

Temporary Tables, Table Variables, and Table-Valued Functions in SQL Server

Burton Roberts



In this article, Burton Roberts shows how to use SQL Server's temporary tables in your Access Data Projects. With that out of the way, he'll introduce two powerful new features of SQL Server 2000.

WHEN you construct a complex query in Access/ Jet, it's sometimes useful to break the query up into simpler queries and combine them. Access and Jet allow you to both create queries that join and embed parameterized queries with other parameterized queries. When you run a query from the database window, you're prompted for the parameters of the embedded queries as well as the outer query. If you have compound queries like this in your Access/Jet database and you intend to upsize your database to use SQL Server 7, you're going to be disappointed: These queries don't translate to the new environment.

In SQL Server, Jet queries are replaced by views and stored procedures. You can create a view using the graphical designer, which looks similar to the query designer in Access/Jet. You can join views together and embed them within other views. Views, however, don't take parameters. In order to add parameters to your view, you must embed the view in a stored procedure. A simple example would look like this procedure, which embeds a SQL query in a stored procedure:

```
CREATE PROCEDURE procMyProcedure
@intParam int
AS
SET NOCOUNT ON
SELECT column1, column2
FROM vwMyView
WHERE column3 = @intParam
RETURN
```

When you run `procMyProcedure` from the Access Project container, you'll be prompted for the integer parameter, `@intParam`, and then the query will run.

You can also join views in your stored procedure and add parameters like this:

```
CREATE PROCEDURE procMyProcedure1
@intParam1 int,
```

```
@intParam2 int
AS
SET NOCOUNT ON
SELECT a.column1, a.column2, b.column1, b.column2
FROM vwMyView a
INNER JOIN vwMyOtherView b
ON a.column3 = b.column3
WHERE a.column1 = @intParam1
AND b.column1 = @intParam2
RETURN
```

A problem arises if you want to parameterize the queries *before* you join them together within another query. You can do this in Access/Jet, but in SQL Server 7 you must use local temporary tables.

Temporary tables

Local temporary tables are created in stored procedures and are prefixed with a single pound sign (for example, `#tblTemp`). To create a local temporary table, you use the `CREATE TABLE` command like this:

```
CREATE TABLE #TempA
(column1 int, column2 varchar(50), column3 int)
```

SQL Server doesn't store these tables in the application's file, but in the `tempdb` file on the server. As a result, temporary tables never appear in the tables section of the project database container. Many local temporary tables of the same name may exist in the `tempdb` file and be used simultaneously without conflict. SQL Server creates a new instance of the local temporary table each time one is created and adds a unique suffix that reserves that instance of the table for use by the client that created it. Once the stored procedure is finished, the temporary table is automatically destroyed.

The following stored procedure fills a couple of temporary tables with pre-filtered data and then joins them to produce the same result as we'd get in the previous example:

```
CREATE PROCEDURE procMyProcedure2
@intParam1 int,
@intParam2 int,
AS
SET NOCOUNT ON
```

```

-- Create the 1st temp table
CREATE TABLE #TempA
(column1 int, column2 varchar(50), column3 int)

--Fill it with filtered records
INSERT #TempA (column1, column2, column3)
  SELECT column1, column2, column3
  FROM vwMyView
  WHERE column1 = @intParam1

--Create 2nd temp table
CREATE Table #TempB
(column1 int, column2 varchar(50), column3 int)

--Fill it with filtered records
INSERT #TempB (column1, column2, column3)
  SELECT column1, column2, column3
  FROM vwMyOtherView
  WHERE column1 = @intParam2

-- Select records from the join of two temp tables
SELECT a.column1, a.column2, b.column1, b.column2
  FROM #TempA a
  INNER join #TempB b
  ON a.column3 = b.column3

RETURN

```

Chances are that in a contrived example like this one, you wouldn't see any performance benefit from using temporary tables. However, there are some specialized occasions when you'll get a significant performance boost. The more complex your query, the greater chance that employing temporary tables in this manner will improve performance. It's something you have to test.

Notice that I use the command SET NOCOUNT ON in every stored procedure. It's important that you include this command in any stored procedure that returns data from temporary tables. If you don't use SET NOCOUNT ON, temporary table data won't return to your Access Data Project. It's considered good practice to use SET NOCOUNT ON in all of your stored procedures unless you have a reason to use SET NOCOUNT OFF (the default). SET NOCOUNT ON eliminates SQL Server messages that report the count of rows affected by each action of the stored procedure. These messages, called DONE_IN_PROC messages, are sent to SQL Server client utilities like Query Analyzer. Eliminating these messages saves resources, improves performance, and allows you to return temporary table data to your Access client.

Table variables

Microsoft introduced the new table data type, or table variable, in SQL Server 2000. The table variable can be used like a temporary table. To use a table variable in a stored procedure, you first declare a variable as the table data type like this:

```

DECLARE @tblVarA TABLE
(column1 int, column2 varchar(50),
 column3 int)

```

Now you can fill the table variable in the same way that you would a temporary table. Here's a stored

procedure that uses table variables instead of temporary tables. It looks very similar to my original code:

```

CREATE PROCEDURE procMyProcedure3
@intParam1 int,
@intParam2 int,
AS
SET NOCOUNT ON

-- Declare the 1st table variable
DECLARE @tblVarA Table
(column1 int, column2 varchar(50), column3 int)

--Fill it with filtered records
INSERT @tblVarA (column1, column2, column3)
SELECT column1, column2, column3
  FROM vwMyView
  WHERE column1 = @intParam1

--Declare the 2nd table variable
DECLARE @tblVarB Table
(column1 int, column2 varchar(50), column3 int)

--Fill it with filtered records
INSERT @tblVarB (column1, column2, column3)
SELECT column1, column2, column3
  FROM vwMyOtherView
  WHERE column1 = @intParam2

-- Select records from the join of two table variables
SELECT a.column1, a.column2, b.column1, b.column2
  FROM @tblVarA a
  INNER JOIN @tblVarB b
  ON a.column3 = b.column3

RETURN

```

I don't need to use SET NOCOUNT ON to return records this time because I'm not returning temporary table data. SET NOCOUNT ON isn't required when returning data from table variables. I use it anyway, though, because it's good programming practice. Unlike temporary tables, table variables don't use the tempdb database. They're in-memory objects like any other variables that you might declare and use in a stored procedure. Microsoft recommends that you use table variables instead of temporary tables in SQL Server 2000.

Table-valued functions

A big advantage of the table variable over the temporary table is that you can pass a table variable to a stored procedure from a SQL Server 2000 user-defined function. User-defined functions (UDFs) are also new in SQL Server 2000. There are three types of UDFs: the scalar-valued function, which returns a single value (like a VBA function); the inline table-valued function, which returns a table variable; and the multi-statement table-valued function, which also returns a table variable.

I could write an entire article on each of these function-types. I introduced *Smart Access* readers to scalar-valued functions in the April 2001 issue. In this article, the relevant function type is the inline table-valued function. It's equivalent to a parameterized view, or a parameterized Jet query. The first inline

table-valued function that I need for my example looks like this:

```
CREATE FUNCTION tvfTableA (@intParam int)
RETURNS TABLE
RETURN
(SELECT column1, column2, column3
 FROM vwMyView
 WHERE column1 = @intParam)
```

Here's the other one:

```
CREATE FUNCTION tvfTableB (@intParam int)
RETURNS TABLE
RETURN
(SELECT column1, column2, column3
 FROM vwMyOtherView
 WHERE column1 = @intParam)
```

Now I combine these functions in a single SELECT statement in a stored procedure to get the results that I want:

```
CREATE PROCEDURE procMyProcedure4
@intParam1 int,
@intParam2 int
AS
SET NOCOUNT ON
SELECT a.column1, a.column2, b.column1, b.column2
FROM dbo.tvfTableA(@intParam1) a
INNER JOIN dbo.tvfTableB(@intParam2) b
```

```
ON a.column3 = b.column3
```

Notice that I must qualify my call to the UDF with an owner designation (in this case, "dbo"). Some SQL programmers prefer to keep their UDFs in a separate database, in which case you'd have to include the name of that database in your call to the UDF. (For instance, "udfs.dbo.tvfTableA" refers to a UDF owned by dbo and kept in the udfs database.)

In this article, I've demonstrated three ways to emulate complex joins of parameterized Jet queries in SQL Server. If you're using SQL Server 7 or earlier, you're limited to using temporary tables. If you're using SQL Server 2000, you can use the new table data type and table-valued functions. In any case, don't forget to explicitly SET NOCOUNT ON. ▲



[TMPTABLE.ZIP at www.smartaccessnewsletter.com](http://www.smartaccessnewsletter.com)

Burton Roberts and his wife, Jennifer, are principals in Extended Day Services, a provider of school-aged childcare for school districts in the South Hills of Pittsburgh, PA. As an independent consultant, Burton has authored applications for businesses in occupational health and specialty steel. Burton is an Access MCP and has an MBA from Carnegie Mellon University.