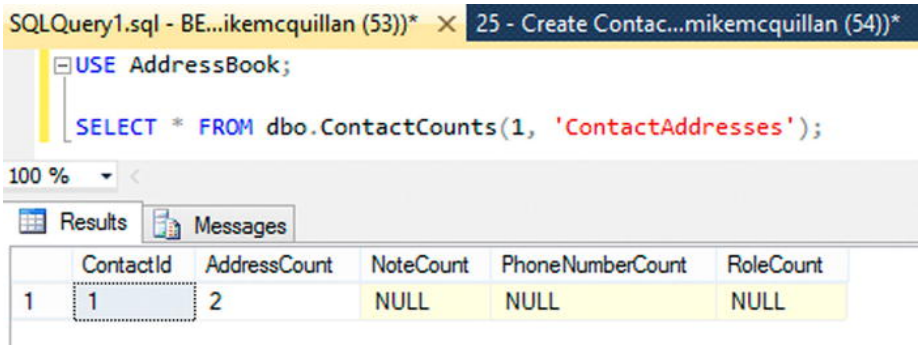


Figure 17-3. *Returning the count for the `ContactAddresses` table*

The function returns just the count we asked for. If we specify a table name that is not recognized by the function, no row is returned (as you can see in Figure 17-4).

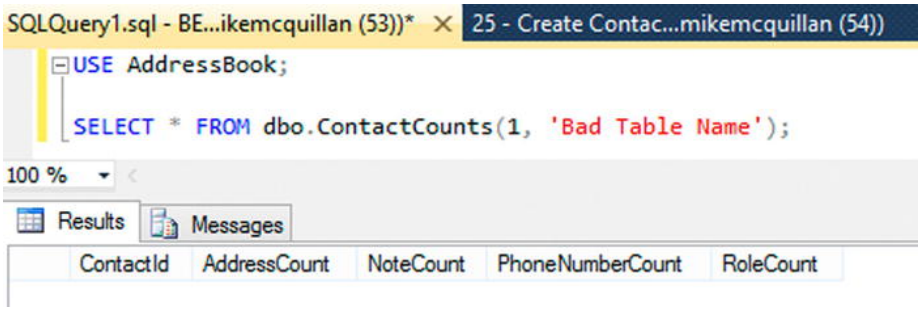


Figure 17-4. *Trying to return a count for a nonexistent table*

Calling a TVF

Maybe you've noticed we called the TVF by using a `SELECT`. In fact, the `SELECT` statement looks exactly like any other `SELECT` statement we've used so far, except it has parameters in brackets after it. But apart from this difference, SQL Server essentially treats the function just like a table. These are only rules to follow when using a TVF in a `SELECT`:

- The schema name must be specified

- A value for each parameter must be provided (you can specify default values for parameters, but even if a default is specified, you must provide the `DEFAULT` keyword)

I mentioned TVFs are treated just like a table, which means we should be able to join the TVF to other tables. We'll see if and how that is possible.

Joining to a TVF

A long time ago in a chapter not far, far away, we discussed the various types of join: `INNER`, `LEFT`, `RIGHT`, `FULL`, and `CROSS JOIN`s. We're going to modify our basic `SELECT` statement to return some contact details alongside the totals. Here's a statement using an `INNER JOIN`:

```
SELECT dbo.ContactName(C.FirstName, C.LastName) AS ContactName, C.DateOfBirth, CC.AddressCount, CC.
```

If you try to run this, you'll see an error.

```
Msg 4104, Level 16, State 1, Line 10
The multi-part identifier "C.ContactId" could not be bound.
```

This is the `C.ContactId` referenced on the `INNER JOIN` line. You can try changing this to a `LEFT OUTER JOIN`, a `RIGHT OUTER JOIN`, a `FULL OUTER JOIN`, or a `CROSS JOIN`. None of them will work. However, if we change the line:

```
INNER JOIN dbo.ContactCounts(C.ContactId, 'All') CC
```

to:

```
INNER JOIN dbo.ContactCounts(1, 'All') CC
```

then you'll see the results shown in Figure 17-5.

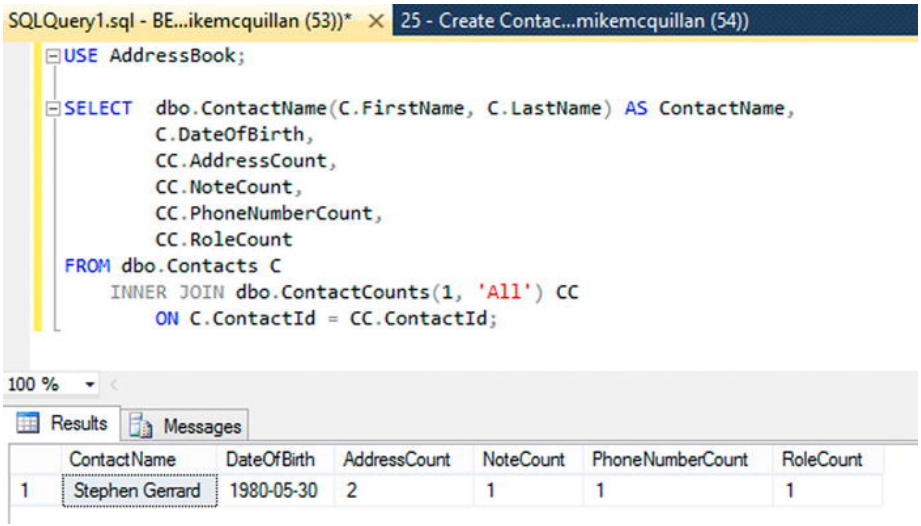


Figure 17-5. *Using the function (badly) with an* `INNER JOIN`

This isn't much good to us though—we need to explicitly specify a value for `ContactId`. It's actually worse than it looks though. If we keep the value 1, but change the `INNER JOIN` to `LEFT JOIN`, all contacts will be returned, but with the exception of `ContactId` 1, all totals will be `NULL`. You can see the `NULL` columns in Figure 17-6.

SQLQuery1.sql - BE...ikemcquillan (53))* | 25 - Create Contac...mikemcquillan (54))

```
USE AddressBook;

SELECT  dbo.ContactName(C.FirstName, C.LastName) AS ContactName,
        C.DateOfBirth,
        CC.AddressCount,
        CC.NoteCount,
        CC.PhoneNumberCount,
        CC.RoleCount
FROM dbo.Contacts C
    LEFT JOIN dbo.ContactCounts(1, 'All') CC
        ON C.ContactId = CC.ContactId;
```

100 %

Results | Messages

	ContactName	DateOfBirth	AddressCount	NoteCount	PhoneNumberCount	RoleCount
1	Stephen Gerrard	1980-05-30	2	1	1	1
2	Dennis Potter	1935-05-17	NULL	NULL	NULL	NULL
3	Richard Adams	1920-05-09	NULL	NULL	NULL	NULL
4	Bertie McQuillan	2001-06-30	NULL	NULL	NULL	NULL
5	Walt Disney	1966-12-05	NULL	NULL	NULL	NULL
6	Barbara Gordon	1952-01-11	NULL	NULL	NULL	NULL
7	Josephine Bailey	1949-05-31	NULL	NULL	NULL	NULL
8	Linda Canoglu	1959-07-11	NULL	NULL	NULL	NULL

Figure 17-6. *Using the function with a* `LEFT JOIN` *(still used badly)*

Why can't we join, passing the `ContactId` value to the function? The problem is that the results from the `Contacts` table haven't been generated at the time the `ContactCounts` function is being returned, so there is no `ContactId` column available to pass to the function.

Using a join, the `Contacts` table and the `ContactCounts` TVF are evaluated at the same time. But to be able to pass the `ContactId` column to `ContactCounts`, the `Contacts` table needs to be evaluated first, so the `ContactId` is available to be passed to the `ContactCounts` table. Hard-coding the value of 1 works because no evaluation is required—SQL Server knows that the value of 1 is 1.

SQL Server provides something special that allows us to join tables to TVFs correctly. Say hello to the `APPLY` operator.

The APPLY Operator

The `APPLY` operator executes the TVF against each row returned by the principal table in a query. The principal table doesn't have to be a physical table; it could be a query used to return a logical table, a view, or another TVF. For simplicity, we'll carry on using a physical table. The principal table acts as the left table in the query, and the TVF acts as the right table. The rows in the left table are evaluated first, and then applied to the TVF on the right-hand side of the query. Columns from both sides of the query can be combined to produce a result set.

CROSS APPLY

There are two types of `APPLY`. The `CROSS APPLY` operator acts like an `INNER JOIN`. A row is only returned if it exists in both the left- and right-hand sides of a query. A `CROSS APPLY` doesn't have an `ON` clause, so a statement using it looks like this:

```
SELECT Columns FROM LeftTable CROSS APPLY RightTable;
```

Here is our previous `SELECT` statement, changed to use a `CROSS APPLY`.

```
SELECT dbo.ContactName(C.FirstName, C.LastName) AS ContactName, C.DateOfBirth, CC.AddressCount,
```

Run this, and as shown in Figure 17-7, the results we were looking for earlier appear!

SQLQuery1.sql - BE...ikemcquillan (53))* | 25 - Create Contac...mikemcquillan (54))

```
USE AddressBook;

SELECT  dbo.ContactName(C.FirstName, C.LastName) AS ContactName,
        C.DateOfBirth,
        CC.AddressCount,
        CC.NoteCount,
        CC.PhoneNumberCount,
        CC.RoleCount
FROM dbo.Contacts C
    CROSS APPLY dbo.ContactCounts(C.ContactId, 'All') CC;
```

100 %

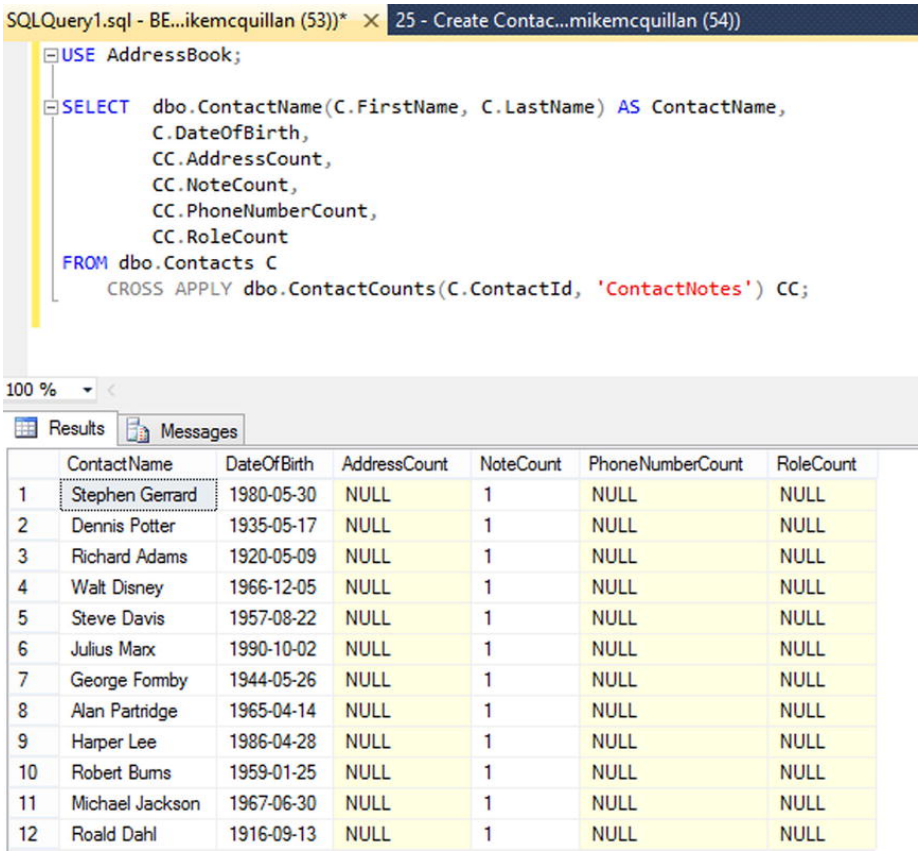
Results | Messages

	ContactName	DateOfBirth	AddressCount	NoteCount	PhoneNumberCount	RoleCount
1	Stephen Gerrard	1980-05-30	2	1	1	1
2	Dennis Potter	1935-05-17	1	1	0	2
3	Richard Adams	1920-05-09	1	1	1	1
4	Bertie McQuillan	2001-06-30	1	0	0	1
5	Walt Disney	1966-12-05	1	1	0	1
6	Barbara Gordon	1952-01-11	1	0	0	3
7	Josephine Bailey	1949-05-31	1	0	1	1
8	Linda Canoglu	1959-07-11	1	0	1	1
9	Grace McQuillan	1993-09-27	1	0	0	1

Figure 17-7. *Using the function with* `CROSS APPLY` *(successfully!)*

You can use the `APPLY` operator with anything that returns a result set. You could wrap another `SELECT` statement in brackets and use that with a `CROSS APPLY`, for example.

Remember that `CROSS APPLY` acts as an `INNER JOIN`, so if we say just return the totals for `'ContactNotes'` instead of `'All'`, this returns 12 results as shown in Figure 17-8, where the original call using `'All'` returned 20. This is because several records don't actually have any note records associated with them, so they've been removed from the query. We need some sort of outer join to resolve this problem.



	ContactName	DateOfBirth	AddressCount	NoteCount	PhoneNumberCount	RoleCount
1	Stephen Gerrard	1980-05-30	NULL	1	NULL	NULL
2	Dennis Potter	1935-05-17	NULL	1	NULL	NULL
3	Richard Adams	1920-05-09	NULL	1	NULL	NULL
4	Walt Disney	1966-12-05	NULL	1	NULL	NULL
5	Steve Davis	1957-08-22	NULL	1	NULL	NULL
6	Julius Marx	1990-10-02	NULL	1	NULL	NULL
7	George Formby	1944-05-26	NULL	1	NULL	NULL
8	Alan Partridge	1965-04-14	NULL	1	NULL	NULL
9	Harper Lee	1986-04-28	NULL	1	NULL	NULL
10	Robert Burns	1959-01-25	NULL	1	NULL	NULL
11	Michael Jackson	1967-06-30	NULL	1	NULL	NULL
12	Roald Dahl	1916-09-13	NULL	1	NULL	NULL

Figure 17-8. *Using the function for* `ContactNotes` *only*

OUTER APPLY

Just like `CROSS APPLY` acts like an `INNER JOIN`, `OUTER APPLY` acts like a `LEFT OUTER JOIN`. `OUTER APPLY` works in exactly the same manner as `CROSS APPLY`, except it will return all rows from the table on the left of the query, regardless of whether there are matching rows on the right side.

If we change the query in the previous section to use an `OUTER APPLY` instead of `CROSS APPLY`, our result set brings back 20 rows or more (depending upon when you last rebuilt your database). Check out **Figure 17-9**.

SQLQuery1.sql - BE...ikemcquillan (53))* | 25 - Create Contac...mikemcquillan (54))

```
USE AddressBook;

SELECT  dbo.ContactName(C.FirstName, C.LastName) AS ContactName,
        C.DateOfBirth,
        CC.AddressCount,
        CC.NoteCount,
        CC.PhoneNumberCount,
        CC.RoleCount
FROM dbo.Contacts C
    OUTER APPLY dbo.ContactCounts(C.ContactId, 'ContactNotes') CC;
```

100 %

Results | Messages

	ContactName	DateOfBirth	AddressCount	NoteCount	PhoneNumberCount	RoleCount
1	Stephen Gerrard	1980-05-30	NULL	1	NULL	NULL
2	Dennis Potter	1935-05-17	NULL	1	NULL	NULL
3	Richard Adams	1920-05-09	NULL	1	NULL	NULL
4	Bertie McQuillan	2001-06-30	NULL	NULL	NULL	NULL
5	Walt Disney	1966-12-05	NULL	1	NULL	NULL
6	Barbara Gordon	1952-01-11	NULL	NULL	NULL	NULL
7	Josephine Bailey	1949-05-31	NULL	NULL	NULL	NULL
8	Linda Canoglu	1959-07-11	NULL	NULL	NULL	NULL
9	Grace McQuillan	1993-09-27	NULL	NULL	NULL	NULL
10	Vera Black	1984-08-03	NULL	NULL	NULL	NULL
11	Angelica Jones	1981-02-04	NULL	NULL	NULL	NULL
12	Steve Davis	1957-08-22	NULL	1	NULL	NULL
13	Allison Fisher	1968-02-24	NULL	NULL	NULL	NULL
14	Julius Marx	1990-10-02	NULL	1	NULL	NULL
15	George Formby	1944-05-26	NULL	1	NULL	NULL
16	Alan Partridge	1965-04-14	NULL	1	NULL	NULL
17	Harper Lee	1986-04-28	NULL	1	NULL	NULL
18	Robert Burns	1959-01-25	NULL	1	NULL	NULL
19	Michael Jackson	1967-06-30	NULL	1	NULL	NULL
20	Roald Dahl	1916-09-13	NULL	1	NULL	NULL
21	Laura Robson	1994-01-21	NULL	NULL	NULL	NULL
22	Bryan Ferry	1945-09-26	NULL	NULL	NULL	NULL

Figure 17-9. *Using the function with* `OUTER APPLY` *(still successfully!)*

Any row that doesn't have a `NoteCount` now returns a `NULL` value in that column, but the column itself is returned. You can try this with the other tables. If you use `CROSS APPLY`, only rows with a record in the particular table will be returned; if you use `OUTER APPLY`, every row in `Contacts` will be returned, with a `NULL` value displayed in the appropriate column when no corresponding row exists.

Performance Issues

`APPLY` sounds great, and resolves a key issue for us: the ability to join record sets to TVFs. And if it is used correctly, `APPLY` can be an extremely elegant solution. But like most features in SQL Server, `APPLY` can drastically affect performance. Think about the function we wrote earlier; it contains four `SELECT` statements:

- `AddressCount`
- `NoteCount`
- `PhoneNumberCount`
- `RoleCount`

Now, think about what happens if we `CROSS APPLY` the `ContactCounts` function, using the `'All'` option

- All rows from the `Contacts` table are returned.
- For each row in the `Contacts` table, the `ContactCounts` function is executed. This means four `SELECT` statements are executed for every single row.
- The `Contacts` table in my database currently contains 22 rows, so that means we execute 22 × 4 `SELECT` statements = 88 `SELECT` statements.
- Imagine if the `Contacts` table contained 10,000 rows. That would be 40,000 `SELECT` statements executed!

There are better ways to implement the counts function we've seen here, but the solution we've implemented is perfectly valid, especially for the small record sets we're dealing with. We'd probably need to look at a different solution if our `Contacts` table drastically grew, though.

Summary

This chapter has covered some really interesting ground. Once we'd figured out what a table-valued function is and how they can be used, we took a look at how to utilize them in queries. We now know how to join not only to tables and views, but also to TVFs, using the `APPLY` operator. The `APPLY` operator is very powerful and can be used for much more than just TVFs.

We did try to use joins with our TVFs, and found that while this is possible, the values for the parameters we need to pass to our TVF must be available before the TVF is called. For this reason, `APPLY` is generally used, rather than joins.

We've covered almost all programmatic aspects of T-SQL now, in terms of the objects available to us. We're now going to take a look at the most commonly used programmatic object in T-SQL: stored procedures. "Proceed" to the next chapter!

PREV
Chapter 16 : Functions

NEXT
Chapter 18 : Stored Procedures…