Table-Valued Parameters

10/12/2018 • 8 minutes to read •

Table-valued parameters provide an easy way to marshal multiple rows of data from a client application to SQL Server without requiring multiple round trips or special server-side logic for processing the data. You can use table-valued parameters to encapsulate rows of data in a client application and send the data to the server in a single parameterized command. The incoming data rows are stored in a table variable that can then be operated on by using Transact-SQL.

Column values in table-valued parameters can be accessed using standard Transact-SQL SELECT statements. Table-valued parameters are strongly typed and their structure is automatically validated. The size of table-valued parameters is limited only by server memory.

> **Note**
>
> You cannot return data in a table-valued parameter. Table-valued parameters are input-only; the OUTPUT keyword is not supported.

For more information about table-valued parameters, see the following resources.

Resource	Description
Use Table-Valued Parameters (Database Engine)	Describes how to create and use table-valued parameters.
User-Defined Table Types	Describes user-defined table types that are used to declare table-valued parameters.

Passing Multiple Rows in Previous Versions of SQL Server

Before table-valued parameters were introduced to SQL Server 2008, the options for passing multiple rows of data to a stored procedure or a parameterized SQL command were limited. A developer could choose from the following options for passing multiple rows to the server:

- Use a series of individual parameters to represent the values in multiple columns and rows of data. The amount of data that can be passed by using this method is limited by the number of parameters allowed. SQL Server procedures can have, at most, 2100 parameters. Server-side logic is required to assemble these individual values into a table variable or a temporary table for processing.
- Bundle multiple data values into delimited strings or XML documents and then pass those text values to a procedure or statement. This requires the procedure or logic necessary for validating the data structures and unbundling the values.
- Create a series of individual SQL statements for data modifications that affect multiple rows, such as those created by calling the `Update` method of a SqlDataAdapter. Changes can be submitted to the server individually or batched into groups. However, even when changes are batched, this can mean that many statements, each executed separately on the server.
- Use the `bcp` utility program or the SqlBulkCopy object to load many rows of data into a table. Although this technique is very efficient, it does not support server-side processing unless the data is loaded into a temporary table or table variable.

Creating Table-Valued Parameter Types

Table-valued parameters are based on *strongly typed* table structures that are defined by using Transact-SQL CREATE TYPE statements. You have to create a table type and define the structure in SQL Server before you can use table-valued parameters in your client applications. For more information about creating table types, see User-Defined Table Types.

The following statement creates a table type named CategoryTableType that consists of CategoryID and CategoryName data types:

```
SQL                                                                    Copy

CREATE TYPE dbo.CategoryTableType AS TABLE
    ( CategoryID int, CategoryName nvarchar(50) )
```

After you create a table type, you can declare table-valued parameters based on that type. The following Transact-SQL fragment demonstrates how to declare a table-valued parameter in a stored procedure definition. Note that the READONLY keyword is required for declaring a table-valued parameter.

```
SQL                                                                    Copy

CREATE PROCEDURE usp_UpdateCategories
    (@tvpNewCategories dbo.CategoryTableType READONLY)
```

Modifying Data with Table-Valued Parameters (Transact-SQL)

Table-valued parameters can be used in set-based data modifications that affect multiple rows by executing a single statement. For example, you can select all the rows in a table-valued parameter and insert them into a database table, or you can create an update statement by joining a table-valued parameter to the table you want to update.

The following Transact-SQL UPDATE statement demonstrates how to use a table-valued parameter by joining it to the Categories table. When you use a table-valued parameter with a JOIN in a FROM clause, you must also alias it, as shown here, where the table-valued parameter is aliased as "ec":

```
SQL                                                                    Copy

UPDATE dbo.Categories
    SET Categories.CategoryName = ec.CategoryName
    FROM dbo.Categories INNER JOIN @tvpEditedCategories AS ec
    ON dbo.Categories.CategoryID = ec.CategoryID;
```

This Transact-SQL example demonstrates how to select rows from a table-valued parameter to perform an INSERT in a single set-based operation.

```
SQL                                                                    Copy

INSERT INTO dbo.Categories (CategoryID, CategoryName)
    SELECT nc.CategoryID, nc.CategoryName FROM @tvpNewCategories AS nc;
```

Limitations of Table-Valued Parameters

There are several limitations to table-valued parameters:

- You cannot pass table-valued parameters to CLR user-defined functions.
- Table-valued parameters can only be indexed to support UNIQUE or PRIMARY KEY constraints. SQL Server does not maintain statistics on table-valued parameters.
- Table-valued parameters are read-only in Transact-SQL code. You cannot update the column values in the rows of a table-valued parameter and you cannot insert or delete rows. To modify the data that is passed to a stored procedure or parameterized statement in table-valued parameter, you must insert the data into a temporary table or into a table variable.
- You cannot use ALTER TABLE statements to modify the design of table-valued parameters.

Configuring a SqlParameter Example

System.Data.SqlClient supports populating table-valued parameters from DataTable, DbDataReader or IEnumerable<T> SqlDataRecord objects. You must specify a type name for the table-valued parameter by using the TypeName property of a SqlParameter. The `TypeName` must match the name of a compatible type previously created on the server. The following code fragment demonstrates how to configure SqlParameter to insert data.

In the following example, the `addedCategories` variable contains a DataTable. To see how the variable is populated, see the examples in the next section, Passing a Table-Valued Parameter to a Stored Procedure.

```
C#                                                                     Copy

// Configure the command and parameter.
SqlCommand insertCommand = new SqlCommand(sqlInsert, connection);
SqlParameter tvpParam = insertCommand.Parameters.AddWithValue("@tvpNewCategories",
    addedCategories);
tvpParam.SqlDbType = SqlDbType.Structured;
tvpParam.TypeName = "dbo.CategoryTableType";
```

You can also use any object derived from DbDataReader to stream rows of data to a table-valued parameter, as shown in this fragment:

```
C#                                                                     Copy

// Configure the SqlCommand and table-valued parameter.
SqlCommand insertCommand = new SqlCommand("usp_InsertCategories", connection);
insertCommand.CommandType = CommandType.StoredProcedure;
SqlParameter tvpParam = insertCommand.Parameters.AddWithValue("@tvpNewCategories",
    dataReader);
tvpParam.SqlDbType = SqlDbType.Structured;
```

Passing a Table-Valued Parameter to a Stored Procedure

This example demonstrates how to pass table-valued parameter data to a stored procedure. The code extracts added rows into a new DataTable by using the GetChanges method. The code then defines a SqlCommand, setting the CommandType property to StoredProcedure. The SqlParameter is populated by using the AddWithValue method and the SqlDbType is set to Structured. The SqlCommand is then executed by using the ExecuteNonQuery method.

```
C#                                                                     Copy

// Assumes connection is an open SqlConnection object.
using (connection)
{
    // Create a DataTable with the modified rows.
    DataTable addedCategories = CategoryDataTable.GetChanges(DataRowState.Added);

    // Configure the SqlCommand and SqlParameter.
    SqlCommand insertCommand = new SqlCommand("usp_InsertCategories", connection);
    insertCommand.CommandType = CommandType.StoredProcedure;
    SqlParameter tvpParam = insertCommand.Parameters.AddWithValue("@tvpNewCategories",
    tvpParam.SqlDbType = SqlDbType.Structured;

    // Execute the command.
    insertCommand.ExecuteNonQuery();
}
```

Passing a Table-Valued Parameter to a Parameterized SQL Statement

The following example demonstrates how to insert data into the dbo.Categories table by using an INSERT statement with a SELECT subquery that has a table-valued parameter as the data source. When passing a table-valued parameter to a parameterized SQL statement, you must specify a type name for the table-valued parameter by using the new TypeName property of a SqlParameter. This `TypeName` must match the name of a compatible type previously created on the server. The code in this example uses the `TypeName` property to reference the type structure defined in dbo.CategoryTableType.

> **Note**
>
> If you supply a value for an identity column in a table-valued parameter, you must issue the SET IDENTITY_INSERT statement for the session.

```
C#                                                                     Copy

// Assumes connection is an open SqlConnection.
using (connection)
{
    // Create a DataTable with the modified rows.
    DataTable addedCategories = CategoryDataTable.GetChanges(DataRowState.Added);

    // Define the INSERT-SELECT statement.
    string sqlInsert =
        "INSERT INTO dbo.Categories (CategoryID, CategoryName)"
        + " SELECT nc.CategoryID, nc.CategoryName"
        + " FROM @tvpNewCategories AS nc;";

    // Configure the command and parameter.
    SqlCommand insertCommand = new SqlCommand(sqlInsert, connection);
    SqlParameter tvpParam = insertCommand.Parameters.AddWithValue("@tvpNewCategories",
    tvpParam.SqlDbType = SqlDbType.Structured;
    tvpParam.TypeName = "dbo.CategoryTableType";

    // Execute the command.
    insertCommand.ExecuteNonQuery();
}
```

Streaming Rows with a DataReader

You can also use any object derived from DbDataReader to stream rows of data to a table-valued parameter. The following code fragment demonstrates retrieving data from an Oracle database by using an OracleCommand and an OracleDataReader. The code then configures a SqlCommand to invoke a stored procedure with a single input parameter. The SqlDbType property of the SqlParameter is set to `Structured`. The AddWithValue passes the `OracleDataReader` result set to the stored procedure as a table-valued parameter.

```
C#                                                                     Copy

// Assumes connection is an open SqlConnection.
// Retrieve data from Oracle.
OracleCommand selectCommand = new OracleCommand(
    "Select CategoryID, CategoryName FROM Categories;",
    oracleConnection);
OracleDataReader oracleReader = selectCommand.ExecuteReader(
    CommandBehavior.CloseConnection);

// Configure the SqlCommand and table-valued parameter.
SqlCommand insertCommand = new SqlCommand(
    "usp_InsertCategories", connection);
insertCommand.CommandType = CommandType.StoredProcedure;
SqlParameter tvpParam =
    insertCommand.Parameters.AddWithValue(
    "@tvpNewCategories", oracleReader);
tvpParam.SqlDbType = SqlDbType.Structured;

// Execute the command.
insertCommand.ExecuteNonQuery();
```

See also

- Configuring Parameters and Parameter Data Types
- Commands and Parameters
- DataAdapter Parameters
- SQL Server Data Operations in ADO.NET
- ADO.NET Overview